****
183184 Mon Jul 28 07:44:13 2014
new/usr/src/uts/common/crypto/io/crypto.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
 617 static crypto_minor_t *
 618 crypto_hold_minor(minor_t minor)
 619 {
 620         crypto_minor_t *cm;
 621         kcf_lock_withpad_t *mp;

 623         if (minor > crypto_minors_table_count)
 624                 return (NULL);

 626         mp = &crypto_locks[CPU_SEQID];
 627         mutex_enter(&mp->kl_lock);

 629         if ((cm = crypto_minors[minor - 1]) != NULL) {
 630                 atomic_inc_32(&cm->cm_refcnt);
 630                 atomic_add_32(&cm->cm_refcnt, 1);
 631         }
 632         mutex_exit(&mp->kl_lock);
 633         return (cm);
 634 }

 636 static void
 637 crypto_release_minor(crypto_minor_t *cm)
 638 {
 639         if (atomic_dec_32_nv(&cm->cm_refcnt) == 0) {
 639         if (atomic_add_32_nv(&cm->cm_refcnt, -1) == 0) {
 640                 cv_signal(&cm->cm_cv);
 641         }
 642 }
```

_____***unchanged_portion_omitted_***

****
298794 Mon Jul 28 07:44:13 2014
new/usr/src/uts/common/crypto/io/dprov.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
1099 /*
1100  * If a session has a reference to a dprov_object_t,
1101  * it REFHOLD()s.
1102  */
1103 #define DPROV_OBJECT_REFHOLD(object) {          \
1104         atomic_inc_32(&(object)->do_refcnt);    \
1104         atomic_add_32(&(object)->do_refcnt, 1); \
1105         ASSERT((object)->do_refcnt != 0);               \
1106 }

1108 /*
1109  * Releases a reference to an object. When the last
1110  * reference is released, the object is freed.
1111  */
1112 #define DPROV_OBJECT_REFRELE(object) {                          \
1113         ASSERT((object)->do_refcnt != 0);                      \
1114         membar_exit();                                         \
1115         if (atomic_dec_32_nv(&(object)->do_refcnt) == 0)       \
1115         if (atomic_add_32_nv(&(object)->do_refcnt, -1) == 0)   \
1116                 dprov_free_object(object);                     \
1117 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   52301 Mon Jul 28 07:44:13 2014
new/usr/src/uts/common/disp/cmt.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1353 /* ARGSUSED */
1354 static void
1355 cmt_ev_thread_swtch(pg_t *pg, cpu_t *cp, hrtime_t now, kthread_t *old,
1356                     kthread_t *new)
1357 {
1358         pg_cmt_t        *cmt_pg = (pg_cmt_t *)pg;

1360         if (old == cp->cpu_idle_thread) {
1361                 atomic_inc_32(&cmt_pg->cmt_utilization);
1361                 atomic_add_32(&cmt_pg->cmt_utilization, 1);
1362         } else if (new == cp->cpu_idle_thread) {
1363                 atomic_dec_32(&cmt_pg->cmt_utilization);
1363                 atomic_add_32(&cmt_pg->cmt_utilization, -1);
1364         }
1365 }

1367 /*
1368  * Macro to test whether a thread is currently runnable on a CPU in a PG.
1369  */
1370 #define THREAD_RUNNABLE_IN_PG(t, pg)                                    \
1371         ((t)->t_state == TS_RUN &&                                      \
1372             (t)->t_disp_queue->disp_cpu &&                              \
1373             bitset_in_set(&(pg)->cmt_cpus_actv_set,                     \
1374             (t)->t_disp_queue->disp_cpu->cpu_seqid))

1376 static void
1377 cmt_ev_thread_swtch_pwr(pg_t *pg, cpu_t *cp, hrtime_t now, kthread_t *old,
1378     kthread_t *new)
1379 {
1380         pg_cmt_t        *cmt = (pg_cmt_t *)pg;
1381         cpupm_domain_t  *dom;
1382         uint32_t        u;

1384         if (old == cp->cpu_idle_thread) {
1385                 ASSERT(new != cp->cpu_idle_thread);
1386                 u = atomic_inc_32_nv(&cmt->cmt_utilization);
1386                 u = atomic_add_32_nv(&cmt->cmt_utilization, 1);
1387                 if (u == 1) {
1388                         /*
1389                          * Notify the CPU power manager that the domain
1390                          * is non-idle.
1391                          */
1392                         dom = (cpupm_domain_t *)cmt->cmt_pg.pghw_handle;
1393                         cpupm_utilization_event(cp, now, dom,
1394                             CPUPM_DOM_BUSY_FROM_IDLE);
1395                 }
1396         } else if (new == cp->cpu_idle_thread) {
1397                 ASSERT(old != cp->cpu_idle_thread);
1398                 u = atomic_dec_32_nv(&cmt->cmt_utilization);
1398                 u = atomic_add_32_nv(&cmt->cmt_utilization, -1);
1399                 if (u == 0) {
1400                         /*
1401                          * The domain is idle, notify the CPU power
1402                          * manager.
1403                          *
1404                          * Avoid notifying if the thread is simply migrating
1405                          * between CPUs in the domain.
1406                          */
1407                         if (!THREAD_RUNNABLE_IN_PG(old, cmt)) {
```



```
1408                                 dom = (cpupm_domain_t *)cmt->cmt_pg.pghw_handle;
1409                                 cpupm_utilization_event(cp, now, dom,
1410                                     CPUPM_DOM_IDLE_FROM_BUSY);
1411                         }
1412                 }
1413         }
1414 }
______unchanged_portion_omitted_
```

****
62745 Mon Jul 28 07:44:14 2014
new/usr/src/uts/common/dtrace/fasttrap.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____unchanged_portion_omitted_

```
1171 static fasttrap_proc_t *
1172 fasttrap_proc_lookup(pid_t pid)
1173 {
1174         fasttrap_bucket_t *bucket;
1175         fasttrap_proc_t *fprc, *new_fprc;

1177         bucket = &fasttrap_procs.fth_table[FASTTRAP_PROCS_INDEX(pid)];
1178         mutex_enter(&bucket->ftb_mtx);

1180         for (fprc = bucket->ftb_data; fprc != NULL; fprc = fprc->ftpc_next) {
1181                 if (fprc->ftpc_pid == pid && fprc->ftpc_acount != 0) {
1182                         mutex_enter(&fprc->ftpc_mtx);
1183                         mutex_exit(&bucket->ftb_mtx);
1184                         fprc->ftpc_rcount++;
1185                         atomic_inc_64(&fprc->ftpc_acount);
1185                         atomic_add_64(&fprc->ftpc_acount, 1);
1186                         ASSERT(fprc->ftpc_acount <= fprc->ftpc_rcount);
1187                         mutex_exit(&fprc->ftpc_mtx);

1189                         return (fprc);
1190                 }
1191         }

1193         /*
1194          * Drop the bucket lock so we don't try to perform a sleeping
1195          * allocation under it.
1196          */
1197         mutex_exit(&bucket->ftb_mtx);

1199         new_fprc = kmem_zalloc(sizeof (fasttrap_proc_t), KM_SLEEP);
1200         new_fprc->ftpc_pid = pid;
1201         new_fprc->ftpc_rcount = 1;
1202         new_fprc->ftpc_acount = 1;

1204         mutex_enter(&bucket->ftb_mtx);

1206         /*
1207          * Take another lap through the list to make sure a proc hasn't
1208          * been created for this pid while we weren't under the bucket lock.
1209          */
1210         for (fprc = bucket->ftb_data; fprc != NULL; fprc = fprc->ftpc_next) {
1211                 if (fprc->ftpc_pid == pid && fprc->ftpc_acount != 0) {
1212                         mutex_enter(&fprc->ftpc_mtx);
1213                         mutex_exit(&bucket->ftb_mtx);
1214                         fprc->ftpc_rcount++;
1215                         atomic_inc_64(&fprc->ftpc_acount);
1215                         atomic_add_64(&fprc->ftpc_acount, 1);
1216                         ASSERT(fprc->ftpc_acount <= fprc->ftpc_rcount);
1217                         mutex_exit(&fprc->ftpc_mtx);

1219                         kmem_free(new_fprc, sizeof (fasttrap_proc_t));

1221                         return (fprc);
1222                 }
1223         }

1225         new_fprc->ftpc_next = bucket->ftb_data;
1226         bucket->ftb_data = new_fprc;
```



```
1228         mutex_exit(&bucket->ftb_mtx);

1230         return (new_fprc);
1231 }
```

_____unchanged_portion_omitted_

```
1408 static void
1409 fasttrap_provider_free(fasttrap_provider_t *provider)
1410 {
1411         pid_t pid = provider->ftp_pid;
1412         proc_t *p;

1414         /*
1415          * There need to be no associated enabled probes, no consumers
1416          * creating probes, and no meta providers referencing this provider.
1417          */
1418         ASSERT(provider->ftp_rcount == 0);
1419         ASSERT(provider->ftp_ccount == 0);
1420         ASSERT(provider->ftp_mcount == 0);

1422         /*
1423          * If this provider hasn't been retired, we need to explicitly drop the
1424          * count of active providers on the associated process structure.
1425          */
1426         if (!provider->ftp_retired) {
1427                 atomic_dec_64(&provider->ftp_proc->ftpc_acount);
1427                 atomic_add_64(&provider->ftp_proc->ftpc_acount, -1);
1428                 ASSERT(provider->ftp_proc->ftpc_acount <
1429                     provider->ftp_proc->ftpc_rcount);
1430         }

1432         fasttrap_proc_release(provider->ftp_proc);

1434         kmem_free(provider, sizeof (fasttrap_provider_t));

1436         /*
1437          * Decrement p_dtrace_probes on the process whose provider we're
1438          * freeing. We don't have to worry about clobbering somone else's
1439          * modifications to it because we have locked the bucket that
1440          * corresponds to this process's hash chain in the provider hash
1441          * table. Don't sweat it if we can't find the process.
1442          */
1443         mutex_enter(&pidlock);
1444         if ((p = prfind(pid)) == NULL) {
1445                 mutex_exit(&pidlock);
1446                 return;
1447         }

1449         mutex_enter(&p->p_lock);
1450         mutex_exit(&pidlock);

1452         p->p_dtrace_probes--;
1453         mutex_exit(&p->p_lock);
1454 }

1456 static void
1457 fasttrap_provider_retire(pid_t pid, const char *name, int mprov)
1458 {
1459         fasttrap_provider_t *fp;
1460         fasttrap_bucket_t *bucket;
1461         dtrace_provider_id_t provid;

1463         ASSERT(strlen(name) < sizeof (fp->ftp_name));

1465         bucket = &fasttrap_provs.fth_table[FASTTRAP_PROVS_INDEX(pid, name)];
1466         mutex_enter(&bucket->ftb_mtx);
```

```
1468         for (fp = bucket->ftb_data; fp != NULL; fp = fp->ftp_next) {
1469                 if (fp->ftp_pid == pid && strcmp(fp->ftp_name, name) == 0 &&
1470                     !fp->ftp_retired)
1471                         break;
1472         }

1474         if (fp == NULL) {
1475                 mutex_exit(&bucket->ftb_mtx);
1476                 return;
1477         }

1479         mutex_enter(&fp->ftp_mtx);
1480         ASSERT(!mprov || fp->ftp_mcount > 0);
1481         if (mprov && --fp->ftp_mcount != 0)  {
1482                 mutex_exit(&fp->ftp_mtx);
1483                 mutex_exit(&bucket->ftb_mtx);
1484                 return;
1485         }

1487         /*
1488          * Mark the provider to be removed in our post-processing step, mark it
1489          * retired, and drop the active count on its proc. Marking it indicates
1490          * that we should try to remove it; setting the retired flag indicates
1491          * that we're done with this provider; dropping the active the proc
1492          * releases our hold, and when this reaches zero (as it will during
1493          * exit or exec) the proc and associated providers become defunct.
1494          *
1495          * We obviously need to take the bucket lock before the provider lock
1496          * to perform the lookup, but we need to drop the provider lock
1497          * before calling into the DTrace framework since we acquire the
1498          * provider lock in callbacks invoked from the DTrace framework. The
1499          * bucket lock therefore protects the integrity of the provider hash
1500          * table.
1501          */
1502         atomic_dec_64(&fp->ftp_proc->ftpc_acount);
1502         atomic_add_64(&fp->ftp_proc->ftpc_acount, -1);
1503         ASSERT(fp->ftp_proc->ftpc_acount < fp->ftp_proc->ftpc_rcount);

1505         fp->ftp_retired = 1;
1506         fp->ftp_marked = 1;
1507         provid = fp->ftp_provid;
1508         mutex_exit(&fp->ftp_mtx);

1510         /*
1511          * We don't have to worry about invalidating the same provider twice
1512          * since fasttrap_provider_lookup() will ignore provider that have
1513          * been marked as retired.
1514          */
1515         dtrace_invalidate(provid);

1517         mutex_exit(&bucket->ftb_mtx);

1519         fasttrap_pid_cleanup();
1520 }
_____unchanged_portion_omitted_

1534 static int
1535 fasttrap_add_probe(fasttrap_probe_spec_t *pdata)
1536 {
1537         fasttrap_provider_t *provider;
1538         fasttrap_probe_t *pp;
1539         fasttrap_tracepoint_t *tp;
1540         char *name;
1541         int i, aframes, whack;
```



```
1543         /*
1544          * There needs to be at least one desired trace point.
1545          */
1546         if (pdata->ftps_noffs == 0)
1547                 return (EINVAL);

1549         switch (pdata->ftps_type) {
1550         case DTFTP_ENTRY:
1551                 name = "entry";
1552                 aframes = FASTTRAP_ENTRY_AFRAMES;
1553                 break;
1554         case DTFTP_RETURN:
1555                 name = "return";
1556                 aframes = FASTTRAP_RETURN_AFRAMES;
1557                 break;
1558         case DTFTP_OFFSETS:
1559                 name = NULL;
1560                 break;
1561         default:
1562                 return (EINVAL);
1563         }

1565         if ((provider = fasttrap_provider_lookup(pdata->ftps_pid,
1566             FASTTRAP_PID_NAME, &pid_attr)) == NULL)
1567                 return (ESRCH);

1569         /*
1570          * Increment this reference count to indicate that a consumer is
1571          * actively adding a new probe associated with this provider. This
1572          * prevents the provider from being deleted -- we'll need to check
1573          * for pending deletions when we drop this reference count.
1574          */
1575         provider->ftp_ccount++;
1576         mutex_exit(&provider->ftp_mtx);

1578         /*
1579          * Grab the creation lock to ensure consistency between calls to
1580          * dtrace_probe_lookup() and dtrace_probe_create() in the face of
1581          * other threads creating probes. We must drop the provider lock
1582          * before taking this lock to avoid a three-way deadlock with the
1583          * DTrace framework.
1584          */
1585         mutex_enter(&provider->ftp_cmtx);

1587         if (name == NULL) {
1588                 for (i = 0; i < pdata->ftps_noffs; i++) {
1589                         char name_str[17];

1591                         (void) sprintf(name_str, "%llx",
1592                             (unsigned long long)pdata->ftps_offs[i]);

1594                         if (dtrace_probe_lookup(provider->ftp_provid,
1595                             pdata->ftps_mod, pdata->ftps_func, name_str) != 0)
1596                                 continue;

1598                         atomic_inc_32(&fasttrap_total);
1598                         atomic_add_32(&fasttrap_total, 1);

1600                         if (fasttrap_total > fasttrap_max) {
1601                                 atomic_dec_32(&fasttrap_total);
1601                                 atomic_add_32(&fasttrap_total, -1);
1602                                 goto no_mem;
1603                         }

1605                         pp = kmem_zalloc(sizeof (fasttrap_probe_t), KM_SLEEP);
```

```
1607                        pp->ftp_prov = provider;
1608                        pp->ftp_faddr = pdata->ftps_pc;
1609                        pp->ftp_fsize = pdata->ftps_size;
1610                        pp->ftp_pid = pdata->ftps_pid;
1611                        pp->ftp_ntps = 1;

1613                        tp = kmem_zalloc(sizeof (fasttrap_tracepoint_t),
1614                            KM_SLEEP);

1616                        tp->ftt_proc = provider->ftp_proc;
1617                        tp->ftt_pc = pdata->ftps_offs[i] + pdata->ftps_pc;
1618                        tp->ftt_pid = pdata->ftps_pid;

1620                        pp->ftp_tps[0].fit_tp = tp;
1621                        pp->ftp_tps[0].fit_id.fti_probe = pp;
1622                        pp->ftp_tps[0].fit_id.fti_ptype = pdata->ftps_type;

1624                        pp->ftp_id = dtrace_probe_create(provider->ftp_provid,
1625                            pdata->ftps_mod, pdata->ftps_func, name_str,
1626                            FASTTRAP_OFFSET_AFRAMES, pp);
1627                }

1629        } else if (dtrace_probe_lookup(provider->ftp_provid, pdata->ftps_mod,
1630            pdata->ftps_func, name) == 0) {
1631                atomic_add_32(&fasttrap_total, pdata->ftps_noffs);

1633                if (fasttrap_total > fasttrap_max) {
1634                        atomic_add_32(&fasttrap_total, -pdata->ftps_noffs);
1635                        goto no_mem;
1636                }

1638                /*
1639                 * Make sure all tracepoint program counter values are unique.
1640                 * We later assume that each probe has exactly one tracepoint
1641                 * for a given pc.
1642                 */
1643                qsort(pdata->ftps_offs, pdata->ftps_noffs,
1644                    sizeof (uint64_t), fasttrap_uint64_cmp);
1645                for (i = 1; i < pdata->ftps_noffs; i++) {
1646                        if (pdata->ftps_offs[i] > pdata->ftps_offs[i - 1])
1647                                continue;

1649                        atomic_add_32(&fasttrap_total, -pdata->ftps_noffs);
1650                        goto no_mem;
1651                }

1653                ASSERT(pdata->ftps_noffs > 0);
1654                pp = kmem_zalloc(offsetof(fasttrap_probe_t,
1655                    ftp_tps[pdata->ftps_noffs]), KM_SLEEP);

1657                pp->ftp_prov = provider;
1658                pp->ftp_faddr = pdata->ftps_pc;
1659                pp->ftp_fsize = pdata->ftps_size;
1660                pp->ftp_pid = pdata->ftps_pid;
1661                pp->ftp_ntps = pdata->ftps_noffs;

1663                for (i = 0; i < pdata->ftps_noffs; i++) {
1664                        tp = kmem_zalloc(sizeof (fasttrap_tracepoint_t),
1665                            KM_SLEEP);

1667                        tp->ftt_proc = provider->ftp_proc;
1668                        tp->ftt_pc = pdata->ftps_offs[i] + pdata->ftps_pc;
1669                        tp->ftt_pid = pdata->ftps_pid;

1671                        pp->ftp_tps[i].fit_tp = tp;
1672                        pp->ftp_tps[i].fit_id.fti_probe = pp;
```



```
1673                        pp->ftp_tps[i].fit_id.fti_ptype = pdata->ftps_type;
1674                }

1676                pp->ftp_id = dtrace_probe_create(provider->ftp_provid,
1677                    pdata->ftps_mod, pdata->ftps_func, name, aframes, pp);
1678        }

1680        mutex_exit(&provider->ftp_cmtx);

1682        /*
1683         * We know that the provider is still valid since we incremented the
1684         * creation reference count. If someone tried to clean up this provider
1685         * while we were using it (e.g. because the process called exec(2) or
1686         * exit(2)), take note of that and try to clean it up now.
1687         */
1688        mutex_enter(&provider->ftp_mtx);
1689        provider->ftp_ccount--;
1690        whack = provider->ftp_retired;
1691        mutex_exit(&provider->ftp_mtx);

1693        if (whack)
1694                fasttrap_pid_cleanup();

1696        return (0);

1698 no_mem:
1699        /*
1700         * If we've exhausted the allowable resources, we'll try to remove
1701         * this provider to free some up. This is to cover the case where
1702         * the user has accidentally created many more probes than was
1703         * intended (e.g. pid123:::).
1704         */
1705        mutex_exit(&provider->ftp_cmtx);
1706        mutex_enter(&provider->ftp_mtx);
1707        provider->ftp_ccount--;
1708        provider->ftp_marked = 1;
1709        mutex_exit(&provider->ftp_mtx);

1711        fasttrap_pid_cleanup();

1713        return (ENOMEM);
1714 }
______unchanged_portion_omitted_
```

```
**********************************************************
   13803 Mon Jul 28 07:44:14 2014
new/usr/src/uts/common/dtrace/profile.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 159 static void
 160 profile_create(hrtime_t interval, const char *name, int kind)
 161 {
 162         profile_probe_t *prof;
 163         int nr_frames = PROF_ARTIFICIAL_FRAMES + dtrace_mach_aframes();

 165         if (profile_aframes)
 166                 nr_frames = profile_aframes;

 168         if (interval < profile_interval_min)
 169                 return;

 171         if (dtrace_probe_lookup(profile_id, NULL, NULL, name) != 0)
 172                 return;

 174         atomic_inc_32(&profile_total);
 174         atomic_add_32(&profile_total, 1);
 175         if (profile_total > profile_max) {
 176                 atomic_dec_32(&profile_total);
 176                 atomic_add_32(&profile_total, -1);
 177                 return;
 178         }

 180         prof = kmem_zalloc(sizeof (profile_probe_t), KM_SLEEP);
 181         (void) strcpy(prof->prof_name, name);
 182         prof->prof_interval = interval;
 183         prof->prof_cyclic = CYCLIC_NONE;
 184         prof->prof_kind = kind;
 185         prof->prof_id = dtrace_probe_create(profile_id,
 186             NULL, NULL, name, nr_frames, prof);
 187 }
______unchanged_portion_omitted_

 321 /*ARGSUSED*/
 322 static void
 323 profile_destroy(void *arg, dtrace_id_t id, void *parg)
 324 {
 325         profile_probe_t *prof = parg;

 327         ASSERT(prof->prof_cyclic == CYCLIC_NONE);
 328         kmem_free(prof, sizeof (profile_probe_t));

 330         ASSERT(profile_total >= 1);
 331         atomic_dec_32(&profile_total);
 331         atomic_add_32(&profile_total, -1);
 332 }
______unchanged_portion_omitted_
```

****
12421 Mon Jul 28 07:44:14 2014
new/usr/src/uts/common/fs/ctfs/ctfs_root.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
213 /*
214  * ctfs_mount - the VFS_MOUNT entry point
215  */
216 static int
217 ctfs_mount(vfs_t *vfsp, vnode_t *mvp, struct mounta *uap, cred_t *cr)
218 {
219         ctfs_vfs_t *data;
220         dev_t dev;
221         gfs_dirent_t *dirent;
222         int i;

224         if (secpolicy_fs_mount(cr, mvp, vfsp) != 0)
225                 return (EPERM);

227         if (mvp->v_type != VDIR)
228                 return (ENOTDIR);

230         if ((uap->flags & MS_OVERLAY) == 0 &&
231             (mvp->v_count > 1 || (mvp->v_flag & VROOT)))
232                 return (EBUSY);

234         data = kmem_alloc(sizeof (ctfs_vfs_t), KM_SLEEP);

236         /*
237          * Initialize vfs fields not initialized by VFS_INIT/domount
238          */
239         vfsp->vfs_bsize = DEV_BSIZE;
240         vfsp->vfs_fstype = ctfs_fstype;
241         do {
242                 dev = makedevice(ctfs_major,
243                     atomic_inc_32_nv(&ctfs_minor) & L_MAXMIN32);
243                     atomic_add_32_nv(&ctfs_minor, 1) & L_MAXMIN32);
244         } while (vfs_devismounted(dev));
245         vfs_make_fsid(&vfsp->vfs_fsid, dev, ctfs_fstype);
246         vfsp->vfs_data = data;
247         vfsp->vfs_dev = dev;

249         /*
250          * Dynamically create gfs_dirent_t array for the root directory.
251          */
252         dirent = kmem_zalloc((ct_ntypes + 2) * sizeof (gfs_dirent_t), KM_SLEEP);
253         for (i = 0; i < ct_ntypes; i++) {
254                 dirent[i].gfse_name = (char *)ct_types[i]->ct_type_name;
255                 dirent[i].gfse_ctor = ctfs_create_tdirnode;
256                 dirent[i].gfse_flags = GFS_CACHE_VNODE;
257         }
258         dirent[i].gfse_name = "all";
259         dirent[i].gfse_ctor = ctfs_create_adirnode;
260         dirent[i].gfse_flags = GFS_CACHE_VNODE;
261         dirent[i+1].gfse_name = NULL;

263         /*
264          * Create root vnode
265          */
266         data->ctvfs_root = gfs_root_create(sizeof (ctfs_rootnode_t),
267             vfsp, ctfs_ops_root, CTFS_INO_ROOT, dirent, ctfs_root_do_inode,
268             CTFS_NAME_MAX, NULL, NULL);

270         kmem_free(dirent, (ct_ntypes + 2) * sizeof (gfs_dirent_t));
```



```
272         return (0);
273 }
```
_____***unchanged_portion_omitted_***

```
**********************************************************
   52575 Mon Jul 28 07:44:14 2014
new/usr/src/uts/common/fs/dnlc.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 251 /*
 252  * Free an entry.
 253  */
 254 #define dnlc_free(ncp) \
 255 { \
 256         kmem_free((ncp), sizeof (ncache_t) + (ncp)->namlen); \
 257         atomic_dec_32(&dnlc_nentries); \
 257         atomic_add_32(&dnlc_nentries, -1); \
 258 }
______unchanged_portion_omitted_

1004 /*
1005  * Get a new name cache entry.
1006  * If the dnlc_reduce_cache() taskq isn't keeping up with demand, or memory
1007  * is short then just return NULL. If we're over ncsize then kick off a
1008  * thread to free some in use entries down to dnlc_nentries_low_water.
1009  * Caller must initialise all fields except namlen.
1010  * Component names are defined to be less than MAXNAMELEN
1011  * which includes a null.
1012  */
1013 static ncache_t *
1014 dnlc_get(uchar_t namlen)
1015 {
1016         ncache_t *ncp;

1018         if (dnlc_nentries > dnlc_max_nentries) {
1019                 dnlc_max_nentries_cnt++; /* keep a statistic */
1020                 return (NULL);
1021         }
1022         ncp = kmem_alloc(sizeof (ncache_t) + namlen, KM_NOSLEEP);
1023         if (ncp == NULL) {
1024                 return (NULL);
1025         }
1026         ncp->namlen = namlen;
1027         atomic_inc_32(&dnlc_nentries);
1027         atomic_add_32(&dnlc_nentries, 1);
1028         dnlc_reduce_cache(NULL);
1029         return (ncp);
1030 }
______unchanged_portion_omitted_
```

****
86998 Mon Jul 28 07:44:14 2014
new/usr/src/uts/common/fs/fem.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
 363 /*
 364  * Addref can only be called while its head->lock is held.
 365  */

 367 static void
 368 fem_addref(struct fem_list *sp)
 369 {
 370         atomic_inc_32(&sp->feml_refc);
 370         atomic_add_32(&sp->feml_refc, 1);
 371 }

 373 static uint32_t
 374 fem_delref(struct fem_list *sp)
 375 {
 376         return (atomic_dec_32_nv(&sp->feml_refc));
 376         return (atomic_add_32_nv(&sp->feml_refc, -1));
 377 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   20356 Mon Jul 28 07:44:15 2014
new/usr/src/uts/common/fs/lofs/lofs_subr.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

/*
 * The idea behind composition-based stacked filesystems is to add a
 * vnode to the stack of vnodes for each mount. These vnodes have their
 * own set of mount options and filesystem-specific functions, so they
 * can modify data or operations before they are passed along. Such a
 * filesystem must maintain a mapping from the underlying vnodes to its
 * interposing vnodes.
 *
 * In lofs, this mapping is implemented by a hashtable. Each bucket
 * contains a count of the number of nodes currently contained, the
 * chain of vnodes, and a lock to protect the list of vnodes. The
 * hashtable dynamically grows if the number of vnodes in the table as a
 * whole exceeds the size of the table left-shifted by
 * lo_resize_threshold. In order to minimize lock contention, there is
 * no global lock protecting the hashtable, hence obtaining the
 * per-bucket locks consists of a dance to make sure we've actually
 * locked the correct bucket. Acquiring a bucket lock doesn't involve
 * locking the hashtable itself, so we refrain from freeing old
 * hashtables, and store them in a linked list of retired hashtables;
 * the list is freed when the filesystem is unmounted.
 */

#include <sys/param.h>
#include <sys/kmem.h>
#include <sys/vfs.h>
#include <sys/vnode.h>
#include <sys/cmn_err.h>
#include <sys/systm.h>
#include <sys/t_lock.h>
#include <sys/debug.h>
#include <sys/atomic.h>

#include <sys/fs/lofs_node.h>
#include <sys/fs/lofs_info.h>
```



```
/*
 * Due to the hashing algorithm, the size of the hash table needs to be a
 * power of 2.
 */
#define LOFS_DEFAULT_HTSIZE     (1 << 6)

#define ltablehash(vp, tblsz)   ((((intptr_t)(vp))>>10) & ((tblsz)-1))

/*
 * The following macros can only be safely used when the desired bucket
 * is already locked.
 */
/*
 * The lock in the hashtable associated with the given vnode.
 */
#define TABLE_LOCK(vp, li)      \
        (&(li)->li_hashtable[ltablehash((vp), (li)->li_htsize)].lh_lock)

/*
 * The bucket in the hashtable that the given vnode hashes to.
 */
#define TABLE_BUCKET(vp, li)    \
        ((li)->li_hashtable[ltablehash((vp), (li)->li_htsize)].lh_chain)

/*
 * Number of elements currently in the bucket that the vnode hashes to.
 */
#define TABLE_COUNT(vp, li)     \
        ((li)->li_hashtable[ltablehash((vp), (li)->li_htsize)].lh_count)

/*
 * Grab/Drop the lock for the bucket this vnode hashes to.
 */
#define TABLE_LOCK_ENTER(vp, li)        table_lock_enter(vp, li)
#define TABLE_LOCK_EXIT(vp, li)         \
        mutex_exit(&(li)->li_hashtable[ltablehash((vp), \
            (li)->li_htsize)].lh_lock)

static lnode_t *lfind(struct vnode *, struct loinfo *);
static void lsave(lnode_t *, struct loinfo *);
static struct vfs *makelfsnode(struct vfs *, struct loinfo *);
static struct lfsnode *lfsfind(struct vfs *, struct loinfo *);

uint_t lo_resize_threshold = 1;
uint_t lo_resize_factor = 2;

static kmem_cache_t *lnode_cache;

/*
 * Since the hashtable itself isn't protected by a lock, obtaining a
 * per-bucket lock proceeds as follows:
 *
 * (a) li->li_htlock protects li->li_hashtable, li->li_htsize, and
 * li->li_retired.
 *
 * (b) Per-bucket locks (lh_lock) protect the contents of the bucket.
 *
 * (c) Locking order for resizing the hashtable is li_htlock then
 * lh_lock.
 *
 * To grab the bucket lock we:
 *
 * (1) Stash away the htsize and the pointer to the hashtable to make
 * sure neither change while we're using them.
 *
 * (2) lgrow() updates the pointer to the hashtable before it updates
```

```
126  * the size: the worst case scenario is that we have the wrong size (but
127  * the correct table), so we hash to the wrong bucket, grab the wrong
128  * lock, and then realize that things have changed, rewind and start
129  * again. If both the size and the table changed since we loaded them,
130  * we'll realize that too and restart.
131  *
132  * (3) The protocol for growing the hashtable involves holding *all* the
133  * locks in the table, hence the unlocking code (TABLE_LOCK_EXIT())
134  * doesn't need to do any dances, since neither the table nor the size
135  * can change while any bucket lock is held.
136  *
137  * (4) If the hashtable is growing (by thread t1) while another thread
138  * (t2) is trying to grab a bucket lock, t2 might have a stale reference
139  * to li->li_htsize:
140  *
141  * - t1 grabs all locks in lgrow()
142  *      - t2 loads li->li_htsize and li->li_hashtable
143  * - t1 changes li->hashtable
144  *      - t2 loads from an offset in the "stale" hashtable and tries to grab
145  *      the relevant mutex.
146  *
147  * If t1 had free'd the stale hashtable, t2 would be in trouble. Hence,
148  * stale hashtables are not freed but stored in a list of "retired"
149  * hashtables, which is emptied when the filesystem is unmounted.
150  */
151 static void
152 table_lock_enter(vnode_t *vp, struct loinfo *li)
153 {
154         struct lobucket *chain;
155         uint_t htsize;
156         uint_t hash;

158         for (;;) {
159                 htsize = li->li_htsize;
160                 membar_consumer();
161                 chain = (struct lobucket *)li->li_hashtable;
162                 hash = ltablehash(vp, htsize);
163                 mutex_enter(&chain[hash].lh_lock);
164                 if (li->li_hashtable == chain && li->li_htsize == htsize)
165                         break;
166                 mutex_exit(&chain[hash].lh_lock);
167         }
168 }
```

*unchanged_portion_omitted_*

```
237 /*
238  * Return a looped back vnode for the given vnode.
239  * If no lnode exists for this vnode create one and put it
240  * in a table hashed by vnode.  If the lnode for
241  * this vnode is already in the table return it (ref count is
242  * incremented by lfind).  The lnode will be flushed from the
243  * table when lo_inactive calls freelonode.  The creation of
244  * a new lnode can be forced via the LOF_FORCE flag even if
245  * the vnode exists in the table.  This is used in the creation
246  * of a terminating lnode when looping is detected.  A unique
247  * lnode is required for the correct evaluation of the current
248  * working directory.
249  * NOTE: vp is assumed to be a held vnode.
250  */
251 struct vnode *
252 makelonode(struct vnode *vp, struct loinfo *li, int flag)
253 {
254         lnode_t *lp, *tlp;
255         struct vfs *vfsp;
256         vnode_t *nvp;
```



```
258         lp = NULL;
259         TABLE_LOCK_ENTER(vp, li);
260         if (flag != LOF_FORCE)
261                 lp = lfind(vp, li);
262         if ((flag == LOF_FORCE) || (lp == NULL)) {
263                 /*
264                  * Optimistically assume that we won't need to sleep.
265                  */
266                 lp = kmem_cache_alloc(lnode_cache, KM_NOSLEEP);
267                 nvp = vn_alloc(KM_NOSLEEP);
268                 if (lp == NULL || nvp == NULL) {
269                         TABLE_LOCK_EXIT(vp, li);
270                         /* The lnode allocation may have succeeded, save it */
271                         tlp = lp;
272                         if (tlp == NULL) {
273                                 tlp = kmem_cache_alloc(lnode_cache, KM_SLEEP);
274                         }
275                         if (nvp == NULL) {
276                                 nvp = vn_alloc(KM_SLEEP);
277                         }
278                         lp = NULL;
279                         TABLE_LOCK_ENTER(vp, li);
280                         if (flag != LOF_FORCE)
281                                 lp = lfind(vp, li);
282                         if (lp != NULL) {
283                                 kmem_cache_free(lnode_cache, tlp);
284                                 vn_free(nvp);
285                                 VN_RELE(vp);
286                                 goto found_lnode;
287                         }
288                         lp = tlp;
289                 }
290                 atomic_inc_32(&li->li_refct);
292                 atomic_add_32(&li->li_refct, 1);
291                 vfsp = makelfsnode(vp->v_vfsp, li);
292                 lp->lo_vnode = nvp;
293                 VN_SET_VFS_TYPE_DEV(nvp, vfsp, vp->v_type, vp->v_rdev);
294                 nvp->v_flag |= (vp->v_flag & (VNOMOUNT|VNOMAP|VDIROPEN));
295                 vn_setops(nvp, lo_vnodeops);
296                 nvp->v_data = (caddr_t)lp;
297                 lp->lo_vp = vp;
298                 lp->lo_looping = 0;
299                 lsave(lp, li);
300                 vn_exists(vp);
301         } else {
302                 VN_RELE(vp);
303         }

305 found_lnode:
306         TABLE_LOCK_EXIT(vp, li);
307         return (ltov(lp));
308 }
```

*unchanged_portion_omitted_*

```
622 /*
623  * Our version of vfs_rele() that stops at 1 instead of 0, and calls
624  * freelfsnode() instead of kmem_free().
625  */
626 static void
627 lfs_rele(struct lfsnode *lfs, struct loinfo *li)
628 {
629         vfs_t *vfsp = &lfs->lfs_vfs;

631         ASSERT(MUTEX_HELD(&li->li_lfslock));
632         ASSERT(vfsp->vfs_count > 1);
633         if (atomic_dec_32_nv(&vfsp->vfs_count) == 1)
```

```
635         if (atomic_add_32_nv(&vfsp->vfs_count, -1) == 1)
634                 freelfsnode(lfs, li);
635 }

637 /*
638  * Remove a lnode from the table
639  */
640 void
641 freelonode(lnode_t *lp)
642 {
643         lnode_t *lt;
644         lnode_t *ltprev = NULL;
645         struct lfsnode *lfs, *nextlfs;
646         struct vfs *vfsp;
647         struct vnode *vp = ltov(lp);
648         struct vnode *realvp = realvp(vp);
649         struct loinfo *li = vtoli(vp->v_vfsp);

651 #ifdef LODEBUG
652         lo_dprint(4, "freelonode lp %p hash %d\n",
653             lp, ltablehash(lp->lo_vp, li));
654 #endif
655         TABLE_LOCK_ENTER(lp->lo_vp, li);

657         mutex_enter(&vp->v_lock);
658         if (vp->v_count > 1) {
659                 vp->v_count--;  /* release our hold from vn_rele */
660                 mutex_exit(&vp->v_lock);
661                 TABLE_LOCK_EXIT(lp->lo_vp, li);
662                 return;
663         }
664         mutex_exit(&vp->v_lock);

666         for (lt = TABLE_BUCKET(lp->lo_vp, li); lt != NULL;
667             ltprev = lt, lt = lt->lo_next) {
668                 if (lt == lp) {
669 #ifdef LODEBUG
670                         lo_dprint(4, "freeing %p, vfsp %p\n",
671                             vp, vp->v_vfsp);
672 #endif
673                         atomic_dec_32(&li->li_refct);
675                         atomic_add_32(&li->li_refct, -1);
674                         vfsp = vp->v_vfsp;
675                         vn_invalid(vp);
676                         if (vfsp != li->li_mountvfs) {
677                                 mutex_enter(&li->li_lfslock);
678                                 /*
679                                  * Check for unused lfs
680                                  */
681                                 lfs = li->li_lfs;
682                                 while (lfs != NULL) {
683                                         nextlfs = lfs->lfs_next;
684                                         if (vfsp == &lfs->lfs_vfs) {
685                                                 lfs_rele(lfs, li);
686                                                 break;
687                                         }
688                                         if (lfs->lfs_vfs.vfs_count == 1) {
689                                                 /*
690                                                  * Lfs is idle
691                                                  */
692                                                 freelfsnode(lfs, li);
693                                         }
694                                         lfs = nextlfs;
695                                 }
696                                 mutex_exit(&li->li_lfslock);
697                         }
```



```
698                         if (ltprev == NULL) {
699                                 TABLE_BUCKET(lt->lo_vp, li) = lt->lo_next;
700                         } else {
701                                 ltprev->lo_next = lt->lo_next;
702                         }
703                         TABLE_COUNT(lt->lo_vp, li)--;
704                         TABLE_LOCK_EXIT(lt->lo_vp, li);
705                         kmem_cache_free(lnode_cache, lt);
706                         vn_free(vp);
707                         VN_RELE(realvp);
708                         return;
709                 }
710         }
711         panic("freelonode");
712         /*NOTREACHED*/
713 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   55726 Mon Jul 28 07:44:15 2014
new/usr/src/uts/common/fs/mntfs/mntvnops.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 836 /* ARGSUSED */
 837 static int
 838 mntopen(vnode_t **vpp, int flag, cred_t *cr, caller_context_t *ct)
 839 {
 840         vnode_t *vp = *vpp;
 841         mntnode_t *nmnp;

 843         /*
 844          * Not allowed to open for writing, return error.
 845          */
 846         if (flag & FWRITE)
 847                 return (EPERM);
 848         /*
 849          * Create a new mnt/vnode for each open, this will give us a handle to
 850          * hang the snapshot on.
 851          */
 852         nmnp = mntgetnode(vp);

 854         *vpp = MTOV(nmnp);
 855         atomic_inc_32(&MTOD(nmnp)->mnt_nopen);
 855         atomic_add_32(&MTOD(nmnp)->mnt_nopen, 1);
 856         VN_RELE(vp);
 857         return (0);
 858 }

 860 /* ARGSUSED */
 861 static int
 862 mntclose(vnode_t *vp, int flag, int count, offset_t offset, cred_t *cr,
 863         caller_context_t *ct)
 864 {
 865         mntnode_t *mnp = VTOM(vp);

 867         /* Clean up any locks or shares held by the current process */
 868         cleanlocks(vp, ttoproc(curthread)->p_pid, 0);
 869         cleanshares(vp, ttoproc(curthread)->p_pid);

 871         if (count > 1)
 872                 return (0);
 873         if (vp->v_count == 1) {
 874                 rw_enter(&mnp->mnt_contents, RW_WRITER);
 875                 mntfs_freesnap(mnp, &mnp->mnt_read);
 876                 mntfs_freesnap(mnp, &mnp->mnt_ioctl);
 877                 rw_exit(&mnp->mnt_contents);
 878                 atomic_dec_32(&MTOD(mnp)->mnt_nopen);
 878                 atomic_add_32(&MTOD(mnp)->mnt_nopen, -1);
 879         }
 880         return (0);
 881 }
______unchanged_portion_omitted_
```

****
171885 Mon Jul 28 07:44:15 2014
new/usr/src/uts/common/fs/nfs/nfs3_vnops.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____unchanged_portion_omitted_

```
5215 /* ARGSUSED */
5216 static int
5217 nfs3_map(vnode_t *vp, offset_t off, struct as *as, caddr_t *addrp,
5218         size_t len, uchar_t prot, uchar_t maxprot, uint_t flags,
5219         cred_t *cr, caller_context_t *ct)
5220 {
5221         struct segvn_crargs vn_a;
5222         int error;
5223         rnode_t *rp;
5224         struct vattr va;

5226         if (nfs_zone() != VTOMI(vp)->mi_zone)
5227                 return (EIO);

5229         if (vp->v_flag & VNOMAP)
5230                 return (ENOSYS);

5232         if (off < 0 || off + len < 0)
5233                 return (ENXIO);

5235         if (vp->v_type != VREG)
5236                 return (ENODEV);

5238         /*
5239          * If there is cached data and if close-to-open consistency
5240          * checking is not turned off and if the file system is not
5241          * mounted readonly, then force an over the wire getattr.
5242          * Otherwise, just invoke nfs3getattr to get a copy of the
5243          * attributes.  The attribute cache will be used unless it
5244          * is timed out and if it is, then an over the wire getattr
5245          * will be issued.
5246          */
5247         va.va_mask = AT_ALL;
5248         if (vn_has_cached_data(vp) &&
5249             !(VTOMI(vp)->mi_flags & MI_NOCTO) && !vn_is_readonly(vp))
5250                 error = nfs3_getattr_otw(vp, &va, cr);
5251         else
5252                 error = nfs3getattr(vp, &va, cr);
5253         if (error)
5254                 return (error);

5256         /*
5257          * Check to see if the vnode is currently marked as not cachable.
5258          * This means portions of the file are locked (through VOP_FRLOCK).
5259          * In this case the map request must be refused.  We use
5260          * rp->r_lkserlock to avoid a race with concurrent lock requests.
5261          */
5262         rp = VTOR(vp);

5264         /*
5265          * Atomically increment r_inmap after acquiring r_rwlock. The
5266          * idea here is to acquire r_rwlock to block read/write and
5267          * not to protect r_inmap. r_inmap will inform nfs3_read/write()
5268          * that we are in nfs3_map(). Now, r_rwlock is acquired in order
5269          * and we can prevent the deadlock that would have occurred
5270          * when nfs3_addmap() would have acquired it out of order.
5271          *
5272          * Since we are not protecting r_inmap by any lock, we do not
5273          * hold any lock when we decrement it. We atomically decrement
```



```
5274          * r_inmap after we release r_lkserlock.
5275          */

5277         if (nfs_rw_enter_sig(&rp->r_rwlock, RW_WRITER, INTR(vp)))
5278                 return (EINTR);
5279         atomic_inc_uint(&rp->r_inmap);
5279         atomic_add_int(&rp->r_inmap, 1);
5280         nfs_rw_exit(&rp->r_rwlock);

5282         if (nfs_rw_enter_sig(&rp->r_lkserlock, RW_READER, INTR(vp))) {
5283                 atomic_dec_uint(&rp->r_inmap);
5283                 atomic_add_int(&rp->r_inmap, -1);
5284                 return (EINTR);
5285         }

5287         if (vp->v_flag & VNOCACHE) {
5288                 error = EAGAIN;
5289                 goto done;
5290         }

5292         /*
5293          * Don't allow concurrent locks and mapping if mandatory locking is
5294          * enabled.
5295          */
5296         if ((flk_has_remote_locks(vp) || lm_has_sleep(vp)) &&
5297             MANDLOCK(vp, va.va_mode)) {
5298                 error = EAGAIN;
5299                 goto done;
5300         }

5302         as_rangelock(as);
5303         error = choose_addr(as, addrp, len, off, ADDR_VACALIGN, flags);
5304         if (error != 0) {
5305                 as_rangeunlock(as);
5306                 goto done;
5307         }

5309         vn_a.vp = vp;
5310         vn_a.offset = off;
5311         vn_a.type = (flags & MAP_TYPE);
5312         vn_a.prot = (uchar_t)prot;
5313         vn_a.maxprot = (uchar_t)maxprot;
5314         vn_a.flags = (flags & ~MAP_TYPE);
5315         vn_a.cred = cr;
5316         vn_a.amp = NULL;
5317         vn_a.szc = 0;
5318         vn_a.lgrp_mem_policy_flags = 0;

5320         error = as_map(as, *addrp, len, segvn_create, &vn_a);
5321         as_rangeunlock(as);

5323 done:
5324         nfs_rw_exit(&rp->r_lkserlock);
5325         atomic_dec_uint(&rp->r_inmap);
5325         atomic_add_int(&rp->r_inmap, -1);
5326         return (error);
5327 }
```

_____unchanged_portion_omitted_

```
**********************************************************
  116599 Mon Jul 28 07:44:16 2014
new/usr/src/uts/common/fs/nfs/nfs4_client.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_
3148 void
3149 mi_hold(mntinfo4_t *mi)
3150 {
3151         atomic_inc_32(&mi->mi_count);
3151         atomic_add_32(&mi->mi_count, 1);
3152         ASSERT(mi->mi_count != 0);
3153 }

3155 void
3156 mi_rele(mntinfo4_t *mi)
3157 {
3158         ASSERT(mi->mi_count != 0);
3159         if (atomic_dec_32_nv(&mi->mi_count) == 0) {
3159         if (atomic_add_32_nv(&mi->mi_count, -1) == 0) {
3160                 nfs_free_mi4(mi);
3161         }
3162 }
______unchanged_portion_omitted_

4111 void
4112 fn_hold(nfs4_fname_t *fnp)
4113 {
4114         atomic_inc_32(&fnp->fn_refcnt);
4114         atomic_add_32(&fnp->fn_refcnt, 1);
4115         NFS4_DEBUG(nfs4_fname_debug, (CE_NOTE,
4116             "fn_hold %p:%s, new refcnt=%d",
4117             (void *)fnp, fnp->fn_name, fnp->fn_refcnt));
4118 }

4120 /*
4121  * Decrement the reference count of the given fname, and destroy it if its
4122  * reference count goes to zero.  Nulls out the given pointer.
4123  */

4125 void
4126 fn_rele(nfs4_fname_t **fnpp)
4127 {
4128         nfs4_fname_t *parent;
4129         uint32_t newref;
4130         nfs4_fname_t *fnp;

4132 recur:
4133         fnp = *fnpp;
4134         *fnpp = NULL;

4136         mutex_enter(&fnp->fn_lock);
4137         parent = fnp->fn_parent;
4138         if (parent != NULL)
4139                 mutex_enter(&parent->fn_lock);  /* prevent new references */
4140         newref = atomic_dec_32_nv(&fnp->fn_refcnt);
4140         newref = atomic_add_32_nv(&fnp->fn_refcnt, -1);
4141         if (newref > 0) {
4142                 NFS4_DEBUG(nfs4_fname_debug, (CE_NOTE,
4143                     "fn_rele %p:%s, new refcnt=%d",
4144                     (void *)fnp, fnp->fn_name, fnp->fn_refcnt));
4145                 if (parent != NULL)
4146                         mutex_exit(&parent->fn_lock);
4147                 mutex_exit(&fnp->fn_lock);
4148                 return;
4149         }
```



```
4151         NFS4_DEBUG(nfs4_fname_debug, (CE_NOTE,
4152             "fn_rele %p:%s, last reference, deleting...",
4153             (void *)fnp, fnp->fn_name));
4154         if (parent != NULL) {
4155                 avl_remove(&parent->fn_children, fnp);
4156                 mutex_exit(&parent->fn_lock);
4157         }
4158         kmem_free(fnp->fn_name, fnp->fn_len + 1);
4159         sfh4_rele(&fnp->fn_sfh);
4160         mutex_destroy(&fnp->fn_lock);
4161         avl_destroy(&fnp->fn_children);
4162         kmem_free(fnp, sizeof (nfs4_fname_t));
4163         /*
4164          * Recursivly fn_rele the parent.
4165          * Use goto instead of a recursive call to avoid stack overflow.
4166          */
4167         if (parent != NULL) {
4168                 fnpp = &parent;
4169                 goto recur;
4170         }
4171 }
______unchanged_portion_omitted_
```

```
**********************************************************
   62532 Mon Jul 28 07:44:16 2014
new/usr/src/uts/common/fs/nfs/nfs4_client_state.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

*unchanged_portion_omitted_*

```
 665 /*
 666  * Sequence number used when a new open owner is needed.
 667  * This is used so as to not confuse the server.  Since a open owner
 668  * is based off of cred, a cred could be re-used quickly, and the server
 669  * may not release all state for a cred.
 670  */
 671 static uint64_t open_owner_seq_num = 0;

 673 uint64_t
 674 nfs4_get_new_oo_name(void)
 675 {
 676         return (atomic_inc_64_nv(&open_owner_seq_num));
 676         return (atomic_add_64_nv(&open_owner_seq_num, 1));
 677 }
```

*unchanged_portion_omitted_*

```
 806 static uint64_t lock_owner_seq_num = 0;

 808 /*
 809  * Create a new lock owner and add it to the rnode's list.
 810  * Assumes the rnode's r_statev4_lock is held.
 811  * The created lock owner has a reference count of 2: one for the list and
 812  * one for the caller to use.  Returns the lock owner locked down.
 813  */
 814 nfs4_lock_owner_t *
 815 create_lock_owner(rnode4_t *rp, pid_t pid)
 816 {
 817         nfs4_lock_owner_t       *lop;

 819         NFS4_DEBUG(nfs4_client_state_debug, (CE_NOTE,
 820             "create_lock_owner: pid %x", pid));

 822         ASSERT(mutex_owned(&rp->r_statev4_lock));

 824         lop = kmem_alloc(sizeof (nfs4_lock_owner_t), KM_SLEEP);
 825         lop->lo_ref_count = 2;
 826         lop->lo_valid = 1;
 827         bzero(&lop->lock_stateid, sizeof (stateid4));
 828         lop->lo_pid = pid;
 829         lop->lock_seqid = 0;
 830         lop->lo_pending_rqsts = 0;
 831         lop->lo_just_created = NFS4_JUST_CREATED;
 832         lop->lo_flags = 0;
 833         lop->lo_seqid_holder = NULL;

 835         /*
 836          * A Solaris lock_owner is <seq_num><pid>
 837          */
 838         lop->lock_owner_name.ln_seq_num =
 839             atomic_inc_64_nv(&lock_owner_seq_num);
 839             atomic_add_64_nv(&lock_owner_seq_num, 1);
 840         lop->lock_owner_name.ln_pid = pid;

 842         cv_init(&lop->lo_cv_seqid_sync, NULL, CV_DEFAULT, NULL);
 843         mutex_init(&lop->lo_lock, NULL, MUTEX_DEFAULT, NULL);

 845         mutex_enter(&lop->lo_lock);

 847         /* now add the lock owner to rp */
```



```
 848         lop->lo_prev_rnode = &rp->r_lo_head;
 849         lop->lo_next_rnode = rp->r_lo_head.lo_next_rnode;
 850         rp->r_lo_head.lo_next_rnode->lo_prev_rnode = lop;
 851         rp->r_lo_head.lo_next_rnode = lop;

 853         return (lop);

 855 }
```

*unchanged_portion_omitted_*

```
 872 static void
 873 nfs4_set_new_lock_owner_args(lock_owner4 *owner, pid_t pid)
 874 {
 875         nfs4_lo_name_t *cast_namep;

 877         NFS4_DEBUG(nfs4_client_state_debug, (CE_NOTE,
 878             "nfs4_set_new_lock_owner_args"));

 880         owner->owner_len = sizeof (*cast_namep);
 881         owner->owner_val = kmem_alloc(owner->owner_len, KM_SLEEP);
 882         /*
 883          * A Solaris lock_owner is <seq_num><pid>
 884          */
 885         cast_namep = (nfs4_lo_name_t *)owner->owner_val;
 886         cast_namep->ln_seq_num = atomic_inc_64_nv(&lock_owner_seq_num);
 886         cast_namep->ln_seq_num = atomic_add_64_nv(&lock_owner_seq_num, 1);
 887         cast_namep->ln_pid = pid;
 888 }
```

*unchanged_portion_omitted_*

```
*********************************************************
   23458 Mon Jul 28 07:44:16 2014
new/usr/src/uts/common/fs/nfs/nfs4_db.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
*********************************************************
______unchanged_portion_omitted_

  60 void
  61 rfs4_dbe_hold(rfs4_dbe_t *entry)
  62 {
  63         atomic_inc_32(&entry->dbe_refcnt);
  63         atomic_add_32(&entry->dbe_refcnt, 1);
  64 }

  66 /*
  67  * rfs4_dbe_rele_nolock only decrements the reference count of the entry.
  68  */
  69 void
  70 rfs4_dbe_rele_nolock(rfs4_dbe_t *entry)
  71 {
  72         atomic_dec_32(&entry->dbe_refcnt);
  72         atomic_add_32(&entry->dbe_refcnt, -1);
  73 }
______unchanged_portion_omitted_

 127 void
 128 rfs4_dbe_rele(rfs4_dbe_t *entry)
 129 {
 130         mutex_enter(entry->dbe_lock);
 131         ASSERT(entry->dbe_refcnt > 1);
 132         atomic_dec_32(&entry->dbe_refcnt);
 132         atomic_add_32(&entry->dbe_refcnt, -1);
 133         entry->dbe_time_rele = gethrestime_sec();
 134         mutex_exit(entry->dbe_lock);
 135 }
______unchanged_portion_omitted_
```

```
**********************************************************
   49016 Mon Jul 28 07:44:16 2014
new/usr/src/uts/common/fs/nfs/nfs4_rnode.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 562 /*
 563  * Find or create the vnode for the given filehandle and filesystem.
 564  * *newnode is set to zero if the vnode already existed; non-zero if it had
 565  * to be created.
 566  *
 567  * Note: make_rnode4() may upgrade the hash bucket lock to exclusive.
 568  */

 570 static vnode_t *
 571 make_rnode4(nfs4_sharedfh_t *fh, r4hashq_t *rhtp, struct vfs *vfsp,
 572     struct vnodeops *vops,
 573     int (*putapage)(vnode_t *, page_t *, u_offset_t *, size_t *, int, cred_t *),
 574     int *newnode, cred_t *cr)
 575 {
 576         rnode4_t *rp;
 577         rnode4_t *trp;
 578         vnode_t *vp;
 579         mntinfo4_t *mi;

 581         ASSERT(RW_READ_HELD(&rhtp->r_lock));

 583         mi = VFTOMI4(vfsp);

 585 start:
 586         if ((rp = r4find(rhtp, fh, vfsp)) != NULL) {
 587                 vp = RTOV4(rp);
 588                 *newnode = 0;
 589                 return (vp);
 590         }
 591         rw_exit(&rhtp->r_lock);

 593         mutex_enter(&rp4freelist_lock);

 595         if (rp4freelist != NULL && rnode4_new >= nrnode) {
 596                 rp = rp4freelist;
 597                 rp4_rmfree(rp);
 598                 mutex_exit(&rp4freelist_lock);

 600                 vp = RTOV4(rp);

 602                 if (rp->r_flags & R4HASHED) {
 603                         rw_enter(&rp->r_hashq->r_lock, RW_WRITER);
 604                         mutex_enter(&vp->v_lock);
 605                         if (vp->v_count > 1) {
 606                                 vp->v_count--;
 607                                 mutex_exit(&vp->v_lock);
 608                                 rw_exit(&rp->r_hashq->r_lock);
 609                                 rw_enter(&rhtp->r_lock, RW_READER);
 610                                 goto start;
 611                         }
 612                         mutex_exit(&vp->v_lock);
 613                         rp4_rmhash_locked(rp);
 614                         rw_exit(&rp->r_hashq->r_lock);
 615                 }

 617                 r4inactive(rp, cr);

 619                 mutex_enter(&vp->v_lock);
 620                 if (vp->v_count > 1) {
```



```
 621                         vp->v_count--;
 622                         mutex_exit(&vp->v_lock);
 623                         rw_enter(&rhtp->r_lock, RW_READER);
 624                         goto start;
 625                 }
 626                 mutex_exit(&vp->v_lock);
 627                 vn_invalid(vp);

 629                 /*
 630                  * destroy old locks before bzero'ing and
 631                  * recreating the locks below.
 632                  */
 633                 uninit_rnode4(rp);

 635                 /*
 636                  * Make sure that if rnode is recycled then
 637                  * VFS count is decremented properly before
 638                  * reuse.
 639                  */
 640                 VFS_RELE(vp->v_vfsp);
 641                 vn_reinit(vp);
 642         } else {
 643                 vnode_t *new_vp;

 645                 mutex_exit(&rp4freelist_lock);

 647                 rp = kmem_cache_alloc(rnode4_cache, KM_SLEEP);
 648                 new_vp = vn_alloc(KM_SLEEP);

 650                 atomic_inc_ulong((ulong_t *)&rnode4_new);
 650                 atomic_add_long((ulong_t *)&rnode4_new, 1);
 651 #ifdef DEBUG
 652                 clstat4_debug.nrnode.value.ui64++;
 653 #endif
 654                 vp = new_vp;
 655         }

 657         bzero(rp, sizeof (*rp));
 658         rp->r_vnode = vp;
 659         nfs_rw_init(&rp->r_rwlock, NULL, RW_DEFAULT, NULL);
 660         nfs_rw_init(&rp->r_lkserlock, NULL, RW_DEFAULT, NULL);
 661         mutex_init(&rp->r_svlock, NULL, MUTEX_DEFAULT, NULL);
 662         mutex_init(&rp->r_statelock, NULL, MUTEX_DEFAULT, NULL);
 663         mutex_init(&rp->r_statev4_lock, NULL, MUTEX_DEFAULT, NULL);
 664         mutex_init(&rp->r_os_lock, NULL, MUTEX_DEFAULT, NULL);
 665         rp->created_v4 = 0;
 666         list_create(&rp->r_open_streams, sizeof (nfs4_open_stream_t),
 667             offsetof(nfs4_open_stream_t, os_node));
 668         rp->r_lo_head.lo_prev_rnode = &rp->r_lo_head;
 669         rp->r_lo_head.lo_next_rnode = &rp->r_lo_head;
 670         cv_init(&rp->r_cv, NULL, CV_DEFAULT, NULL);
 671         cv_init(&rp->r_commit.c_cv, NULL, CV_DEFAULT, NULL);
 672         rp->r_flags = R4READDIRWATTR;
 673         rp->r_fh = fh;
 674         rp->r_hashq = rhtp;
 675         sfh4_hold(rp->r_fh);
 676         rp->r_server = mi->mi_curr_serv;
 677         rp->r_deleg_type = OPEN_DELEGATE_NONE;
 678         rp->r_deleg_needs_recovery = OPEN_DELEGATE_NONE;
 679         nfs_rw_init(&rp->r_deleg_recall_lock, NULL, RW_DEFAULT, NULL);

 681         rddir4_cache_create(rp);
 682         rp->r_putapage = putapage;
 683         vn_setops(vp, vops);
 684         vp->v_data = (caddr_t)rp;
 685         vp->v_vfsp = vfsp;
```

```
 686         VFS_HOLD(vfsp);
 687         vp->v_type = VNON;
 688         vp->v_flag |= VMODSORT;
 689         if (isrootfh(fh, rp))
 690                 vp->v_flag = VROOT;
 691         vn_exists(vp);

 693         /*
 694          * There is a race condition if someone else
 695          * alloc's the rnode while no locks are held, so we
 696          * check again and recover if found.
 697          */
 698         rw_enter(&rhtp->r_lock, RW_WRITER);
 699         if ((trp = r4find(rhtp, fh, vfsp)) != NULL) {
 700                 vp = RTOV4(trp);
 701                 *newnode = 0;
 702                 rw_exit(&rhtp->r_lock);
 703                 rp4_addfree(rp, cr);
 704                 rw_enter(&rhtp->r_lock, RW_READER);
 705                 return (vp);
 706         }
 707         rp4_addhash(rp);
 708         *newnode = 1;
 709         return (vp);
 710 }
______unchanged_portion_omitted_

1207 /*
1208  * This routine destroys all the resources of an rnode
1209  * and finally the rnode itself.
1210  */
1211 static void
1212 destroy_rnode4(rnode4_t *rp)
1213 {
1214         vnode_t *vp;
1215         vfs_t *vfsp;

1217         ASSERT(rp->r_deleg_type == OPEN_DELEGATE_NONE);

1219         vp = RTOV4(rp);
1220         vfsp = vp->v_vfsp;

1222         uninit_rnode4(rp);
1223         atomic_dec_ulong((ulong_t *)&rnode4_new);
1223         atomic_add_long((ulong_t *)&rnode4_new, -1);
1224 #ifdef DEBUG
1225         clstat4_debug.nrnode.value.ui64--;
1226 #endif
1227         kmem_cache_free(rnode4_cache, rp);
1228         vn_invalid(vp);
1229         vn_free(vp);
1230         VFS_RELE(vfsp);
1231 }
______unchanged_portion_omitted_
```

```
**********************************************************
   77321 Mon Jul 28 07:44:17 2014
new/usr/src/uts/common/fs/nfs/nfs4_subr.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 820 /*
 821  * Common handle get program for NFS, NFS ACL, and NFS AUTH client.
 822  */
 823 int
 824 clget4(clinfo_t *ci, servinfo4_t *svp, cred_t *cr, CLIENT **newcl,
 825     struct chtab **chp, struct nfs4_clnt *nfscl)
 826 {
 827         struct chhead *ch, *newch;
 828         struct chhead **plistp;
 829         struct chtab *cp;
 830         int error;
 831         k_sigset_t smask;

 833         if (newcl == NULL || chp == NULL || ci == NULL)
 834                 return (EINVAL);

 836         *newcl = NULL;
 837         *chp = NULL;

 839         /*
 840          * Find an unused handle or create one
 841          */
 842         newch = NULL;
 843         nfscl->nfscl_stat.clgets.value.ui64++;
 844 top:
 845         /*
 846          * Find the correct entry in the cache to check for free
 847          * client handles.  The search is based on the RPC program
 848          * number, program version number, dev_t for the transport
 849          * device, and the protocol family.
 850          */
 851         mutex_enter(&nfscl->nfscl_chtable4_lock);
 852         plistp = &nfscl->nfscl_chtable4;
 853         for (ch = nfscl->nfscl_chtable4; ch != NULL; ch = ch->ch_next) {
 854                 if (ch->ch_prog == ci->cl_prog &&
 855                     ch->ch_vers == ci->cl_vers &&
 856                     ch->ch_dev == svp->sv_knconf->knc_rdev &&
 857                     (strcmp(ch->ch_protofmly,
 858                     svp->sv_knconf->knc_protofmly) == 0))
 859                         break;
 860                 plistp = &ch->ch_next;
 861         }

 863         /*
 864          * If we didn't find a cache entry for this quadruple, then
 865          * create one.  If we don't have one already preallocated,
 866          * then drop the cache lock, create one, and then start over.
 867          * If we did have a preallocated entry, then just add it to
 868          * the front of the list.
 869          */
 870         if (ch == NULL) {
 871                 if (newch == NULL) {
 872                         mutex_exit(&nfscl->nfscl_chtable4_lock);
 873                         newch = kmem_alloc(sizeof (*newch), KM_SLEEP);
 874                         newch->ch_timesused = 0;
 875                         newch->ch_prog = ci->cl_prog;
 876                         newch->ch_vers = ci->cl_vers;
 877                         newch->ch_dev = svp->sv_knconf->knc_rdev;
 878                         newch->ch_protofmly = kmem_alloc(
```



```
 879                             strlen(svp->sv_knconf->knc_protofmly) + 1,
 880                             KM_SLEEP);
 881                         (void) strcpy(newch->ch_protofmly,
 882                             svp->sv_knconf->knc_protofmly);
 883                         newch->ch_list = NULL;
 884                         goto top;
 885                 }
 886                 ch = newch;
 887                 newch = NULL;
 888                 ch->ch_next = nfscl->nfscl_chtable4;
 889                 nfscl->nfscl_chtable4 = ch;
 890         /*
 891          * We found a cache entry, but if it isn't on the front of the
 892          * list, then move it to the front of the list to try to take
 893          * advantage of locality of operations.
 894          */
 895         } else if (ch != nfscl->nfscl_chtable4) {
 896                 *plistp = ch->ch_next;
 897                 ch->ch_next = nfscl->nfscl_chtable4;
 898                 nfscl->nfscl_chtable4 = ch;
 899         }

 901         /*
 902          * If there was a free client handle cached, then remove it
 903          * from the list, init it, and use it.
 904          */
 905         if (ch->ch_list != NULL) {
 906                 cp = ch->ch_list;
 907                 ch->ch_list = cp->ch_list;
 908                 mutex_exit(&nfscl->nfscl_chtable4_lock);
 909                 if (newch != NULL) {
 910                         kmem_free(newch->ch_protofmly,
 911                             strlen(newch->ch_protofmly) + 1);
 912                         kmem_free(newch, sizeof (*newch));
 913                 }
 914                 (void) clnt_tli_kinit(cp->ch_client, svp->sv_knconf,
 915                     &svp->sv_addr, ci->cl_readsize, ci->cl_retrans, cr);

 917                 /*
 918                  * Get an auth handle.
 919                  */
 920                 error = authget(svp, cp->ch_client, cr);
 921                 if (error || cp->ch_client->cl_auth == NULL) {
 922                         CLNT_DESTROY(cp->ch_client);
 923                         kmem_cache_free(chtab4_cache, cp);
 924                         return ((error != 0) ? error : EINTR);
 925                 }
 926                 ch->ch_timesused++;
 927                 *newcl = cp->ch_client;
 928                 *chp = cp;
 929                 return (0);
 930         }

 932         /*
 933          * There weren't any free client handles which fit, so allocate
 934          * a new one and use that.
 935          */
 936 #ifdef DEBUG
 937         atomic_inc_64(&nfscl->nfscl_stat.clalloc.value.ui64);
 937         atomic_add_64(&nfscl->nfscl_stat.clalloc.value.ui64, 1);
 938 #endif
 939         mutex_exit(&nfscl->nfscl_chtable4_lock);

 941         nfscl->nfscl_stat.cltoomany.value.ui64++;
 942         if (newch != NULL) {
 943                 kmem_free(newch->ch_protofmly, strlen(newch->ch_protofmly) + 1);
```

```
 944                 kmem_free(newch, sizeof (*newch));
 945         }

 947         cp = kmem_cache_alloc(chtab4_cache, KM_SLEEP);
 948         cp->ch_head = ch;

 950         sigintr(&smask, (int)ci->cl_flags & MI4_INT);
 951         error = clnt_tli_kcreate(svp->sv_knconf, &svp->sv_addr, ci->cl_prog,
 952             ci->cl_vers, ci->cl_readsize, ci->cl_retrans, cr, &cp->ch_client);
 953         sigunintr(&smask);

 955         if (error != 0) {
 956                 kmem_cache_free(chtab4_cache, cp);
 957 #ifdef DEBUG
 958                 atomic_dec_64(&nfscl->nfscl_stat.clalloc.value.ui64);
 958                 atomic_add_64(&nfscl->nfscl_stat.clalloc.value.ui64, -1);
 959 #endif
 960                 /*
 961                  * Warning is unnecessary if error is EINTR.
 962                  */
 963                 if (error != EINTR) {
 964                         nfs_cmn_err(error, CE_WARN,
 965                             "clget: couldn't create handle: %m\n");
 966                 }
 967                 return (error);
 968         }
 969         (void) CLNT_CONTROL(cp->ch_client, CLSET_PROGRESS, NULL);
 970         auth_destroy(cp->ch_client->cl_auth);

 972         /*
 973          * Get an auth handle.
 974          */
 975         error = authget(svp, cp->ch_client, cr);
 976         if (error || cp->ch_client->cl_auth == NULL) {
 977                 CLNT_DESTROY(cp->ch_client);
 978                 kmem_cache_free(chtab4_cache, cp);
 979 #ifdef DEBUG
 980                 atomic_dec_64(&nfscl->nfscl_stat.clalloc.value.ui64);
 980                 atomic_add_64(&nfscl->nfscl_stat.clalloc.value.ui64, -1);
 981 #endif
 982                 return ((error != 0) ? error : EINTR);
 983         }
 984         ch->ch_timesused++;
 985         *newcl = cp->ch_client;
 986         ASSERT(cp->ch_client->cl_nosignal == FALSE);
 987         *chp = cp;
 988         return (0);
 989 }
```

_____unchanged_portion_omitted_

```
2618 /*
2619  * Allocate a cache element and return it.  Can return NULL if memory is
2620  * low.
2621  */
2622 static rddir4_cache *
2623 rddir4_cache_alloc(int flags)
2624 {
2625         rddir4_cache_impl       *rdip = NULL;
2626         rddir4_cache            *rc = NULL;

2628         rdip = kmem_alloc(sizeof (rddir4_cache_impl), flags);

2630         if (rdip != NULL) {
2631                 rc = &rdip->rc;
2632                 rc->data = (void *)rdip;
2633                 rc->nfs4_cookie = 0;
```



```
2634                 rc->nfs4_ncookie = 0;
2635                 rc->entries = NULL;
2636                 rc->eof = 0;
2637                 rc->entlen = 0;
2638                 rc->buflen = 0;
2639                 rc->actlen = 0;
2640                 /*
2641                  * A readdir is required so set the flag.
2642                  */
2643                 rc->flags = RDDIRREQ;
2644                 cv_init(&rc->cv, NULL, CV_DEFAULT, NULL);
2645                 rc->error = 0;
2646                 mutex_init(&rdip->lock, NULL, MUTEX_DEFAULT, NULL);
2647                 rdip->count = 1;
2648 #ifdef DEBUG
2649                 atomic_inc_64(&clstat4_debug.dirent.value.ui64);
2649                 atomic_add_64(&clstat4_debug.dirent.value.ui64, 1);
2650 #endif
2651         }
2652         return (rc);
2653 }
```

_____unchanged_portion_omitted_

```
2697 /*
2698  * Free a cache element.
2699  */
2700 static void
2701 rddir4_cache_free(rddir4_cache_impl *rdip)
2702 {
2703         rddir4_cache *rc = &rdip->rc;

2705 #ifdef DEBUG
2706         atomic_dec_64(&clstat4_debug.dirent.value.ui64);
2706         atomic_add_64(&clstat4_debug.dirent.value.ui64, -1);
2707 #endif
2708         if (rc->entries != NULL)
2709                 kmem_free(rc->entries, rc->buflen);
2710         cv_destroy(&rc->cv);
2711         mutex_destroy(&rdip->lock);
2712         kmem_free(rdip, sizeof (*rdip));
2713 }
```

_____unchanged_portion_omitted_

****
430064 Mon Jul 28 07:44:17 2014
new/usr/src/uts/common/fs/nfs/nfs4_vnops.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____unchanged_portion_omitted_

```
10432 #ifdef DEBUG
10433 int nfs4_force_open_before_mmap = 0;
10434 #endif

10436 /* ARGSUSED */
10437 static int
10438 nfs4_map(vnode_t *vp, offset_t off, struct as *as, caddr_t *addrp,
10439     size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
10440     caller_context_t *ct)
10441 {
10442         struct segvn_crargs vn_a;
10443         int error = 0;
10444         rnode4_t *rp = VTOR4(vp);
10445         mntinfo4_t *mi = VTOMI4(vp);

10447         if (nfs_zone() != VTOMI4(vp)->mi_zone)
10448                 return (EIO);

10450         if (vp->v_flag & VNOMAP)
10451                 return (ENOSYS);

10453         if (off < 0 || (off + len) < 0)
10454                 return (ENXIO);

10456         if (vp->v_type != VREG)
10457                 return (ENODEV);

10459         /*
10460          * If the file is delegated to the client don't do anything.
10461          * If the file is not delegated, then validate the data cache.
10462          */
10463         mutex_enter(&rp->r_statev4_lock);
10464         if (rp->r_deleg_type == OPEN_DELEGATE_NONE) {
10465                 mutex_exit(&rp->r_statev4_lock);
10466                 error = nfs4_validate_caches(vp, cr);
10467                 if (error)
10468                         return (error);
10469         } else {
10470                 mutex_exit(&rp->r_statev4_lock);
10471         }

10473         /*
10474          * Check to see if the vnode is currently marked as not cachable.
10475          * This means portions of the file are locked (through VOP_FRLOCK).
10476          * In this case the map request must be refused.  We use
10477          * rp->r_lkserlock to avoid a race with concurrent lock requests.
10478          *
10479          * Atomically increment r_inmap after acquiring r_rwlock. The
10480          * idea here is to acquire r_rwlock to block read/write and
10481          * not to protect r_inmap. r_inmap will inform nfs4_read/write()
10482          * that we are in nfs4_map(). Now, r_rwlock is acquired in order
10483          * and we can prevent the deadlock that would have occurred
10484          * when nfs4_addmap() would have acquired it out of order.
10485          *
10486          * Since we are not protecting r_inmap by any lock, we do not
10487          * hold any lock when we decrement it. We atomically decrement
10488          * r_inmap after we release r_lkserlock.
10489          */
```



```
10491         if (nfs_rw_enter_sig(&rp->r_rwlock, RW_WRITER, INTR4(vp)))
10492                 return (EINTR);
10493         atomic_inc_uint(&rp->r_inmap);
10493         atomic_add_int(&rp->r_inmap, 1);
10494         nfs_rw_exit(&rp->r_rwlock);

10496         if (nfs_rw_enter_sig(&rp->r_lkserlock, RW_READER, INTR4(vp))) {
10497                 atomic_dec_uint(&rp->r_inmap);
10497                 atomic_add_int(&rp->r_inmap, -1);
10498                 return (EINTR);
10499         }


10502         if (vp->v_flag & VNOCACHE) {
10503                 error = EAGAIN;
10504                 goto done;
10505         }

10507         /*
10508          * Don't allow concurrent locks and mapping if mandatory locking is
10509          * enabled.
10510          */
10511         if (flk_has_remote_locks(vp)) {
10512                 struct vattr va;
10513                 va.va_mask = AT_MODE;
10514                 error = nfs4getattr(vp, &va, cr);
10515                 if (error != 0)
10516                         goto done;
10517                 if (MANDLOCK(vp, va.va_mode)) {
10518                         error = EAGAIN;
10519                         goto done;
10520                 }
10521         }

10523         /*
10524          * It is possible that the rnode has a lost lock request that we
10525          * are still trying to recover, and that the request conflicts with
10526          * this map request.
10527          *
10528          * An alternative approach would be for nfs4_safemap() to consider
10529          * queued lock requests when deciding whether to set or clear
10530          * VNOCACHE.  This would require the frlock code path to call
10531          * nfs4_safemap() after enqueing a lost request.
10532          */
10533         if (nfs4_map_lost_lock_conflict(vp)) {
10534                 error = EAGAIN;
10535                 goto done;
10536         }

10538         as_rangelock(as);
10539         error = choose_addr(as, addrp, len, off, ADDR_VACALIGN, flags);
10540         if (error != 0) {
10541                 as_rangeunlock(as);
10542                 goto done;
10543         }

10545         if (vp->v_type == VREG) {
10546                 /*
10547                  * We need to retrieve the open stream
10548                  */
10549                 nfs4_open_stream_t      *osp = NULL;
10550                 nfs4_open_owner_t       *oop = NULL;

10552                 oop = find_open_owner(cr, NFS4_PERM_CREATED, mi);
10553                 if (oop != NULL) {
10554                         /* returns with 'os_sync_lock' held */
```

```
10555                           osp = find_open_stream(oop, rp);
10556                           open_owner_rele(oop);
10557                   }
10558                   if (osp == NULL) {
10559 #ifdef DEBUG
10560                           if (nfs4_force_open_before_mmap) {
10561                                   error = EIO;
10562                                   goto done;
10563                           }
10564 #endif
10565                           /* returns with 'os_sync_lock' held */
10566                           error = open_and_get_osp(vp, cr, &osp);
10567                           if (osp == NULL) {
10568                                   NFS4_DEBUG(nfs4_mmap_debug, (CE_NOTE,
10569                                       "nfs4_map: we tried to OPEN the file "
10570                                       "but again no osp, so fail with EIO"));
10571                                   goto done;
10572                           }
10573                   }

10575                   if (osp->os_failed_reopen) {
10576                           mutex_exit(&osp->os_sync_lock);
10577                           open_stream_rele(osp, rp);
10578                           NFS4_DEBUG(nfs4_open_stream_debug, (CE_NOTE,
10579                               "nfs4_map: os_failed_reopen set on "
10580                               "osp %p, cr %p, rp %s", (void *)osp,
10581                               (void *)cr, rnode4info(rp)));
10582                           error = EIO;
10583                           goto done;
10584                   }
10585                   mutex_exit(&osp->os_sync_lock);
10586                   open_stream_rele(osp, rp);
10587           }

10589           vn_a.vp = vp;
10590           vn_a.offset = off;
10591           vn_a.type = (flags & MAP_TYPE);
10592           vn_a.prot = (uchar_t)prot;
10593           vn_a.maxprot = (uchar_t)maxprot;
10594           vn_a.flags = (flags & ~MAP_TYPE);
10595           vn_a.cred = cr;
10596           vn_a.amp = NULL;
10597           vn_a.szc = 0;
10598           vn_a.lgrp_mem_policy_flags = 0;

10600           error = as_map(as, *addrp, len, segvn_create, &vn_a);
10601           as_rangeunlock(as);

10603 done:
10604           nfs_rw_exit(&rp->r_lkserlock);
10605           atomic_dec_uint(&rp->r_inmap);
10605           atomic_add_int(&rp->r_inmap, -1);
10606           return (error);
10607 }
```

*unchanged_portion_omitted_*

```
**********************************************************
  126094 Mon Jul 28 07:44:17 2014
new/usr/src/uts/common/fs/nfs/nfs_subr.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_
 212 #endif  /* DEBUG */

 214 /*
 215  * We keep a global list of per-zone client data, so we can clean up all zones
 216  * if we get low on memory.
 217  */
 218 static list_t nfs_clnt_list;
 219 static kmutex_t nfs_clnt_list_lock;
 220 static zone_key_t nfsclnt_zone_key;

 222 static struct kmem_cache *chtab_cache;

 224 /*
 225  * Some servers do not properly update the attributes of the
 226  * directory when changes are made.  To allow interoperability
 227  * with these broken servers, the nfs_disable_rddir_cache
 228  * parameter must be set in /etc/system
 229  */
 230 int nfs_disable_rddir_cache = 0;

 232 int             clget(clinfo_t *, servinfo_t *, cred_t *, CLIENT **,
 233                     struct chtab **);
 234 void            clfree(CLIENT *, struct chtab *);
 235 static int      acl_clget(mntinfo_t *, servinfo_t *, cred_t *, CLIENT **,
 236                     struct chtab **, struct nfs_clnt *);
 237 static int      nfs_clget(mntinfo_t *, servinfo_t *, cred_t *, CLIENT **,
 238                     struct chtab **, struct nfs_clnt *);
 239 static void     clreclaim(void *);
 240 static int      nfs_feedback(int, int, mntinfo_t *);
 241 static int      rfscall(mntinfo_t *, rpcproc_t, xdrproc_t, caddr_t, xdrproc_t,
 242                     caddr_t, cred_t *, int *, enum clnt_stat *, int,
 243                     failinfo_t *);
 244 static int      aclcall(mntinfo_t *, rpcproc_t, xdrproc_t, caddr_t, xdrproc_t,
 245                     caddr_t, cred_t *, int *, int, failinfo_t *);
 246 static void     rinactive(rnode_t *, cred_t *);
 247 static int      rtablehash(nfs_fhandle *);
 248 static vnode_t  *make_rnode(nfs_fhandle *, rhashq_t *, struct vfs *,
 249                     struct vnodeops *,
 250                     int (*)(vnode_t *, page_t *, u_offset_t *, size_t *, int,
 251                         cred_t *),
 252                     int (*)(const void *, const void *), int *, cred_t *,
 253                     char *, char *);
 254 static void     rp_rmfree(rnode_t *);
 255 static void     rp_addhash(rnode_t *);
 256 static void     rp_rmhash_locked(rnode_t *);
 257 static rnode_t  *rfind(rhashq_t *, nfs_fhandle *, struct vfs *);
 258 static void     destroy_rnode(rnode_t *);
 259 static void     rddir_cache_free(rddir_cache *);
 260 static int      nfs_free_data_reclaim(rnode_t *);
 261 static int      nfs_active_data_reclaim(rnode_t *);
 262 static int      nfs_free_reclaim(void);
 263 static int      nfs_active_reclaim(void);
 264 static int      nfs_rnode_reclaim(void);
 265 static void     nfs_reclaim(void *);
 266 static int      failover_safe(failinfo_t *);
 267 static void     failover_newserver(mntinfo_t *mi);
 268 static void     failover_thread(mntinfo_t *mi);
 269 static int      failover_wait(mntinfo_t *);
 270 static int      failover_remap(failinfo_t *);
 271 static int      failover_lookup(char *, vnode_t *,
```



```
 272                     int (*)(vnode_t *, char *, vnode_t **,
 273                         struct pathname *, int, vnode_t *, cred_t *, int),
 274                     int (*)(vnode_t *, vnode_t **, bool_t, cred_t *, int),
 275                     vnode_t **);
 276 static void     nfs_free_r_path(rnode_t *);
 277 static void     nfs_set_vroot(vnode_t *);
 278 static char     *nfs_getsrvnames(mntinfo_t *, size_t *);

 280 /*
 281  * from rpcsec module (common/rpcsec)
 282  */
 283 extern int sec_clnt_geth(CLIENT *, struct sec_data *, cred_t *, AUTH **);
 284 extern void sec_clnt_freeh(AUTH *);
 285 extern void sec_clnt_freeinfo(struct sec_data *);

 287 /*
 288  * used in mount policy
 289  */
 290 extern ts_label_t *getflabel_cipso(vfs_t *);

 292 /*
 293  * EIO or EINTR are not recoverable errors.
 294  */
 295 #define IS_RECOVERABLE_ERROR(error)     !((error == EINTR) || (error == EIO))

 297 #ifdef DEBUG
 298 #define SRV_QFULL_MSG   "send queue to NFS%d server %s is full; still trying\n"
 299 #define SRV_NOTRESP_MSG "NFS%d server %s not responding still trying\n"
 300 #else
 301 #define SRV_QFULL_MSG   "send queue to NFS server %s is full still trying\n"
 302 #define SRV_NOTRESP_MSG "NFS server %s not responding still trying\n"
 303 #endif
 304 /*
 305  * Common handle get program for NFS, NFS ACL, and NFS AUTH client.
 306  */
 307 static int
 308 clget_impl(clinfo_t *ci, servinfo_t *svp, cred_t *cr, CLIENT **newcl,
 309     struct chtab **chp, struct nfs_clnt *nfscl)
 310 {
 311         struct chhead *ch, *newch;
 312         struct chhead **plistp;
 313         struct chtab *cp;
 314         int error;
 315         k_sigset_t smask;

 317         if (newcl == NULL || chp == NULL || ci == NULL)
 318                 return (EINVAL);

 320         *newcl = NULL;
 321         *chp = NULL;

 323         /*
 324          * Find an unused handle or create one
 325          */
 326         newch = NULL;
 327         nfscl->nfscl_stat.clgets.value.ui64++;
 328 top:
 329         /*
 330          * Find the correct entry in the cache to check for free
 331          * client handles.  The search is based on the RPC program
 332          * number, program version number, dev_t for the transport
 333          * device, and the protocol family.
 334          */
 335         mutex_enter(&nfscl->nfscl_chtable_lock);
 336         plistp = &nfscl->nfscl_chtable;
 337         for (ch = nfscl->nfscl_chtable; ch != NULL; ch = ch->ch_next) {
```

```
338                 if (ch->ch_prog == ci->cl_prog &&
339                     ch->ch_vers == ci->cl_vers &&
340                     ch->ch_dev == svp->sv_knconf->knc_rdev &&
341                     (strcmp(ch->ch_protofmly,
342                     svp->sv_knconf->knc_protofmly) == 0))
343                         break;
344                 plistp = &ch->ch_next;
345         }

347         /*
348          * If we didn't find a cache entry for this quadruple, then
349          * create one.  If we don't have one already preallocated,
350          * then drop the cache lock, create one, and then start over.
351          * If we did have a preallocated entry, then just add it to
352          * the front of the list.
353          */
354         if (ch == NULL) {
355                 if (newch == NULL) {
356                         mutex_exit(&nfscl->nfscl_chtable_lock);
357                         newch = kmem_alloc(sizeof (*newch), KM_SLEEP);
358                         newch->ch_timesused = 0;
359                         newch->ch_prog = ci->cl_prog;
360                         newch->ch_vers = ci->cl_vers;
361                         newch->ch_dev = svp->sv_knconf->knc_rdev;
362                         newch->ch_protofmly = kmem_alloc(
363                             strlen(svp->sv_knconf->knc_protofmly) + 1,
364                             KM_SLEEP);
365                         (void) strcpy(newch->ch_protofmly,
366                             svp->sv_knconf->knc_protofmly);
367                         newch->ch_list = NULL;
368                         goto top;
369                 }
370                 ch = newch;
371                 newch = NULL;
372                 ch->ch_next = nfscl->nfscl_chtable;
373                 nfscl->nfscl_chtable = ch;
374         /*
375          * We found a cache entry, but if it isn't on the front of the
376          * list, then move it to the front of the list to try to take
377          * advantage of locality of operations.
378          */
379         } else if (ch != nfscl->nfscl_chtable) {
380                 *plistp = ch->ch_next;
381                 ch->ch_next = nfscl->nfscl_chtable;
382                 nfscl->nfscl_chtable = ch;
383         }

385         /*
386          * If there was a free client handle cached, then remove it
387          * from the list, init it, and use it.
388          */
389         if (ch->ch_list != NULL) {
390                 cp = ch->ch_list;
391                 ch->ch_list = cp->ch_list;
392                 mutex_exit(&nfscl->nfscl_chtable_lock);
393                 if (newch != NULL) {
394                         kmem_free(newch->ch_protofmly,
395                             strlen(newch->ch_protofmly) + 1);
396                         kmem_free(newch, sizeof (*newch));
397                 }
398                 (void) clnt_tli_kinit(cp->ch_client, svp->sv_knconf,
399                     &svp->sv_addr, ci->cl_readsize, ci->cl_retrans, cr);
400                 error = sec_clnt_geth(cp->ch_client, svp->sv_secdata, cr,
401                     &cp->ch_client->cl_auth);
402                 if (error || cp->ch_client->cl_auth == NULL) {
403                         CLNT_DESTROY(cp->ch_client);
```



```
404                         kmem_cache_free(chtab_cache, cp);
405                         return ((error != 0) ? error : EINTR);
406                 }
407                 ch->ch_timesused++;
408                 *newcl = cp->ch_client;
409                 *chp = cp;
410                 return (0);
411         }

413         /*
414          * There weren't any free client handles which fit, so allocate
415          * a new one and use that.
416          */
417 #ifdef DEBUG
418         atomic_inc_64(&nfscl->nfscl_stat.clalloc.value.ui64);
418         atomic_add_64(&nfscl->nfscl_stat.clalloc.value.ui64, 1);
419 #endif
420         mutex_exit(&nfscl->nfscl_chtable_lock);

422         nfscl->nfscl_stat.cltoomany.value.ui64++;
423         if (newch != NULL) {
424                 kmem_free(newch->ch_protofmly, strlen(newch->ch_protofmly) + 1);
425                 kmem_free(newch, sizeof (*newch));
426         }

428         cp = kmem_cache_alloc(chtab_cache, KM_SLEEP);
429         cp->ch_head = ch;

431         sigintr(&smask, (int)ci->cl_flags & MI_INT);
432         error = clnt_tli_kcreate(svp->sv_knconf, &svp->sv_addr, ci->cl_prog,
433             ci->cl_vers, ci->cl_readsize, ci->cl_retrans, cr, &cp->ch_client);
434         sigunintr(&smask);

436         if (error != 0) {
437                 kmem_cache_free(chtab_cache, cp);
438 #ifdef DEBUG
439                 atomic_dec_64(&nfscl->nfscl_stat.clalloc.value.ui64);
439                 atomic_add_64(&nfscl->nfscl_stat.clalloc.value.ui64, -1);
440 #endif
441                 /*
442                  * Warning is unnecessary if error is EINTR.
443                  */
444                 if (error != EINTR) {
445                         nfs_cmn_err(error, CE_WARN,
446                             "clget: couldn't create handle: %m\n");
447                 }
448                 return (error);
449         }
450         (void) CLNT_CONTROL(cp->ch_client, CLSET_PROGRESS, NULL);
451         auth_destroy(cp->ch_client->cl_auth);
452         error = sec_clnt_geth(cp->ch_client, svp->sv_secdata, cr,
453             &cp->ch_client->cl_auth);
454         if (error || cp->ch_client->cl_auth == NULL) {
455                 CLNT_DESTROY(cp->ch_client);
456                 kmem_cache_free(chtab_cache, cp);
457 #ifdef DEBUG
458                 atomic_dec_64(&nfscl->nfscl_stat.clalloc.value.ui64);
458                 atomic_add_64(&nfscl->nfscl_stat.clalloc.value.ui64, -1);
459 #endif
460                 return ((error != 0) ? error : EINTR);
461         }
462         ch->ch_timesused++;
463         *newcl = cp->ch_client;
464         ASSERT(cp->ch_client->cl_nosignal == FALSE);
465         *chp = cp;
466         return (0);
```

```
 467 }
_____unchanged_portion_omitted_

2456 static vnode_t *
2457 make_rnode(nfs_fhandle *fh, rhashq_t *rhtp, struct vfs *vfsp,
2458     struct vnodeops *vops,
2459     int (*putapage)(vnode_t *, page_t *, u_offset_t *, size_t *, int, cred_t *),
2460     int (*compar)(const void *, const void *),
2461     int *newnode, cred_t *cr, char *dnm, char *nm)
2462 {
2463         rnode_t *rp;
2464         rnode_t *trp;
2465         vnode_t *vp;
2466         mntinfo_t *mi;

2468         ASSERT(RW_READ_HELD(&rhtp->r_lock));

2470         mi = VFTOMI(vfsp);
2471 start:
2472         if ((rp = rfind(rhtp, fh, vfsp)) != NULL) {
2473                 vp = RTOV(rp);
2474                 nfs_set_vroot(vp);
2475                 *newnode = 0;
2476                 return (vp);
2477         }
2478         rw_exit(&rhtp->r_lock);

2480         mutex_enter(&rpfreelist_lock);
2481         if (rpfreelist != NULL && rnew >= nrnode) {
2482                 rp = rpfreelist;
2483                 rp_rmfree(rp);
2484                 mutex_exit(&rpfreelist_lock);

2486                 vp = RTOV(rp);

2488                 if (rp->r_flags & RHASHED) {
2489                         rw_enter(&rp->r_hashq->r_lock, RW_WRITER);
2490                         mutex_enter(&vp->v_lock);
2491                         if (vp->v_count > 1) {
2492                                 vp->v_count--;
2493                                 mutex_exit(&vp->v_lock);
2494                                 rw_exit(&rp->r_hashq->r_lock);
2495                                 rw_enter(&rhtp->r_lock, RW_READER);
2496                                 goto start;
2497                         }
2498                         mutex_exit(&vp->v_lock);
2499                         rp_rmhash_locked(rp);
2500                         rw_exit(&rp->r_hashq->r_lock);
2501                 }

2503                 rinactive(rp, cr);

2505                 mutex_enter(&vp->v_lock);
2506                 if (vp->v_count > 1) {
2507                         vp->v_count--;
2508                         mutex_exit(&vp->v_lock);
2509                         rw_enter(&rhtp->r_lock, RW_READER);
2510                         goto start;
2511                 }
2512                 mutex_exit(&vp->v_lock);
2513                 vn_invalid(vp);
2514                 /*
2515                  * destroy old locks before bzero'ing and
2516                  * recreating the locks below.
2517                  */
2518                 nfs_rw_destroy(&rp->r_rwlock);
```



```
2519                 nfs_rw_destroy(&rp->r_lkserlock);
2520                 mutex_destroy(&rp->r_statelock);
2521                 cv_destroy(&rp->r_cv);
2522                 cv_destroy(&rp->r_commit.c_cv);
2523                 nfs_free_r_path(rp);
2524                 avl_destroy(&rp->r_dir);
2525                 /*
2526                  * Make sure that if rnode is recycled then
2527                  * VFS count is decremented properly before
2528                  * reuse.
2529                  */
2530                 VFS_RELE(vp->v_vfsp);
2531                 vn_reinit(vp);
2532         } else {
2533                 vnode_t *new_vp;

2535                 mutex_exit(&rpfreelist_lock);

2537                 rp = kmem_cache_alloc(rnode_cache, KM_SLEEP);
2538                 new_vp = vn_alloc(KM_SLEEP);

2540                 atomic_inc_ulong((ulong_t *)&rnew);
2540                 atomic_add_long((ulong_t *)&rnew, 1);
2541 #ifdef DEBUG
2542                 clstat_debug.nrnode.value.ui64++;
2543 #endif
2544                 vp = new_vp;
2545         }

2547         bzero(rp, sizeof (*rp));
2548         rp->r_vnode = vp;
2549         nfs_rw_init(&rp->r_rwlock, NULL, RW_DEFAULT, NULL);
2550         nfs_rw_init(&rp->r_lkserlock, NULL, RW_DEFAULT, NULL);
2551         mutex_init(&rp->r_statelock, NULL, MUTEX_DEFAULT, NULL);
2552         cv_init(&rp->r_cv, NULL, CV_DEFAULT, NULL);
2553         cv_init(&rp->r_commit.c_cv, NULL, CV_DEFAULT, NULL);
2554         rp->r_fh.fh_len = fh->fh_len;
2555         bcopy(fh->fh_buf, rp->r_fh.fh_buf, fh->fh_len);
2556         rp->r_server = mi->mi_curr_serv;
2557         if (FAILOVER_MOUNT(mi)) {
2558                 /*
2559                  * If replicated servers, stash pathnames
2560                  */
2561                 if (dnm != NULL && nm != NULL) {
2562                         char *s, *p;
2563                         uint_t len;

2565                         len = (uint_t)(strlen(dnm) + strlen(nm) + 2);
2566                         rp->r_path = kmem_alloc(len, KM_SLEEP);
2567 #ifdef DEBUG
2568                         clstat_debug.rpath.value.ui64 += len;
2569 #endif
2570                         s = rp->r_path;
2571                         for (p = dnm; *p; p++)
2572                                 *s++ = *p;
2573                         *s++ = '/';
2574                         for (p = nm; *p; p++)
2575                                 *s++ = *p;
2576                         *s = '\0';
2577                 } else {
2578                         /* special case for root */
2579                         rp->r_path = kmem_alloc(2, KM_SLEEP);
2580 #ifdef DEBUG
2581                         clstat_debug.rpath.value.ui64 += 2;
2582 #endif
2583                         *rp->r_path = '.';
```

```
2584                        *(rp->r_path + 1) = '\0';
2585                }
2586        }
2587        VFS_HOLD(vfsp);
2588        rp->r_putapage = putapage;
2589        rp->r_hashq = rhtp;
2590        rp->r_flags = RREADDIRPLUS;
2591        avl_create(&rp->r_dir, compar, sizeof (rddir_cache),
2592            offsetof(rddir_cache, tree));
2593        vn_setops(vp, vops);
2594        vp->v_data = (caddr_t)rp;
2595        vp->v_vfsp = vfsp;
2596        vp->v_type = VNON;
2597        vp->v_flag |= VMODSORT;
2598        nfs_set_vroot(vp);

2600        /*
2601         * There is a race condition if someone else
2602         * alloc's the rnode while no locks are held, so we
2603         * check again and recover if found.
2604         */
2605        rw_enter(&rhtp->r_lock, RW_WRITER);
2606        if ((trp = rfind(rhtp, fh, vfsp)) != NULL) {
2607                vp = RTOV(trp);
2608                nfs_set_vroot(vp);
2609                *newnode = 0;
2610                rw_exit(&rhtp->r_lock);
2611                rp_addfree(rp, cr);
2612                rw_enter(&rhtp->r_lock, RW_READER);
2613                return (vp);
2614        }
2615        rp_addhash(rp);
2616        *newnode = 1;
2617        return (vp);
2618 }
```

unchanged_portion_omitted_

```
3003 /*
3004  * This routine destroys all the resources associated with the rnode
3005  * and then the rnode itself.
3006  */
3007 static void
3008 destroy_rnode(rnode_t *rp)
3009 {
3010        vnode_t *vp;
3011        vfs_t *vfsp;

3013        vp = RTOV(rp);
3014        vfsp = vp->v_vfsp;

3016        ASSERT(vp->v_count == 1);
3017        ASSERT(rp->r_count == 0);
3018        ASSERT(rp->r_lmpl == NULL);
3019        ASSERT(rp->r_mapcnt == 0);
3020        ASSERT(!(rp->r_flags & RHASHED));
3021        ASSERT(rp->r_freef == NULL && rp->r_freeb == NULL);
3022        atomic_dec_ulong((ulong_t *)&rnew);
3022        atomic_add_long((ulong_t *)&rnew, -1);
3023 #ifdef DEBUG
3024        clstat_debug.nrnode.value.ui64--;
3025 #endif
3026        nfs_rw_destroy(&rp->r_rwlock);
3027        nfs_rw_destroy(&rp->r_lkserlock);
3028        mutex_destroy(&rp->r_statelock);
3029        cv_destroy(&rp->r_cv);
3030        cv_destroy(&rp->r_commit.c_cv);
```



```
3031        if (rp->r_flags & RDELMAPLIST)
3032                list_destroy(&rp->r_indelmap);
3033        nfs_free_r_path(rp);
3034        avl_destroy(&rp->r_dir);
3035        vn_invalid(vp);
3036        vn_free(vp);
3037        kmem_cache_free(rnode_cache, rp);
3038        VFS_RELE(vfsp);
3039 }
```

unchanged_portion_omitted_

```
3803 rddir_cache *
3804 rddir_cache_alloc(int flags)
3805 {
3806        rddir_cache *rc;

3808        rc = kmem_alloc(sizeof (*rc), flags);
3809        if (rc != NULL) {
3810                rc->entries = NULL;
3811                rc->flags = RDDIR;
3812                cv_init(&rc->cv, NULL, CV_DEFAULT, NULL);
3813                mutex_init(&rc->lock, NULL, MUTEX_DEFAULT, NULL);
3814                rc->count = 1;
3815 #ifdef DEBUG
3816                atomic_inc_64(&clstat_debug.dirent.value.ui64);
3816                atomic_add_64(&clstat_debug.dirent.value.ui64, 1);
3817 #endif
3818        }
3819        return (rc);
3820 }

3822 static void
3823 rddir_cache_free(rddir_cache *rc)
3824 {

3826 #ifdef DEBUG
3827        atomic_dec_64(&clstat_debug.dirent.value.ui64);
3827        atomic_add_64(&clstat_debug.dirent.value.ui64, -1);
3828 #endif
3829        if (rc->entries != NULL) {
3830 #ifdef DEBUG
3831                rddir_cache_buf_free(rc->entries, rc->buflen);
3832 #else
3833                kmem_free(rc->entries, rc->buflen);
3834 #endif
3835        }
3836        cv_destroy(&rc->cv);
3837        mutex_destroy(&rc->lock);
3838        kmem_free(rc, sizeof (*rc));
3839 }
```

unchanged_portion_omitted_

****
130971 Mon Jul 28 07:44:18 2014
new/usr/src/uts/common/fs/nfs/nfs_vnops.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____unchanged_portion_omitted_

```
4301 /* ARGSUSED */
4302 static int
4303 nfs_map(vnode_t *vp, offset_t off, struct as *as, caddr_t *addrp,
4304         size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
4305         caller_context_t *ct)
4306 {
4307         struct segvn_crargs vn_a;
4308         int error;
4309         rnode_t *rp;
4310         struct vattr va;

4312         if (nfs_zone() != VTOMI(vp)->mi_zone)
4313                 return (EIO);

4315         if (vp->v_flag & VNOMAP)
4316                 return (ENOSYS);

4318         if (off > MAXOFF32_T)
4319                 return (EFBIG);

4321         if (off < 0 || off + len < 0)
4322                 return (ENXIO);

4324         if (vp->v_type != VREG)
4325                 return (ENODEV);

4327         /*
4328          * If there is cached data and if close-to-open consistency
4329          * checking is not turned off and if the file system is not
4330          * mounted readonly, then force an over the wire getattr.
4331          * Otherwise, just invoke nfsgetattr to get a copy of the
4332          * attributes.  The attribute cache will be used unless it
4333          * is timed out and if it is, then an over the wire getattr
4334          * will be issued.
4335          */
4336         va.va_mask = AT_ALL;
4337         if (vn_has_cached_data(vp) &&
4338             !(VTOMI(vp)->mi_flags & MI_NOCTO) && !vn_is_readonly(vp))
4339                 error = nfs_getattr_otw(vp, &va, cr);
4340         else
4341                 error = nfsgetattr(vp, &va, cr);
4342         if (error)
4343                 return (error);

4345         /*
4346          * Check to see if the vnode is currently marked as not cachable.
4347          * This means portions of the file are locked (through VOP_FRLOCK).
4348          * In this case the map request must be refused.  We use
4349          * rp->r_lkserlock to avoid a race with concurrent lock requests.
4350          */
4351         rp = VTOR(vp);

4353         /*
4354          * Atomically increment r_inmap after acquiring r_rwlock. The
4355          * idea here is to acquire r_rwlock to block read/write and
4356          * not to protect r_inmap. r_inmap will inform nfs_read/write()
4357          * that we are in nfs_map(). Now, r_rwlock is acquired in order
4358          * and we can prevent the deadlock that would have occurred
4359          * when nfs_addmap() would have acquired it out of order.
```



```
4360          *
4361          * Since we are not protecting r_inmap by any lock, we do not
4362          * hold any lock when we decrement it. We atomically decrement
4363          * r_inmap after we release r_lkserlock.
4364          */

4366         if (nfs_rw_enter_sig(&rp->r_rwlock, RW_WRITER, INTR(vp)))
4367                 return (EINTR);
4368         atomic_inc_uint(&rp->r_inmap);
4368         atomic_add_int(&rp->r_inmap, 1);
4369         nfs_rw_exit(&rp->r_rwlock);

4371         if (nfs_rw_enter_sig(&rp->r_lkserlock, RW_READER, INTR(vp))) {
4372                 atomic_dec_uint(&rp->r_inmap);
4372                 atomic_add_int(&rp->r_inmap, -1);
4373                 return (EINTR);
4374         }
4375         if (vp->v_flag & VNOCACHE) {
4376                 error = EAGAIN;
4377                 goto done;
4378         }

4380         /*
4381          * Don't allow concurrent locks and mapping if mandatory locking is
4382          * enabled.
4383          */
4384         if ((flk_has_remote_locks(vp) || lm_has_sleep(vp)) &&
4385             MANDLOCK(vp, va.va_mode)) {
4386                 error = EAGAIN;
4387                 goto done;
4388         }

4390         as_rangelock(as);
4391         error = choose_addr(as, addrp, len, off, ADDR_VACALIGN, flags);
4392         if (error != 0) {
4393                 as_rangeunlock(as);
4394                 goto done;
4395         }

4397         vn_a.vp = vp;
4398         vn_a.offset = off;
4399         vn_a.type = (flags & MAP_TYPE);
4400         vn_a.prot = (uchar_t)prot;
4401         vn_a.maxprot = (uchar_t)maxprot;
4402         vn_a.flags = (flags & ~MAP_TYPE);
4403         vn_a.cred = cr;
4404         vn_a.amp = NULL;
4405         vn_a.szc = 0;
4406         vn_a.lgrp_mem_policy_flags = 0;

4408         error = as_map(as, *addrp, len, segvn_create, &vn_a);
4409         as_rangeunlock(as);

4411 done:
4412         nfs_rw_exit(&rp->r_lkserlock);
4413         atomic_dec_uint(&rp->r_inmap);
4413         atomic_add_int(&rp->r_inmap, -1);
4414         return (error);
4415 }
```

_____unchanged_portion_omitted_

****
6095 Mon Jul 28 07:44:18 2014
new/usr/src/uts/common/fs/objfs/objfs_vfs.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
154 /*
155  * VFS entry points
156  */
157 static int
158 objfs_mount(vfs_t *vfsp, vnode_t *mvp, struct mounta *uap, cred_t *cr)
159 {
160         objfs_vfs_t *data;
161         dev_t dev;

163         if (secpolicy_fs_mount(cr, mvp, vfsp) != 0)
164                 return (EPERM);

166         if (mvp->v_type != VDIR)
167                 return (ENOTDIR);

169         if ((uap->flags & MS_OVERLAY) == 0 &&
170             (mvp->v_count > 1 || (mvp->v_flag & VROOT)))
171                 return (EBUSY);

173         data = kmem_alloc(sizeof (objfs_vfs_t), KM_SLEEP);

175         /*
176          * Initialize vfs fields
177          */
178         vfsp->vfs_bsize = DEV_BSIZE;
179         vfsp->vfs_fstype = objfs_fstype;
180         do {
181                 dev = makedevice(objfs_major,
182                     atomic_inc_32_nv(&objfs_minor) & L_MAXMIN32);
182                     atomic_add_32_nv(&objfs_minor, 1) & L_MAXMIN32);
183         } while (vfs_devismounted(dev));
184         vfs_make_fsid(&vfsp->vfs_fsid, dev, objfs_fstype);
185         vfsp->vfs_data = data;
186         vfsp->vfs_dev = dev;

188         /*
189          * Create root
190          */
191         data->objfs_vfs_root = objfs_create_root(vfsp);

193         return (0);
194 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  142900 Mon Jul 28 07:44:18 2014
new/usr/src/uts/common/fs/proc/prvnops.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

4427 #if defined(DEBUG)

4429 static  uint32_t nprnode;
4430 static  uint32_t nprcommon;

4432 #define INCREMENT(x)    atomic_inc_32(&x);
4433 #define DECREMENT(x)    atomic_dec_32(&x);
4432 #define INCREMENT(x)    atomic_add_32(&x, 1);
4433 #define DECREMENT(x)    atomic_add_32(&x, -1);

4435 #else

4437 #define INCREMENT(x)
4438 #define DECREMENT(x)

4440 #endif  /* DEBUG */

4442 /*
4443  * New /proc vnode required; allocate it and fill in most of the fields.
4444  */
4445 prnode_t *
4446 prgetnode(vnode_t *dp, prnodetype_t type)
4447 {
4448         prnode_t *pnp;
4449         prcommon_t *pcp;
4450         vnode_t *vp;
4451         ulong_t nfiles;

4453         INCREMENT(nprnode);
4454         pnp = kmem_zalloc(sizeof (prnode_t), KM_SLEEP);

4456         mutex_init(&pnp->pr_mutex, NULL, MUTEX_DEFAULT, NULL);
4457         pnp->pr_type = type;

4459         pnp->pr_vnode = vn_alloc(KM_SLEEP);

4461         vp = PTOV(pnp);
4462         vp->v_flag = VNOCACHE|VNOMAP|VNOSWAP|VNOMOUNT;
4463         vn_setops(vp, prvnodeops);
4464         vp->v_vfsp = dp->v_vfsp;
4465         vp->v_type = VPROC;
4466         vp->v_data = (caddr_t)pnp;

4468         switch (type) {
4469         case PR_PIDDIR:
4470         case PR_LWPIDDIR:
4471                 /*
4472                  * We need a prcommon and a files array for each of these.
4473                  */
4474                 INCREMENT(nprcommon);

4476                 pcp = kmem_zalloc(sizeof (prcommon_t), KM_SLEEP);
4477                 pcp->prc_refcnt = 1;
4478                 pnp->pr_common = pcp;
4479                 mutex_init(&pcp->prc_mutex, NULL, MUTEX_DEFAULT, NULL);
4480                 cv_init(&pcp->prc_wait, NULL, CV_DEFAULT, NULL);

4482                 nfiles = (type == PR_PIDDIR)? NPIDDIRFILES : NLWPIDDIRFILES;
4483                 pnp->pr_files =
```



```
4484                     kmem_zalloc(nfiles * sizeof (vnode_t *), KM_SLEEP);

4486                 vp->v_type = VDIR;
4487                 /*
4488                  * Mode should be read-search by all, but we cannot so long
4489                  * as we must support compatibility mode with old /proc.
4490                  * Make /proc/<pid> be read by owner only, search by all.
4491                  * Make /proc/<pid>/lwp/<lwpid> read-search by all.  Also,
4492                  * set VDIROPEN on /proc/<pid> so it can be opened for writing.
4493                  */
4494                 if (type == PR_PIDDIR) {
4495                         /* kludge for old /proc interface */
4496                         prnode_t *xpnp = prgetnode(dp, PR_PIDFILE);
4497                         pnp->pr_pidfile = PTOV(xpnp);
4498                         pnp->pr_mode = 0511;
4499                         vp->v_flag |= VDIROPEN;
4500                 } else {
4501                         pnp->pr_mode = 0555;
4502                 }

4504                 break;

4506         case PR_CURDIR:
4507         case PR_ROOTDIR:
4508         case PR_FDDIR:
4509         case PR_OBJECTDIR:
4510         case PR_PATHDIR:
4511         case PR_CTDIR:
4512         case PR_TMPLDIR:
4513                 vp->v_type = VDIR;
4514                 pnp->pr_mode = 0500;    /* read-search by owner only */
4515                 break;

4517         case PR_CT:
4518                 vp->v_type = VLNK;
4519                 pnp->pr_mode = 0500;    /* read-search by owner only */
4520                 break;

4522         case PR_PATH:
4523         case PR_SELF:
4524                 vp->v_type = VLNK;
4525                 pnp->pr_mode = 0777;
4526                 break;

4528         case PR_LWPDIR:
4529                 vp->v_type = VDIR;
4530                 pnp->pr_mode = 0555;    /* read-search by all */
4531                 break;

4533         case PR_AS:
4534         case PR_TMPL:
4535                 pnp->pr_mode = 0600;    /* read-write by owner only */
4536                 break;

4538         case PR_CTL:
4539         case PR_LWPCTL:
4540                 pnp->pr_mode = 0200;    /* write-only by owner only */
4541                 break;

4543         case PR_PIDFILE:
4544         case PR_LWPIDFILE:
4545                 pnp->pr_mode = 0600;    /* read-write by owner only */
4546                 break;

4548         case PR_PSINFO:
4549         case PR_LPSINFO:
```

```
4550         case PR_LWPSINFO:
4551         case PR_USAGE:
4552         case PR_LUSAGE:
4553         case PR_LWPUSAGE:
4554                 pnp->pr_mode = 0444;    /* read-only by all */
4555                 break;

4557         default:
4558                 pnp->pr_mode = 0400;    /* read-only by owner only */
4559                 break;
4560         }
4561         vn_exists(vp);
4562         return (pnp);
4563 }
```

_____***unchanged_portion_omitted_***

****
6253 Mon Jul 28 07:44:19 2014
new/usr/src/uts/common/fs/sharefs/sharefs_vfsops.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
179 /*
180  * VFS entry points
181  */
182 static int
183 sharefs_mount(vfs_t *vfsp, vnode_t *mvp, struct mounta *uap, cred_t *cr)
184 {
185         sharefs_vfs_t   *data;
186         dev_t           dev;

188         if (secpolicy_fs_mount(cr, mvp, vfsp) != 0)
189                 return (EPERM);

191         if ((uap->flags & MS_OVERLAY) == 0 &&
192             (mvp->v_count > 1 || (mvp->v_flag & VROOT)))
193                 return (EBUSY);

195         data = kmem_alloc(sizeof (sharefs_vfs_t), KM_SLEEP);

197         /*
198          * Initialize vfs fields
199          */
200         vfsp->vfs_bsize = DEV_BSIZE;
201         vfsp->vfs_fstype = sharefs_fstype;
202         do {
203                 dev = makedevice(sharefs_major,
204                     atomic_inc_32_nv(&sharefs_minor) & L_MAXMIN32);
204                     atomic_add_32_nv(&sharefs_minor, 1) & L_MAXMIN32);
205         } while (vfs_devismounted(dev));
206         vfs_make_fsid(&vfsp->vfs_fsid, dev, sharefs_fstype);
207         vfsp->vfs_data = data;
208         vfsp->vfs_dev = dev;

210         /*
211          * Create root
212          */
213         data->sharefs_vfs_root = sharefs_create_root_file(vfsp);

215         return (0);
216 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    8435 Mon Jul 28 07:44:19 2014
new/usr/src/uts/common/fs/sharefs/sharefs_vnops.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <fs/fs_subr.h>

#include <sys/errno.h>
#include <sys/file.h>
#include <sys/kmem.h>
#include <sys/kobj.h>
#include <sys/cmn_err.h>
#include <sys/stat.h>
#include <sys/systm.h>
#include <sys/sysmacros.h>
#include <sys/atomic.h>
#include <sys/vfs.h>
#include <sys/vfs_opreg.h>

#include <sharefs/sharefs.h>

/*
 * sharefs_snap_create: create a large character buffer with
 * the shares enumerated.
 */
static int
sharefs_snap_create(shnode_t *sft)
{
        sharetab_t              *sht;
        share_t                 *sh;
        size_t                  sWritten = 0;
        int                     iCount = 0;
        char                    *buf;

        rw_enter(&sharefs_lock, RW_WRITER);
        rw_enter(&sharetab_lock, RW_READER);

        if (sft->sharefs_snap) {
                /*
                 * Nothing has changed, so no need to grab a new copy!
                 */
                if (sft->sharefs_generation == sharetab_generation) {
                        rw_exit(&sharetab_lock);
                        rw_exit(&sharefs_lock);
                        return (0);
                }

                ASSERT(sft->sharefs_size != 0);
                kmem_free(sft->sharefs_snap, sft->sharefs_size + 1);
                sft->sharefs_snap = NULL;
        }

        sft->sharefs_size = sharetab_size;
        sft->sharefs_count = sharetab_count;

        if (sft->sharefs_size == 0) {
                rw_exit(&sharetab_lock);
                rw_exit(&sharefs_lock);
                return (0);
        }

        sft->sharefs_snap = kmem_zalloc(sft->sharefs_size + 1, KM_SLEEP);

        buf = sft->sharefs_snap;

        /*
         * Walk the Sharetab, dumping each entry.
         */
        for (sht = sharefs_sharetab; sht != NULL; sht = sht->s_next) {
                int     i;

                for (i = 0; i < SHARETAB_HASHES; i++) {
                        for (sh = sht->s_buckets[i].ssh_sh;
                            sh != NULL;
                            sh = sh->sh_next) {
                                int     n;

                                if ((sWritten + sh->sh_size) >
                                    sft->sharefs_size) {
                                        goto error_fault;
                                }

                                /*
                                 * Note that sh->sh_size accounts
                                 * for the field seperators.
                                 * We need to add one for the EOL
                                 * marker. And we should note that
                                 * the space is accounted for in
                                 * each share by the EOS marker.
                                 */
                                n = snprintf(&buf[sWritten],
                                    sh->sh_size + 1,
                                    "%s\t%s\t%s\t%s\t%s\n",
                                    sh->sh_path,
                                    sh->sh_res,
                                    sh->sh_fstype,
                                    sh->sh_opts,
                                    sh->sh_descr);

                                if (n != sh->sh_size) {
                                        goto error_fault;
                                }

                                sWritten += n;
```

```
126                                 iCount++;
127                         }
128                 }
129         }

131         /*
132          * We want to record the generation number and
133          * mtime inside this snapshot.
134          */
135         gethrestime(&sharetab_snap_time);
136         sft->sharefs_snap_time = sharetab_snap_time;
137         sft->sharefs_generation = sharetab_generation;

139         ASSERT(iCount == sft->sharefs_count);

141         rw_exit(&sharetab_lock);
142         rw_exit(&sharefs_lock);
143         return (0);

145 error_fault:

147         kmem_free(sft->sharefs_snap, sft->sharefs_size + 1);
148         sft->sharefs_size = 0;
149         sft->sharefs_count = 0;
150         sft->sharefs_snap = NULL;
151         rw_exit(&sharetab_lock);
152         rw_exit(&sharefs_lock);

154         return (EFAULT);
155 }
______unchanged_portion_omitted_

215 /* ARGSUSED */
216 int
217 sharefs_open(vnode_t **vpp, int flag, cred_t *cr, caller_context_t *ct)
218 {
219         vnode_t         *vp;
220         vnode_t         *ovp = *vpp;
221         shnode_t        *sft;
222         int             error = 0;

224         if (flag & FWRITE)
225                 return (EINVAL);

227         /*
228          * Create a new sharefs vnode for each operation. In order to
229          * avoid locks, we create a snapshot which can not change during
230          * reads.
231          */
232         vp = gfs_file_create(sizeof (shnode_t), NULL, sharefs_ops_data);

234         ((gfs_file_t *)vp->v_data)->gfs_ino = SHAREFS_INO_FILE;

236         /*
237          * Hold the parent!
238          */
239         VFS_HOLD(ovp->v_vfsp);

241         VN_SET_VFS_TYPE_DEV(vp, ovp->v_vfsp, VREG, 0);

243         vp->v_flag |= VROOT | VNOCACHE | VNOMAP | VNOSWAP | VNOMOUNT;

245         *vpp = vp;
246         VN_RELE(ovp);

248         sft = VTOSH(vp);
```



```
250         /*
251          * No need for the lock, no other thread can be accessing
252          * this data structure.
253          */
254         atomic_inc_32(&sft->sharefs_refs);
256         atomic_add_32(&sft->sharefs_refs, 1);
255         sft->sharefs_real_vp = 0;

257         /*
258          * Since the sharetab could easily change on us whilst we
259          * are dumping an extremely huge sharetab, we make a copy
260          * of it here and use it to dump instead.
261          */
262         error = sharefs_snap_create(sft);

264         return (error);
265 }

267 /* ARGSUSED */
268 int
269 sharefs_close(vnode_t *vp, int flag, int count,
270     offset_t off, cred_t *cr, caller_context_t *ct)
271 {
272         shnode_t        *sft = VTOSH(vp);

274         if (count > 1)
275                 return (0);

277         rw_enter(&sharefs_lock, RW_WRITER);
278         if (vp->v_count == 1) {
279                 if (sft->sharefs_snap != NULL) {
280                         kmem_free(sft->sharefs_snap, sft->sharefs_size + 1);
281                         sft->sharefs_size = 0;
282                         sft->sharefs_snap = NULL;
283                         sft->sharefs_generation = 0;
284                 }
285         }
286         atomic_dec_32(&sft->sharefs_refs);
288         atomic_add_32(&sft->sharefs_refs, -1);
287         rw_exit(&sharefs_lock);

289         return (0);
290 }
______unchanged_portion_omitted_
```

```
**********************************************************
    9548 Mon Jul 28 07:44:19 2014
new/usr/src/uts/common/fs/sharefs/sharetab.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 106 /*
 107  * If there is no error, then this function is responsible for
 108  * cleaning up the memory associated with the share argument.
 109  */
 110 static int
 111 sharefs_remove(share_t *sh, sharefs_lens_t *shl)
 112 {
 113         int             iHash;
 114         sharetab_t      *sht;
 115         share_t         *s, *p;
 116         int             iPath;

 118         if (!sh)
 119                 return (ENOENT);

 121         rw_enter(&sharetab_lock, RW_WRITER);
 122         for (sht = sharefs_sharetab; sht != NULL; sht = sht->s_next) {
 123                 if (strcmp(sh->sh_fstype, sht->s_fstype) == 0) {
 124                         break;
 125                 }
 126         }

 128         /*
 129          * There does not exist a fstype in memory which
 130          * matches the share passed in.
 131          */
 132         if (!sht) {
 133                 rw_exit(&sharetab_lock);
 134                 return (ENOENT);
 135         }

 137         iPath = shl ? shl->shl_path : strlen(sh->sh_path);
 138         iHash = pkp_tab_hash(sh->sh_path, strlen(sh->sh_path));

 140         /*
 141          * Now walk down the hash table and find the entry to free!
 142          */
 143         for (p = NULL, s = sht->s_buckets[iHash].ssh_sh;
 144             s != NULL; s = s->sh_next) {
 145                 /*
 146                  * We need exact matches.
 147                  */
 148                 if (strcmp(sh->sh_path, s->sh_path) == 0 &&
 149                     strlen(s->sh_path) == iPath) {
 150                         if (p) {
 151                                 p->sh_next = s->sh_next;
 152                         } else {
 153                                 sht->s_buckets[iHash].ssh_sh = s->sh_next;
 154                         }

 156                         ASSERT(sht->s_buckets[iHash].ssh_count != 0);
 157                         atomic_dec_32(&sht->s_buckets[iHash].ssh_count);
 158                         atomic_dec_32(&sht->s_count);
 159                         atomic_dec_32(&sharetab_count);
 157                         atomic_add_32(&sht->s_buckets[iHash].ssh_count, -1);
 158                         atomic_add_32(&sht->s_count, -1);
 159                         atomic_add_32(&sharetab_count, -1);

 161                         ASSERT(sharetab_size >= s->sh_size);
```



```
 162                         sharetab_size -= s->sh_size;

 164                         gethrestime(&sharetab_mtime);
 165                         atomic_inc_32(&sharetab_generation);
 165                         atomic_add_32(&sharetab_generation, 1);

 167                         break;
 168                 }

 170                 p = s;
 171         }

 173         rw_exit(&sharetab_lock);

 175         if (!s) {
 176                 return (ENOENT);
 177         }

 179         s->sh_next = NULL;
 180         sharefree(s, NULL);

 182         /*
 183          * We need to free the share for the caller.
 184          */
 185         sharefree(sh, shl);

 187         return (0);
 188 }

 190 /*
 191  * The caller must have allocated memory for us to use.
 192  */
 193 static int
 194 sharefs_add(share_t *sh, sharefs_lens_t *shl)
 195 {
 196         int             iHash;
 197         sharetab_t      *sht;
 198         share_t         *s, *p;
 199         int             iPath;
 200         int             n;

 202         if (!sh) {
 203                 return (ENOENT);
 204         }

 206         /*
 207          * We need to find the hash buckets for the fstype.
 208          */
 209         rw_enter(&sharetab_lock, RW_WRITER);
 210         for (sht = sharefs_sharetab; sht != NULL; sht = sht->s_next) {
 211                 if (strcmp(sh->sh_fstype, sht->s_fstype) == 0) {
 212                         break;
 213                 }
 214         }

 216         /*
 217          * Did not exist, so allocate one and add it to the
 218          * sharetab.
 219          */
 220         if (!sht) {
 221                 sht = kmem_zalloc(sizeof (*sht), KM_SLEEP);
 222                 n = strlen(sh->sh_fstype);
 223                 sht->s_fstype = kmem_zalloc(n + 1, KM_SLEEP);
 224                 (void) strncpy(sht->s_fstype, sh->sh_fstype, n);

 226                 sht->s_next = sharefs_sharetab;
```

```
227                 sharefs_sharetab = sht;
228         }

230         /*
231          * Now we need to find where we have to add the entry.
232          */
233         iHash = pkp_tab_hash(sh->sh_path, strlen(sh->sh_path));

235         iPath = shl ? shl->shl_path : strlen(sh->sh_path);

237         if (shl) {
238                 sh->sh_size = shl->shl_path + shl->shl_res +
239                     shl->shl_fstype + shl->shl_opts + shl->shl_descr;
240         } else {
241                 sh->sh_size = strlen(sh->sh_path) +
242                     strlen(sh->sh_res) + strlen(sh->sh_fstype) +
243                     strlen(sh->sh_opts) + strlen(sh->sh_descr);
244         }

246         /*
247          * We need to account for field seperators and
248          * the EOL.
249          */
250         sh->sh_size += 5;

252         /*
253          * Now walk down the hash table and add the new entry!
254          */
255         for (p = NULL, s = sht->s_buckets[iHash].ssh_sh;
256             s != NULL; s = s->sh_next) {
257                 /*
258                  * We need exact matches.
259                  *
260                  * We found a matching path. Either we have a
261                  * duplicate path in a share command or we are
262                  * being asked to replace an existing entry.
263                  */
264                 if (strcmp(sh->sh_path, s->sh_path) == 0 &&
265                     strlen(s->sh_path) == iPath) {
266                         if (p) {
267                                 p->sh_next = sh;
268                         } else {
269                                 sht->s_buckets[iHash].ssh_sh = sh;
270                         }

272                         sh->sh_next = s->sh_next;

274                         ASSERT(sharetab_size >= s->sh_size);
275                         sharetab_size -= s->sh_size;
276                         sharetab_size += sh->sh_size;

278                         /*
279                          * Get rid of the old node.
280                          */
281                         sharefree(s, NULL);

283                         gethrestime(&sharetab_mtime);
284                         atomic_inc_32(&sharetab_generation);
284                         atomic_add_32(&sharetab_generation, 1);

286                         ASSERT(sht->s_buckets[iHash].ssh_count != 0);
287                         rw_exit(&sharetab_lock);

289                         return (0);
290                 }
```



```
292                 p = s;
293         }

295         /*
296          * Okay, we have gone through the entire hash chain and not
297          * found a match. We just need to add this node.
298          */
299         sh->sh_next = sht->s_buckets[iHash].ssh_sh;
300         sht->s_buckets[iHash].ssh_sh = sh;
301         atomic_inc_32(&sht->s_buckets[iHash].ssh_count);
302         atomic_inc_32(&sht->s_count);
303         atomic_inc_32(&sharetab_count);
301         atomic_add_32(&sht->s_buckets[iHash].ssh_count, 1);
302         atomic_add_32(&sht->s_count, 1);
303         atomic_add_32(&sharetab_count, 1);
304         sharetab_size += sh->sh_size;

306         gethrestime(&sharetab_mtime);
307         atomic_inc_32(&sharetab_generation);
307         atomic_add_32(&sharetab_generation, 1);

309         rw_exit(&sharetab_lock);

311         return (0);
312 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
   28729 Mon Jul 28 07:44:19 2014
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_subr2.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 304 /*
 305  * NFS: nfs_subr.c:rtablehash
 306  * We use smbfs_hash().
 307  */

 309 /*
 310  * Find or create an smbnode.
 311  * NFS: nfs_subr.c:make_rnode
 312  */
 313 static smbnode_t *
 314 make_smbnode(
 315         smbmntinfo_t *mi,
 316         const char *rpath,
 317         int rplen,
 318         int *newnode)
 319 {
 320         smbnode_t *np;
 321         smbnode_t *tnp;
 322         vnode_t *vp;
 323         vfs_t *vfsp;
 324         avl_index_t where;
 325         char *new_rpath = NULL;

 327         ASSERT(RW_READ_HELD(&mi->smi_hash_lk));
 328         vfsp = mi->smi_vfsp;

 330 start:
 331         np = sn_hashfind(mi, rpath, rplen, NULL);
 332         if (np != NULL) {
 333                 *newnode = 0;
 334                 return (np);
 335         }

 337         /* Note: will retake this lock below. */
 338         rw_exit(&mi->smi_hash_lk);

 340         /*
 341          * see if we can find something on the freelist
 342          */
 343         mutex_enter(&smbfreelist_lock);
 344         if (smbfreelist != NULL && smbnodenew >= nsmbnode) {
 345                 np = smbfreelist;
 346                 sn_rmfree(np);
 347                 mutex_exit(&smbfreelist_lock);

 349                 vp = SMBTOV(np);

 351                 if (np->r_flags & RHASHED) {
 352                         smbmntinfo_t *tmp_mi = np->n_mount;
 353                         ASSERT(tmp_mi != NULL);
 354                         rw_enter(&tmp_mi->smi_hash_lk, RW_WRITER);
 355                         mutex_enter(&vp->v_lock);
 356                         if (vp->v_count > 1) {
 357                                 vp->v_count--;
 358                                 mutex_exit(&vp->v_lock);
 359                                 rw_exit(&tmp_mi->smi_hash_lk);
 360                                 /* start over */
 361                                 rw_enter(&mi->smi_hash_lk, RW_READER);
 362                                 goto start;
```



```
 363                         }
 364                         mutex_exit(&vp->v_lock);
 365                         sn_rmhash_locked(np);
 366                         rw_exit(&tmp_mi->smi_hash_lk);
 367                 }

 369                 sn_inactive(np);

 371                 mutex_enter(&vp->v_lock);
 372                 if (vp->v_count > 1) {
 373                         vp->v_count--;
 374                         mutex_exit(&vp->v_lock);
 375                         rw_enter(&mi->smi_hash_lk, RW_READER);
 376                         goto start;
 377                 }
 378                 mutex_exit(&vp->v_lock);
 379                 vn_invalid(vp);
 380                 /*
 381                  * destroy old locks before bzero'ing and
 382                  * recreating the locks below.
 383                  */
 384                 smbfs_rw_destroy(&np->r_rwlock);
 385                 smbfs_rw_destroy(&np->r_lkserlock);
 386                 mutex_destroy(&np->r_statelock);
 387                 cv_destroy(&np->r_cv);
 388                 /*
 389                  * Make sure that if smbnode is recycled then
 390                  * VFS count is decremented properly before
 391                  * reuse.
 392                  */
 393                 VFS_RELE(vp->v_vfsp);
 394                 vn_reinit(vp);
 395         } else {
 396                 /*
 397                  * allocate and initialize a new smbnode
 398                  */
 399                 vnode_t *new_vp;

 401                 mutex_exit(&smbfreelist_lock);

 403                 np = kmem_cache_alloc(smbnode_cache, KM_SLEEP);
 404                 new_vp = vn_alloc(KM_SLEEP);

 406                 atomic_inc_ulong((ulong_t *)&smbnodenew);
 406                 atomic_add_long((ulong_t *)&smbnodenew, 1);
 407                 vp = new_vp;
 408         }

 410         /*
 411          * Allocate and copy the rpath we'll need below.
 412          */
 413         new_rpath = kmem_alloc(rplen + 1, KM_SLEEP);
 414         bcopy(rpath, new_rpath, rplen);
 415         new_rpath[rplen] = '\0';

 417         /* Initialize smbnode_t */
 418         bzero(np, sizeof (*np));

 420         smbfs_rw_init(&np->r_rwlock, NULL, RW_DEFAULT, NULL);
 421         smbfs_rw_init(&np->r_lkserlock, NULL, RW_DEFAULT, NULL);
 422         mutex_init(&np->r_statelock, NULL, MUTEX_DEFAULT, NULL);
 423         cv_init(&np->r_cv, NULL, CV_DEFAULT, NULL);
 424         /* cv_init(&np->r_commit.c_cv, NULL, CV_DEFAULT, NULL); */

 426         np->r_vnode = vp;
 427         np->n_mount = mi;
```

```
 429         np->n_fid = SMB_FID_UNUSED;
 430         np->n_uid = mi->smi_uid;
 431         np->n_gid = mi->smi_gid;
 432         /* Leave attributes "stale." */

 434 #if 0 /* XXX dircache */
 435         /*
 436          * We don't know if it's a directory yet.
 437          * Let the caller do this?  XXX
 438          */
 439         avl_create(&np->r_dir, compar, sizeof (rddir_cache),
 440             offsetof(rddir_cache, tree));
 441 #endif

 443         /* Now fill in the vnode. */
 444         vn_setops(vp, smbfs_vnodeops);
 445         vp->v_data = (caddr_t)np;
 446         VFS_HOLD(vfsp);
 447         vp->v_vfsp = vfsp;
 448         vp->v_type = VNON;

 450         /*
 451          * We entered with mi->smi_hash_lk held (reader).
 452          * Retake it now, (as the writer).
 453          * Will return with it held.
 454          */
 455         rw_enter(&mi->smi_hash_lk, RW_WRITER);

 457         /*
 458          * There is a race condition where someone else
 459          * may alloc the smbnode while no locks are held,
 460          * so check again and recover if found.
 461          */
 462         tnp = sn_hashfind(mi, rpath, rplen, &where);
 463         if (tnp != NULL) {
 464                 /*
 465                  * Lost the race.  Put the node we were building
 466                  * on the free list and return the one we found.
 467                  */
 468                 rw_exit(&mi->smi_hash_lk);
 469                 kmem_free(new_rpath, rplen + 1);
 470                 smbfs_addfree(np);
 471                 rw_enter(&mi->smi_hash_lk, RW_READER);
 472                 *newnode = 0;
 473                 return (tnp);
 474         }

 476         /*
 477          * Hash search identifies nodes by the remote path
 478          * (n_rpath) so fill that in now, before linking
 479          * this node into the node cache (AVL tree).
 480          */
 481         np->n_rpath = new_rpath;
 482         np->n_rplen = rplen;
 483         np->n_ino = smbfs_gethash(new_rpath, rplen);

 485         sn_addhash_locked(np, where);
 486         *newnode = 1;
 487         return (np);
 488 }
```

unchanged_portion_omitted_

```
1007 /*
1008  * This routine destroys all the resources associated with the smbnode
1009  * and then the smbnode itself.  Note: sn_inactive has been called.
```



```
1010  *
1011  * NFS: nfs_subr.c:destroy_rnode
1012  */
1013 static void
1014 sn_destroy_node(smbnode_t *np)
1015 {
1016         vnode_t *vp;
1017         vfs_t *vfsp;

1019         vp = SMBTOV(np);
1020         vfsp = vp->v_vfsp;

1022         ASSERT(vp->v_count == 1);
1023         ASSERT(np->r_count == 0);
1024         ASSERT(np->r_mapcnt == 0);
1025         ASSERT(np->r_secattr.vsa_aclentp == NULL);
1026         ASSERT(np->r_cred == NULL);
1027         ASSERT(np->n_rpath == NULL);
1028         ASSERT(!(np->r_flags & RHASHED));
1029         ASSERT(np->r_freef == NULL && np->r_freeb == NULL);
1030         atomic_dec_ulong((ulong_t *)&smbnodenew);
1030         atomic_add_long((ulong_t *)&smbnodenew, -1);
1031         vn_invalid(vp);
1032         vn_free(vp);
1033         kmem_cache_free(smbnode_cache, np);
1034         VFS_RELE(vfsp);
1035 }
```

unchanged_portion_omitted_

```
**********************************************************
   53613 Mon Jul 28 07:44:19 2014
new/usr/src/uts/common/fs/sockfs/nl7curi.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 267 #define URI_HASH_IX(hix, which) (hix) = (hix) % (uri_hash_sz[(which)])

 269 #define URI_HASH_MIGRATE(from, hp, to) {                                \
 270         uri_desc_t      *_nuri;                                         \
 271         uint32_t        _nhix;                                          \
 272         uri_hash_t      *_nhp;                                          \
 273                                                                         \
 274         mutex_enter(&(hp)->lock);                                       \
 275         while ((_nuri = (hp)->list) != NULL) {                          \
 276                 (hp)->list = _nuri->hash;                               \
 277                 atomic_dec_32(&uri_hash_cnt[(from)]);           \
 278                 atomic_inc_32(&uri_hash_cnt[(to)]);             \
 277                 atomic_add_32(&uri_hash_cnt[(from)], -1);               \
 278                 atomic_add_32(&uri_hash_cnt[(to)], 1);                  \
 279                 _nhix = _nuri->hvalue;                                  \
 280                 URI_HASH_IX(_nhix, to);                                 \
 281                 _nhp = &uri_hash_ab[(to)][_nhix];                       \
 282                 mutex_enter(&_nhp->lock);                               \
 283                 _nuri->hash = _nhp->list;                               \
 284                 _nhp->list = _nuri;                                     \
 285                 _nuri->hit = 0;                                         \
 286                 mutex_exit(&_nhp->lock);                                \
 287         }                                                               \
 288         mutex_exit(&(hp)->lock);                                        \
 289 }

 291 #define URI_HASH_UNLINK(cur, new, hp, puri, uri) {                      \
 292         if ((puri) != NULL) {                                           \
 293                 (puri)->hash = (uri)->hash;                             \
 294         } else {                                                        \
 295                 (hp)->list = (uri)->hash;                               \
 296         }                                                               \
 297         if (atomic_dec_32_nv(&uri_hash_cnt[(cur)]) == 0 &&              \
 297         if (atomic_add_32_nv(&uri_hash_cnt[(cur)], -1) == 0 &&          \
 298             uri_hash_ab[(new)] != NULL) {                               \
 299                 kmem_free(uri_hash_ab[cur],                             \
 300                     sizeof (uri_hash_t) * uri_hash_sz[cur]);            \
 301                 uri_hash_ab[(cur)] = NULL;                              \
 302                 uri_hash_lru[(cur)] = NULL;                             \
 303                 uri_hash_which = (new);                                 \
 304         } else {                                                        \
 305                 uri_hash_lru[(cur)] = (hp);                             \
 306         }                                                               \
 307 }
______unchanged_portion_omitted_

 564 /*
 565  * Add a uri_desc_t to the URI hash.
 566  */

 568 static void
 569 uri_add(uri_desc_t *uri, krw_t rwlock, boolean_t nonblocking)
 570 {
 571         uint32_t        hix;
 572         uri_hash_t      *hp;
 573         uint32_t        cur = uri_hash_which;
 574         uint32_t        new = cur ? 0 : 1;

 576         /*
```



```
 577          * Caller of uri_add() must hold the uri_hash_access rwlock.
 578          */
 579         ASSERT((rwlock == RW_READER && RW_READ_HELD(&uri_hash_access)) ||
 580             (rwlock == RW_WRITER && RW_WRITE_HELD(&uri_hash_access)));
 581         /*
 582          * uri_add() always succeeds so add a hash ref to the URI now.
 583          */
 584         REF_HOLD(uri);
 585 again:
 586         hix = uri->hvalue;
 587         URI_HASH_IX(hix, cur);
 588         if (uri_hash_ab[new] == NULL &&
 589             uri_hash_cnt[cur] < uri_hash_overflow[cur]) {
 590                 /*
 591                  * Easy case, no new hash and current hasn't overflowed,
 592                  * add URI to current hash and return.
 593                  *
 594                  * Note, the check for uri_hash_cnt[] above aren't done
 595                  * atomictally, i.e. multiple threads can be in this code
 596                  * as RW_READER and update the cnt[], this isn't a problem
 597                  * as the check is only advisory.
 598                  */
 599         fast:
 600                 atomic_inc_32(&uri_hash_cnt[cur]);
 600                 atomic_add_32(&uri_hash_cnt[cur], 1);
 601                 hp = &uri_hash_ab[cur][hix];
 602                 mutex_enter(&hp->lock);
 603                 uri->hash = hp->list;
 604                 hp->list = uri;
 605                 mutex_exit(&hp->lock);
 606                 rw_exit(&uri_hash_access);
 607                 return;
 608         }
 609         if (uri_hash_ab[new] == NULL) {
 610                 /*
 611                  * Need a new a or b hash, if not already RW_WRITER
 612                  * try to upgrade our lock to writer.
 613                  */
 614                 if (rwlock != RW_WRITER && ! rw_tryupgrade(&uri_hash_access)) {
 615                         /*
 616                          * Upgrade failed, we can't simple exit and reenter
 617                          * the lock as after the exit and before the reenter
 618                          * the whole world can change so just wait for writer
 619                          * then do everything again.
 620                          */
 621                         if (nonblocking) {
 622                                 /*
 623                                  * Can't block, use fast-path above.
 624                                  *
 625                                  * XXX should have a background thread to
 626                                  * handle new ab[] in this case so as to
 627                                  * not overflow the cur hash to much.
 628                                  */
 629                                 goto fast;
 630                         }
 631                         rw_exit(&uri_hash_access);
 632                         rwlock = RW_WRITER;
 633                         rw_enter(&uri_hash_access, rwlock);
 634                         cur = uri_hash_which;
 635                         new = cur ? 0 : 1;
 636                         goto again;
 637                 }
 638                 rwlock = RW_WRITER;
 639                 if (uri_hash_ab[new] == NULL) {
 640                         /*
 641                          * Still need a new hash, allocate and initialize
```

```
642                          * the new hash.
643                          */
644                         uri_hash_n[new] = uri_hash_n[cur] + 1;
645                         if (uri_hash_n[new] == 0) {
646                                 /*
647                                  * No larger P2Ps[] value so use current,
648                                  * i.e. 2 of the largest are better than 1 ?
649                                  */
650                                 uri_hash_n[new] = uri_hash_n[cur];
651                                 cmn_err(CE_NOTE, "NL7C: hash index overflow");
652                         }
653                         uri_hash_sz[new] = P2Ps[uri_hash_n[new]];
654                         ASSERT(uri_hash_cnt[new] == 0);
655                         uri_hash_overflow[new] = uri_hash_sz[new] *
656                             URI_HASH_AVRG;
657                         uri_hash_ab[new] = kmem_zalloc(sizeof (uri_hash_t) *
658                             uri_hash_sz[new], nonblocking ? KM_NOSLEEP :
659                             KM_SLEEP);
660                         if (uri_hash_ab[new] == NULL) {
661                                 /*
662                                  * Alloc failed, use fast-path above.
663                                  *
664                                  * XXX should have a background thread to
665                                  * handle new ab[] in this case so as to
666                                  * not overflow the cur hash to much.
667                                  */
668                                 goto fast;
669                         }
670                         uri_hash_lru[new] = uri_hash_ab[new];
671                 }
672         }
673         /*
674          * Hashed against current hash so migrate any current hash chain
675          * members, if any.
676          *
677          * Note, the hash chain list can be checked for a non empty list
678          * outside of the hash chain list lock as the hash chain struct
679          * can't be destroyed while in the uri_hash_access rwlock, worst
680          * case is that a non empty list is found and after acquiring the
681          * lock another thread beats us to it (i.e. migrated the list).
682          */
683         hp = &uri_hash_ab[cur][hix];
684         if (hp->list != NULL) {
685                 URI_HASH_MIGRATE(cur, hp, new);
686         }
687         /*
688          * If new hash has overflowed before current hash has been
689          * completely migrated then walk all current hash chains and
690          * migrate list members now.
691          */
692         if (atomic_inc_32_nv(&uri_hash_cnt[new]) >= uri_hash_overflow[new]) {
692         if (atomic_add_32_nv(&uri_hash_cnt[new], 1) >= uri_hash_overflow[new]) {
693                 for (hix = 0; hix < uri_hash_sz[cur]; hix++) {
694                         hp = &uri_hash_ab[cur][hix];
695                         if (hp->list != NULL) {
696                                 URI_HASH_MIGRATE(cur, hp, new);
697                         }
698                 }
699         }
700         /*
701          * Add URI to new hash.
702          */
703         hix = uri->hvalue;
704         URI_HASH_IX(hix, new);
705         hp = &uri_hash_ab[new][hix];
706         mutex_enter(&hp->lock);
```



```
707         uri->hash = hp->list;
708         hp->list = uri;
709         mutex_exit(&hp->lock);
710         /*
711          * Last, check to see if last cur hash chain has been
712          * migrated, if so free cur hash and make new hash cur.
713          */
714         if (uri_hash_cnt[cur] == 0) {
715                 /*
716                  * If we don't already hold the uri_hash_access rwlock for
717                  * RW_WRITE try to upgrade to RW_WRITE and if successful
718                  * check again and to see if still need to do the free.
719                  */
720                 if ((rwlock == RW_WRITER || rw_tryupgrade(&uri_hash_access)) &&
721                     uri_hash_cnt[cur] == 0 && uri_hash_ab[new] != 0) {
722                         kmem_free(uri_hash_ab[cur],
723                             sizeof (uri_hash_t) * uri_hash_sz[cur]);
724                         uri_hash_ab[cur] = NULL;
725                         uri_hash_lru[cur] = NULL;
726                         uri_hash_which = new;
727                 }
728         }
729         rw_exit(&uri_hash_access);
730 }

732 /*
733  * Lookup a uri_desc_t in the URI hash, if found free the request uri_desc_t
734  * and return the found uri_desc_t with a REF_HOLD() placed on it. Else, if
735  * add B_TRUE use the request URI to create a new hash entry. Else if add
736  * B_FALSE ...
737  */

739 static uri_desc_t *
740 uri_lookup(uri_desc_t *ruri, boolean_t add, boolean_t nonblocking)
741 {
742         uint32_t        hix;
743         uri_hash_t      *hp;
744         uri_desc_t      *uri;
745         uri_desc_t      *puri;
746         uint32_t        cur;
747         uint32_t        new;
748         char            *rcp = ruri->path.cp;
749         char            *rep = ruri->path.ep;

751 again:
752         rw_enter(&uri_hash_access, RW_READER);
753         cur = uri_hash_which;
754         new = cur ? 0 : 1;
755 nexthash:
756         puri = NULL;
757         hix = ruri->hvalue;
758         URI_HASH_IX(hix, cur);
759         hp = &uri_hash_ab[cur][hix];
760         mutex_enter(&hp->lock);
761         for (uri = hp->list; uri != NULL; uri = uri->hash) {
762                 char    *ap = uri->path.cp;
763                 char    *bp = rcp;
764                 char    a, b;

766                 /* Compare paths */
767                 while (bp < rep && ap < uri->path.ep) {
768                         if ((a = *ap) == '%') {
769                                 /* Escaped hex multichar, convert it */
770                                 H2A(ap, uri->path.ep, a);
771                         }
772                         if ((b = *bp) == '%') {
```

```
773                             /* Escaped hex multichar, convert it */
774                             H2A(bp, rep, b);
775                     }
776                     if (a != b) {
777                             /* Char's don't match */
778                             goto nexturi;
779                     }
780                     ap++;
781                     bp++;
782             }
783             if (bp != rep || ap != uri->path.ep) {
784                     /* Not same length */
785                     goto nexturi;
786             }
787             ap = uri->auth.cp;
788             bp = ruri->auth.cp;
789             if (ap != NULL) {
790                     if (bp == NULL) {
791                             /* URI has auth request URI doesn't */
792                             goto nexturi;
793                     }
794                     while (bp < ruri->auth.ep && ap < uri->auth.ep) {
795                             if ((a = *ap) == '%') {
796                                     /* Escaped hex multichar, convert it */
797                                     H2A(ap, uri->path.ep, a);
798                             }
799                             if ((b = *bp) == '%') {
800                                     /* Escaped hex multichar, convert it */
801                                     H2A(bp, rep, b);
802                             }
803                             if (a != b) {
804                                     /* Char's don't match */
805                                     goto nexturi;
806                             }
807                             ap++;
808                             bp++;
809                     }
810                     if (bp != ruri->auth.ep || ap != uri->auth.ep) {
811                             /* Not same length */
812                             goto nexturi;
813                     }
814             } else if (bp != NULL) {
815                     /* URI doesn't have auth and request URI does */
816                     goto nexturi;
817             }
818             /*
819              * Have a path/auth match so before any other processing
820              * of requested URI, check for expire or request no cache
821              * purge.
822              */
823             if (uri->expire >= 0 && uri->expire <= ddi_get_lbolt() ||
824                 ruri->nocache) {
825                     /*
826                      * URI has expired or request specified to not use
827                      * the cached version, unlink the URI from the hash
828                      * chain, release all locks, release the hash ref
829                      * on the URI, and last look it up again.
830                      *
831                      * Note, this will cause all variants of the named
832                      * URI to be purged.
833                      */
834                     if (puri != NULL) {
835                             puri->hash = uri->hash;
836                     } else {
837                             hp->list = uri->hash;
838                     }
```



```
839                     mutex_exit(&hp->lock);
840                     atomic_dec_32(&uri_hash_cnt[cur]);
840                     atomic_add_32(&uri_hash_cnt[cur], -1);
841                     rw_exit(&uri_hash_access);
842                     if (ruri->nocache)
843                             nl7c_uri_purge++;
844                     else
845                             nl7c_uri_expire++;
846                     REF_RELE(uri);
847                     goto again;
848             }
849             if (uri->scheme != NULL) {
850                     /*
851                      * URI has scheme private qualifier(s), if request
852                      * URI doesn't or if no match skip this URI.
853                      */
854                     if (ruri->scheme == NULL ||
855                         ! nl7c_http_cmp(uri->scheme, ruri->scheme))
856                             goto nexturi;
857             } else if (ruri->scheme != NULL) {
858                     /*
859                      * URI doesn't have scheme private qualifiers but
860                      * request URI does, no match, skip this URI.
861                      */
862                     goto nexturi;
863             }
864             /*
865              * Have a match, ready URI for return, first put a reference
866              * hold on the URI, if this URI is currently being processed
867              * then have to wait for the processing to be completed and
868              * redo the lookup, else return it.
869              */
870             REF_HOLD(uri);
871             mutex_enter(&uri->proclock);
872             if (uri->proc != NULL) {
873                     /* The URI is being processed, wait for completion */
874                     mutex_exit(&hp->lock);
875                     rw_exit(&uri_hash_access);
876                     if (! nonblocking &&
877                         cv_wait_sig(&uri->waiting, &uri->proclock)) {
878                             /*
879                              * URI has been processed but things may
880                              * have changed while we were away so do
881                              * most everything again.
882                              */
883                             mutex_exit(&uri->proclock);
884                             REF_RELE(uri);
885                             goto again;
886                     } else {
887                             /*
888                              * A nonblocking socket or an interrupted
889                              * cv_wait_sig() in the first case can't
890                              * block waiting for the processing of the
891                              * uri hash hit uri to complete, in both
892                              * cases just return failure to lookup.
893                              */
894                             mutex_exit(&uri->proclock);
895                             REF_RELE(uri);
896                             return (NULL);
897                     }
898             }
899             mutex_exit(&uri->proclock);
900             uri->hit++;
901             mutex_exit(&hp->lock);
902             rw_exit(&uri_hash_access);
903             return (uri);
```

```
904         nexturi:
905                 puri = uri;
906         }
907         mutex_exit(&hp->lock);
908         if (cur != new && uri_hash_ab[new] != NULL) {
909                 /*
910                  * Not found in current hash and have a new hash so
911                  * check the new hash next.
912                  */
913                 cur = new;
914                 goto nexthash;
915         }
916 add:
917         if (! add) {
918                 /* Lookup only so return failure */
919                 rw_exit(&uri_hash_access);
920                 return (NULL);
921         }
922         /*
923          * URI not hashed, finish intialization of the
924          * request URI, add it to the hash, return it.
925          */
926         ruri->hit = 0;
927         ruri->expire = -1;
928         ruri->response.sz = 0;
929         ruri->proc = (struct sonode *)~NULL;
930         cv_init(&ruri->waiting, NULL, CV_DEFAULT, NULL);
931         mutex_init(&ruri->proclock, NULL, MUTEX_DEFAULT, NULL);
932         uri_add(ruri, RW_READER, nonblocking);
933         /* uri_add() has done rw_exit(&uri_hash_access) */
934         return (ruri);
935 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    5596 Mon Jul 28 07:44:20 2014
new/usr/src/uts/common/fs/sockfs/nl7curi.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _SYS_SOCKFS_NL7CURI_H
#define _SYS_SOCKFS_NL7CURI_H

#pragma ident   "%Z%%M% %I%     %E% SMI"

#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/types.h>
#include <sys/atomic.h>
#include <sys/cmn_err.h>
#include <sys/stropts.h>
#include <sys/socket.h>
#include <sys/socketvar.h>

#undef  PROMIF_DEBUG

/*
 * Some usefull chararcter macros:
 */

#ifndef tolower
#define tolower(c) ((c) >= 'A' && (c) <= 'Z' ? (c) | 0x20 : (c))
#endif

#ifndef isdigit
#define isdigit(c) ((c) >= '0' && (c) <= '9')
#endif

#ifndef isalpha
#define isalpha(c) (((c) >= 'A' && (c) <= 'Z') || ((c) >= 'a' && (c) <= 'z'))
#endif

#ifndef isspace
```



```
#define isspace(c) ((c) == ' ' || (c) == '\t' || (c) == '\n' || \
                (c) == '\r' || (c) == '\f' || (c) == '\013')
#endif

/*
 * ref_t - reference type, ...
 *
 * Note, all struct's must contain a single ref_t, all must use
 * kmem_cache, all must use the REF_* macros for free.
 */

typedef struct ref_s {
        uint32_t        cnt;            /* Reference count */
        void            (*last)(void *); /* Call-back for last ref */
        kmem_cache_t    *kmc;           /* Container allocator cache */
} ref_t;
______unchanged_portion_omitted_

#define REF_HOLD(container) {                                           \
        atomic_inc_32(&(container)->ref.cnt);                           \
        atomic_add_32(&(container)->ref.cnt, 1);                        \
        ASSERT((container)->ref.cnt != 0);                              \
}

#define REF_RELE(container) {                                           \
        if (atomic_dec_32_nv(&(container)->ref.cnt) == 0) {             \
        if (atomic_add_32_nv(&(container)->ref.cnt, -1) == 0) {         \
                (container)->ref.last((container));                     \
                kmem_cache_free((container)->ref.kmc, (container));     \
        }                                                               \
}
______unchanged_portion_omitted_
```

```
**********************************************************
    7226 Mon Jul 28 07:44:20 2014
new/usr/src/uts/common/fs/sockfs/sockfilter_impl.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

  65 #define SOF_GLOBAL_STAT_BUMP(s) \
  66         atomic_inc_64(&sof_stat.sofks_##s.value.ui64)
  66         atomic_add_64(&sof_stat.sofks_##s.value.ui64, 1)

  68 /*
  69  * Per filter statistics.
  70  */
  71 struct sof_entry_kstat {
  72         kstat_named_t   sofek_nactive;          /* # of consumers */
  73         kstat_named_t   sofek_tot_active_attach;
  74         kstat_named_t   sofek_tot_passive_attach;
  75         kstat_named_t   sofek_ndeferred;        /* # of deferred conns */
  76         kstat_named_t   sofek_attach_failures;
  77 };
______unchanged_portion_omitted_
```

****
78813 Mon Jul 28 07:44:20 2014
new/usr/src/uts/common/fs/sockfs/socksyscalls.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
2396 /*
2397  * The callback function used for vpm mapped mblks called when the last ref of
2398  * the mblk is dropped which normally occurs when TCP receives the ack. But it
2399  * can be the driver too due to lazy reclaim.
2400  */
2401 void
2402 snf_vmap_desbfree(snf_vmap_desbinfo *snfv)
2403 {
2404         ASSERT(snfv->snfv_ref != 0);
2405         if (atomic_dec_32_nv(&snfv->snfv_ref) == 0) {
2405         if (atomic_add_32_nv(&snfv->snfv_ref, -1) == 0) {
2406                 vpm_unmap_pages(snfv->snfv_vml, S_READ);
2407                 VN_RELE(snfv->snfv_vp);
2408                 kmem_free(snfv, sizeof (snf_vmap_desbinfo));
2409         }
2410 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   38477 Mon Jul 28 07:44:20 2014
new/usr/src/uts/common/fs/ufs/lufs.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 844 /*
 845  * Disable logging
 846  */
 847 int
 848 lufs_disable(vnode_t *vp, struct fiolog *flp)
 849 {
 850         int             error = 0;
 851         inode_t         *ip = VTOI(vp);
 852         ufsvfs_t        *ufsvfsp = ip->i_ufsvfs;
 853         struct fs       *fs = ufsvfsp->vfs_fs;
 854         struct lockfs   lf;
 855         struct ulockfs  *ulp;

 857         flp->error = FIOLOG_ENONE;

 859         /*
 860          * Logging is already disabled; done
 861          */
 862         if (fs->fs_logbno == 0 || ufsvfsp->vfs_log == NULL)
 863                 return (0);

 865         /*
 866          * Readonly file system
 867          */
 868         if (fs->fs_ronly) {
 869                 flp->error = FIOLOG_EROFS;
 870                 return (0);
 871         }

 873         /*
 874          * File system must be write locked to disable logging
 875          */
 876         error = ufs_fiolfss(vp, &lf);
 877         if (error) {
 878                 return (error);
 879         }
 880         if (!LOCKFS_IS_ULOCK(&lf)) {
 881                 flp->error = FIOLOG_EULOCK;
 882                 return (0);
 883         }
 884         lf.lf_lock = LOCKFS_WLOCK;
 885         lf.lf_flags = 0;
 886         lf.lf_comment = NULL;
 887         error = ufs_fiolfs(vp, &lf, 1);
 888         if (error) {
 889                 flp->error = FIOLOG_EWLOCK;
 890                 return (0);
 891         }

 893         if (ufsvfsp->vfs_log == NULL || fs->fs_logbno == 0)
 894                 goto errout;

 896         /*
 897          * WE ARE COMMITTED TO DISABLING LOGGING PAST THIS POINT
 898          */

 900         /*
 901          * Disable logging:
 902          * Suspend the reclaim thread and force the delete thread to exit.
```



```
 903          *      When a nologging mount has completed there may still be
 904          *      work for reclaim to do so just suspend this thread until
 905          *      it's [deadlock-] safe for it to continue.  The delete
 906          *      thread won't be needed as ufs_iinactive() calls
 907          *      ufs_delete() when logging is disabled.
 908          * Freeze and drain reader ops.
 909          *      Commit any outstanding reader transactions (ufs_flush).
 910          *      Set the ''unmounted'' bit in the ufstrans struct.
 911          *      If debug, remove metadata from matamap.
 912          *      Disable matamap processing.
 913          *      NULL the trans ops table.
 914          *      Free all of the incore structs related to logging.
 915          * Allow reader ops.
 916          */
 917         ufs_thread_suspend(&ufsvfsp->vfs_reclaim);
 918         ufs_thread_exit(&ufsvfsp->vfs_delete);

 920         vfs_lock_wait(ufsvfsp->vfs_vfs);
 921         ulp = &ufsvfsp->vfs_ulockfs;
 922         mutex_enter(&ulp->ul_lock);
 923         atomic_inc_ulong(&ufs_quiesce_pend);
 923         atomic_add_long(&ufs_quiesce_pend, 1);
 924         (void) ufs_quiesce(ulp);

 926         (void) ufs_flush(ufsvfsp->vfs_vfs);

 928         TRANS_MATA_UMOUNT(ufsvfsp);
 929         ufsvfsp->vfs_domatamap = 0;

 931         /*
 932          * Free all of the incore structs
 933          * Aquire the ufs_scan_lock before de-linking the mtm data
 934          * structure so that we keep ufs_sync() and ufs_update() away
 935          * when they execute the ufs_scan_inodes() run while we're in
 936          * progress of enabling/disabling logging.
 937          */
 938         mutex_enter(&ufs_scan_lock);
 939         (void) lufs_unsnarf(ufsvfsp);
 940         mutex_exit(&ufs_scan_lock);

 942         atomic_dec_ulong(&ufs_quiesce_pend);
 942         atomic_add_long(&ufs_quiesce_pend, -1);
 943         mutex_exit(&ulp->ul_lock);
 944         vfs_setmntopt(ufsvfsp->vfs_vfs, MNTOPT_NOLOGGING, NULL, 0);
 945         vfs_unlock(ufsvfsp->vfs_vfs);

 947         fs->fs_rolled = FS_ALL_ROLLED;
 948         ufsvfsp->vfs_nolog_si = 0;

 950         /*
 951          * Free the log space and mark the superblock as FSACTIVE
 952          */
 953         (void) lufs_free(ufsvfsp);

 955         /*
 956          * Allow the reclaim thread to continue.
 957          */
 958         ufs_thread_continue(&ufsvfsp->vfs_reclaim);

 960         /*
 961          * Unlock the file system
 962          */
 963         lf.lf_lock = LOCKFS_ULOCK;
 964         lf.lf_flags = 0;
 965         error = ufs_fiolfs(vp, &lf, 1);
 966         if (error)
```

```
 967                 flp->error = FIOLOG_ENOULOCK;

 969         return (0);

 971 errout:
 972         lf.lf_lock = LOCKFS_ULOCK;
 973         lf.lf_flags = 0;
 974         (void) ufs_fiolfs(vp, &lf, 1);
 975         return (error);
 976 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   24057 Mon Jul 28 07:44:20 2014
new/usr/src/uts/common/fs/ufs/ufs_directio.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 330 uint32_t        ufs_shared_writes;      /* writes done w/ lock shared */
 331 uint32_t        ufs_cur_writes;         /* # concurrent writes */
 332 uint32_t        ufs_maxcur_writes;      /* high water concurrent writes */
 333 uint32_t        ufs_posix_hits;         /* writes done /w lock excl. */

 335 /*
 336  * Force POSIX syncronous data integrity on all writes for testing.
 337  */
 338 uint32_t        ufs_force_posix_sdi = 0;

 340 /*
 341  * Direct Write
 342  */

 344 int
 345 ufs_directio_write(struct inode *ip, uio_t *arg_uio, int ioflag, int rewrite,
 346         cred_t *cr, int *statusp)
 347 {
 348         long            resid, bytes_written;
 349         u_offset_t      size, uoff;
 350         uio_t           *uio = arg_uio;
 351         rlim64_t        limit = uio->uio_llimit;
 352         int             on, n, error, newerror, len, has_holes;
 353         daddr_t         bn;
 354         size_t          nbytes;
 355         struct fs       *fs;
 356         vnode_t         *vp;
 357         iovec_t         *iov;
 358         struct ufsvfs   *ufsvfsp = ip->i_ufsvfs;
 359         struct proc     *procp;
 360         struct as       *as;
 361         struct directio_buf     *tail;
 362         int             exclusive, ncur, bmap_peek;
 363         uio_t           copy_uio;
 364         iovec_t         copy_iov;
 365         char            *copy_base;
 366         long            copy_resid;

 368         /*
 369          * assume that directio isn't possible (normal case)
 370          */
 371         *statusp = DIRECTIO_FAILURE;

 373         /*
 374          * Don't go direct
 375          */
 376         if (ufs_directio_enabled == 0)
 377                 return (0);

 379         /*
 380          * mapped file; nevermind
 381          */
 382         if (ip->i_mapcnt)
 383                 return (0);

 385         /*
 386          * CAN WE DO DIRECT IO?
 387          */
 388         uoff = uio->uio_loffset;
```



```
 389         resid = uio->uio_resid;

 391         /*
 392          * beyond limit
 393          */
 394         if (uoff + resid > limit)
 395                 return (0);

 397         /*
 398          * must be sector aligned
 399          */
 400         if ((uoff & (u_offset_t)(DEV_BSIZE - 1)) || (resid & (DEV_BSIZE - 1)))
 401                 return (0);

 403         /*
 404          * SHOULD WE DO DIRECT IO?
 405          */
 406         size = ip->i_size;
 407         has_holes = -1;

 409         /*
 410          * only on regular files; no metadata
 411          */
 412         if (((ip->i_mode & IFMT) != IFREG) || ip->i_ufsvfs->vfs_qinod == ip)
 413                 return (0);

 415         /*
 416          * Synchronous, allocating writes run very slow in Direct-Mode
 417          *      XXX - can be fixed with bmap_write changes for large writes!!!
 418          *      XXX - can be fixed for updates to "almost-full" files
 419          *      XXX - WARNING - system hangs if bmap_write() has to
 420          *                      allocate lots of pages since pageout
 421          *                      suspends on locked inode
 422          */
 423         if (!rewrite && (ip->i_flag & ISYNC)) {
 424                 if ((uoff + resid) > size)
 425                         return (0);
 426                 has_holes = bmap_has_holes(ip);
 427                 if (has_holes)
 428                         return (0);
 429         }

 431         /*
 432          * Each iovec must be short aligned and sector aligned.  If
 433          * one is not, then kmem_alloc a new buffer and copy all of
 434          * the smaller buffers into the new buffer.  This new
 435          * buffer will be short aligned and sector aligned.
 436          */
 437         iov = uio->uio_iov;
 438         nbytes = uio->uio_iovcnt;
 439         while (nbytes--) {
 440                 if (((uint_t)iov->iov_len & (DEV_BSIZE - 1)) != 0 ||
 441                     (intptr_t)(iov->iov_base) & 1) {
 442                         copy_resid = uio->uio_resid;
 443                         copy_base = kmem_alloc(copy_resid, KM_NOSLEEP);
 444                         if (copy_base == NULL)
 445                                 return (0);
 446                         copy_iov.iov_base = copy_base;
 447                         copy_iov.iov_len = copy_resid;
 448                         copy_uio.uio_iov = &copy_iov;
 449                         copy_uio.uio_iovcnt = 1;
 450                         copy_uio.uio_segflg = UIO_SYSSPACE;
 451                         copy_uio.uio_extflg = UIO_COPY_DEFAULT;
 452                         copy_uio.uio_loffset = uio->uio_loffset;
 453                         copy_uio.uio_resid = uio->uio_resid;
 454                         copy_uio.uio_llimit = uio->uio_llimit;
```

```
455                         error = uiomove(copy_base, copy_resid, UIO_WRITE, uio);
456                         if (error) {
457                                 kmem_free(copy_base, copy_resid);
458                                 return (0);
459                         }
460                         uio = &copy_uio;
461                         break;
462                 }
463                 iov++;
464         }

466         /*
467          * From here on down, all error exits must go to errout and
468          * not simply return a 0.
469          */

471         /*
472          * DIRECTIO
473          */

475         fs = ip->i_fs;

477         /*
478          * POSIX check. If attempting a concurrent re-write, make sure
479          * that this will be a single request to the driver to meet
480          * POSIX synchronous data integrity requirements.
481          */
482         bmap_peek = 0;
483         if (rewrite && ((ioflag & FDSYNC) || ufs_force_posix_sdi)) {
484                 int upgrade = 0;

486                 /* check easy conditions first */
487                 if (uio->uio_iovcnt != 1 || resid > ufsvfsp->vfs_ioclustsz) {
488                         upgrade = 1;
489                 } else {
490                         /* now look for contiguous allocation */
491                         len = (ssize_t)blkroundup(fs, resid);
492                         error = bmap_read(ip, uoff, &bn, &len);
493                         if (error || bn == UFS_HOLE || len == 0)
494                                 goto errout;
495                         /* save a call to bmap_read later */
496                         bmap_peek = 1;
497                         if (len < resid)
498                                 upgrade = 1;
499                 }
500                 if (upgrade) {
501                         rw_exit(&ip->i_contents);
502                         rw_enter(&ip->i_contents, RW_WRITER);
503                         ufs_posix_hits++;
504                 }
505         }


508         /*
509          * allocate space
510          */

512         /*
513          * If attempting a re-write, there is no allocation to do.
514          * bmap_write would trip an ASSERT if i_contents is held shared.
515          */
516         if (rewrite)
517                 goto skip_alloc;

519         do {
520                 on = (int)blkoff(fs, uoff);
```



```
521                 n = (int)MIN(fs->fs_bsize - on, resid);
522                 if ((uoff + n) > ip->i_size) {
523                         error = bmap_write(ip, uoff, (int)(on + n),
524                             (int)(uoff & (offset_t)MAXBOFFSET) == 0,
525                             NULL, cr);
526                         /* Caller is responsible for updating i_seq if needed */
527                         if (error)
528                                 break;
529                         ip->i_size = uoff + n;
530                         ip->i_flag |= IATTCHG;
531                 } else if (n == MAXBSIZE) {
532                         error = bmap_write(ip, uoff, (int)(on + n),
533                             BI_ALLOC_ONLY, NULL, cr);
534                         /* Caller is responsible for updating i_seq if needed */
535                 } else {
536                         if (has_holes < 0)
537                                 has_holes = bmap_has_holes(ip);
538                         if (has_holes) {
539                                 uint_t  blk_size;
540                                 u_offset_t offset;

542                                 offset = uoff & (offset_t)fs->fs_bmask;
543                                 blk_size = (int)blksize(fs, ip,
544                                     (daddr_t)lblkno(fs, offset));
545                                 error = bmap_write(ip, uoff, blk_size,
546                                     BI_NORMAL, NULL, cr);
547                                 /*
548                                  * Caller is responsible for updating
549                                  * i_seq if needed
550                                  */
551                         } else
552                                 error = 0;
553                 }
554                 if (error)
555                         break;
556                 uoff += n;
557                 resid -= n;
558                 /*
559                  * if file has grown larger than 2GB, set flag
560                  * in superblock if not already set
561                  */
562                 if ((ip->i_size > MAXOFF32_T) &&
563                     !(fs->fs_flags & FSLARGEFILES)) {
564                         ASSERT(ufsvfsp->vfs_lfflags & UFS_LARGEFILES);
565                         mutex_enter(&ufsvfsp->vfs_lock);
566                         fs->fs_flags |= FSLARGEFILES;
567                         ufs_sbwrite(ufsvfsp);
568                         mutex_exit(&ufsvfsp->vfs_lock);
569                 }
570         } while (resid);

572         if (error) {
573                 /*
574                  * restore original state
575                  */
576                 if (resid) {
577                         if (size == ip->i_size)
578                                 goto errout;
579                         (void) ufs_itrunc(ip, size, 0, cr);
580                 }
581                 /*
582                  * try non-directio path
583                  */
584                 goto errout;
585         }
586 skip_alloc:
```

```
588         /*
589          * get rid of cached pages
590          */
591         vp = ITOV(ip);
592         exclusive = rw_write_held(&ip->i_contents);
593         if (vn_has_cached_data(vp)) {
594                 if (!exclusive) {
595                         /*
596                          * Still holding i_rwlock, so no allocations
597                          * can happen after dropping contents.
598                          */
599                         rw_exit(&ip->i_contents);
600                         rw_enter(&ip->i_contents, RW_WRITER);
601                 }
602                 (void) VOP_PUTPAGE(vp, (offset_t)0, (size_t)0,
603                     B_INVAL, cr, NULL);
604                 if (vn_has_cached_data(vp))
605                         goto errout;
606                 if (!exclusive)
607                         rw_downgrade(&ip->i_contents);
608                 ufs_directio_kstats.nflushes.value.ui64++;
609         }

611         /*
612          * Direct Writes
613          */

615         if (!exclusive) {
616                 ufs_shared_writes++;
617                 ncur = atomic_inc_32_nv(&ufs_cur_writes);
617                 ncur = atomic_add_32_nv(&ufs_cur_writes, 1);
618                 if (ncur > ufs_maxcur_writes)
619                         ufs_maxcur_writes = ncur;
620         }

622         /*
623          * proc and as are for VM operations in directio_start()
624          */
625         if (uio->uio_segflg == UIO_USERSPACE) {
626                 procp = ttoproc(curthread);
627                 as = procp->p_as;
628         } else {
629                 procp = NULL;
630                 as = &kas;
631         }
632         *statusp = DIRECTIO_SUCCESS;
633         error = 0;
634         newerror = 0;
635         resid = uio->uio_resid;
636         bytes_written = 0;
637         ufs_directio_kstats.logical_writes.value.ui64++;
638         while (error == 0 && newerror == 0 && resid && uio->uio_iovcnt) {
639                 size_t pglck_len, pglck_size;
640                 caddr_t pglck_base;
641                 page_t **pplist, **spplist;

643                 tail = NULL;

645                 /*
646                  * Adjust number of bytes
647                  */
648                 iov = uio->uio_iov;
649                 pglck_len = (size_t)MIN(iov->iov_len, resid);
650                 pglck_base = iov->iov_base;
651                 if (pglck_len == 0) {
```



```
652                         uio->uio_iov++;
653                         uio->uio_iovcnt--;
654                         continue;
655                 }

657                 /*
658                  * Try to Lock down the largest chunck of pages possible.
659                  */
660                 pglck_len = (size_t)MIN(pglck_len,  ufsvfsp->vfs_ioclustsz);
661                 error = as_pagelock(as, &pplist, pglck_base, pglck_len, S_READ);

663                 if (error)
664                         break;

666                 pglck_size = pglck_len;
667                 while (pglck_len) {

669                         nbytes = pglck_len;
670                         uoff = uio->uio_loffset;

672                         if (!bmap_peek) {

674                                 /*
675                                  * Re-adjust number of bytes to contiguous
676                                  * range. May have already called bmap_read
677                                  * in the case of a concurrent rewrite.
678                                  */
679                                 len = (ssize_t)blkroundup(fs, nbytes);
680                                 error = bmap_read(ip, uoff, &bn, &len);
681                                 if (error)
682                                         break;
683                                 if (bn == UFS_HOLE || len == 0)
684                                         break;
685                         }
686                         nbytes = (size_t)MIN(nbytes, len);
687                         bmap_peek = 0;

689                         /*
690                          * Get the pagelist pointer for this offset to be
691                          * passed to directio_start.
692                          */

694                         if (pplist != NULL)
695                                 spplist = pplist +
696                                     btop((uintptr_t)iov->iov_base -
697                                     ((uintptr_t)pglck_base & PAGEMASK));
698                         else
699                                 spplist = NULL;

701                         /*
702                          * Kick off the direct write requests
703                          */
704                         directio_start(ufsvfsp, ip, nbytes, ldbtob(bn),
705                             iov->iov_base, S_READ, procp, &tail, spplist);

707                         /*
708                          * Adjust pointers and counters
709                          */
710                         iov->iov_len -= nbytes;
711                         iov->iov_base += nbytes;
712                         uio->uio_loffset += nbytes;
713                         resid -= nbytes;
714                         pglck_len -= nbytes;
715                 }

717                 /*
```

```
718                  * Wait for outstanding requests
719                  */
720                 newerror = directio_wait(tail, &bytes_written);

722                 /*
723                  * Release VM resources
724                  */
725                 as_pageunlock(as, pplist, pglck_base, pglck_size, S_READ);

727         }

729         if (!exclusive) {
730                 atomic_dec_32(&ufs_cur_writes);
730                 atomic_add_32(&ufs_cur_writes, -1);
731                 /*
732                  * If this write was done shared, readers may
733                  * have pulled in unmodified pages. Get rid of
734                  * these potentially stale pages.
735                  */
736                 if (vn_has_cached_data(vp)) {
737                         rw_exit(&ip->i_contents);
738                         rw_enter(&ip->i_contents, RW_WRITER);
739                         (void) VOP_PUTPAGE(vp, (offset_t)0, (size_t)0,
740                             B_INVAL, cr, NULL);
741                         ufs_directio_kstats.nflushes.value.ui64++;
742                         rw_downgrade(&ip->i_contents);
743                 }
744         }

746         /*
747          * If error, adjust resid to begin at the first
748          * un-writable byte.
749          */
750         if (error == 0)
751                 error = newerror;
752         if (error)
753                 resid = uio->uio_resid - bytes_written;
754         arg_uio->uio_resid = resid;

756         if (!rewrite) {
757                 ip->i_flag |= IUPD | ICHG;
758                 /* Caller will update i_seq */
759                 TRANS_INODE(ip->i_ufsvfs, ip);
760         }
761         /*
762          * If there is a residual; adjust the EOF if necessary
763          */
764         if (resid) {
765                 if (size != ip->i_size) {
766                         if (uio->uio_loffset > size)
767                                 size = uio->uio_loffset;
768                         (void) ufs_itrunc(ip, size, 0, cr);
769                 }
770         }

772         if (uio == &copy_uio)
773                 kmem_free(copy_base, copy_resid);

775         return (error);

777 errout:
778         if (uio == &copy_uio)
779                 kmem_free(copy_base, copy_resid);

781         return (0);
782 }
```

unchanged_portion_omitted_

```
**********************************************************
   16011 Mon Jul 28 07:44:21 2014
new/usr/src/uts/common/fs/ufs/ufs_filio.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 299 /*
 300  * ufs_fiosdio
 301  *      Set delayed-io state.  This ioctl is tailored
 302  *      to metamucil's needs and may change at any time.
 303  */
 304 int
 305 ufs_fiosdio(
 306         struct vnode    *vp,            /* file's vnode */
 307         uint_t          *diop,          /* dio flag */
 308         int             flag,           /* flag from ufs_ioctl */
 309         struct cred     *cr)            /* credentials from ufs_ioctl */
 310 {
 311         uint_t          dio;            /* copy of user's dio */
 312         struct inode    *ip;            /* inode for vp */
 313         struct ufsvfs   *ufsvfsp;
 314         struct fs       *fs;
 315         struct ulockfs  *ulp;
 316         int             error = 0;

 318 #ifdef lint
 319         flag = flag;
 320 #endif

 322         /* check input conditions */
 323         if (secpolicy_fs_config(cr, vp->v_vfsp) != 0)
 324                 return (EPERM);

 326         if (copyin(diop, &dio, sizeof (dio)))
 327                 return (EFAULT);

 329         if (dio > 1)
 330                 return (EINVAL);

 332         /* file system has been forcibly unmounted */
 333         if (VTOI(vp)->i_ufsvfs == NULL)
 334                 return (EIO);

 336         ip = VTOI(vp);
 337         ufsvfsp = ip->i_ufsvfs;
 338         ulp = &ufsvfsp->vfs_ulockfs;

 340         /* logging file system; dio ignored */
 341         if (TRANS_ISTRANS(ufsvfsp))
 342                 return (error);

 344         /* hold the mutex to prevent race with a lockfs request */
 345         vfs_lock_wait(vp->v_vfsp);
 346         mutex_enter(&ulp->ul_lock);
 347         atomic_inc_ulong(&ufs_quiesce_pend);
 347         atomic_add_long(&ufs_quiesce_pend, 1);

 349         if (ULOCKFS_IS_HLOCK(ulp)) {
 350                 error = EIO;
 351                 goto out;
 352         }

 354         if (ULOCKFS_IS_ELOCK(ulp)) {
 355                 error = EBUSY;
 356                 goto out;
```



```
 357         }
 358         /* wait for outstanding accesses to finish */
 359         if (error = ufs_quiesce(ulp))
 360                 goto out;

 362         /* flush w/invalidate */
 363         if (error = ufs_flush(vp->v_vfsp))
 364                 goto out;

 366         /*
 367          * update dio
 368          */
 369         mutex_enter(&ufsvfsp->vfs_lock);
 370         ufsvfsp->vfs_dio = dio;

 372         /*
 373          * enable/disable clean flag processing
 374          */
 375         fs = ip->i_fs;
 376         if (fs->fs_ronly == 0 &&
 377             fs->fs_clean != FSBAD &&
 378             fs->fs_clean != FSLOG) {
 379                 if (dio)
 380                         fs->fs_clean = FSSUSPEND;
 381                 else
 382                         fs->fs_clean = FSACTIVE;
 383                 ufs_sbwrite(ufsvfsp);
 384                 mutex_exit(&ufsvfsp->vfs_lock);
 385         } else
 386                 mutex_exit(&ufsvfsp->vfs_lock);
 387 out:
 388         /*
 389          * we need this broadcast because of the ufs_quiesce call above
 390          */
 391         atomic_dec_ulong(&ufs_quiesce_pend);
 391         atomic_add_long(&ufs_quiesce_pend, -1);
 392         cv_broadcast(&ulp->ul_cv);
 393         mutex_exit(&ulp->ul_lock);
 394         vfs_unlock(vp->v_vfsp);
 395         return (error);
 396 }

 398 /*
 399  * ufs_fioffs - ioctl handler for flushing file system
 400  */
 401 /* ARGSUSED */
 402 int
 403 ufs_fioffs(
 404         struct vnode    *vp,
 405         char            *vap,           /* must be NULL - reserved */
 406         struct cred     *cr)            /* credentials from ufs_ioctl */
 407 {
 408         int error;
 409         struct ufsvfs   *ufsvfsp;
 410         struct ulockfs  *ulp;

 412         /* file system has been forcibly unmounted */
 413         ufsvfsp = VTOI(vp)->i_ufsvfs;
 414         if (ufsvfsp == NULL)
 415                 return (EIO);

 417         ulp = &ufsvfsp->vfs_ulockfs;

 419         /*
 420          * suspend the delete thread
 421          *      this must be done outside the lockfs locking protocol
```

```
 422          */
 423         vfs_lock_wait(vp->v_vfsp);
 424         ufs_thread_suspend(&ufsvfsp->vfs_delete);

 426         /* hold the mutex to prevent race with a lockfs request */
 427         mutex_enter(&ulp->ul_lock);
 428         atomic_inc_ulong(&ufs_quiesce_pend);
 428         atomic_add_long(&ufs_quiesce_pend, 1);

 430         if (ULOCKFS_IS_HLOCK(ulp)) {
 431                 error = EIO;
 432                 goto out;
 433         }
 434         if (ULOCKFS_IS_ELOCK(ulp)) {
 435                 error = EBUSY;
 436                 goto out;
 437         }
 438         /* wait for outstanding accesses to finish */
 439         if (error = ufs_quiesce(ulp))
 440                 goto out;

 442         /*
 443          * If logging, and the logmap was marked as not rollable,
 444          * make it rollable now, and start the trans_roll thread and
 445          * the reclaim thread.  The log at this point is safe to write to.
 446          */
 447         if (ufsvfsp->vfs_log) {
 448                 ml_unit_t       *ul = ufsvfsp->vfs_log;
 449                 struct fs       *fsp = ufsvfsp->vfs_fs;
 450                 int             err;

 452                 if (ul->un_flags & LDL_NOROLL) {
 453                         ul->un_flags &= ~LDL_NOROLL;
 454                         logmap_start_roll(ul);
 455                         if (!fsp->fs_ronly && (fsp->fs_reclaim &
 456                             (FS_RECLAIM|FS_RECLAIMING))) {
 457                                 fsp->fs_reclaim &= ~FS_RECLAIM;
 458                                 fsp->fs_reclaim |= FS_RECLAIMING;
 459                                 ufs_thread_start(&ufsvfsp->vfs_reclaim,
 460                                     ufs_thread_reclaim, vp->v_vfsp);
 461                                 if (!fsp->fs_ronly) {
 462                                         TRANS_SBWRITE(ufsvfsp,
 463                                             TOP_SBUPDATE_UPDATE);
 464                                         if (err =
 465                                             geterror(ufsvfsp->vfs_bufp)) {
 466                                                 refstr_t        *mntpt;
 467                                                 mntpt = vfs_getmntpoint(
 468                                                     vp->v_vfsp);
 469                                                 cmn_err(CE_NOTE,
 470                                                     "Filesystem Flush "
 471                                                     "Failed to update "
 472                                                     "Reclaim Status for "
 473                                                     " %s, Write failed to "
 474                                                     "update superblock, "
 475                                                     "error %d",
 476                                                     refstr_value(mntpt),
 477                                                     err);
 478                                                 refstr_rele(mntpt);
 479                                         }
 480                                 }
 481                         }
 482                 }
 483         }

 485         /* synchronously flush dirty data and metadata */
 486         error = ufs_flush(vp->v_vfsp);
```



```
 488 out:
 489         atomic_dec_ulong(&ufs_quiesce_pend);
 489         atomic_add_long(&ufs_quiesce_pend, -1);
 490         cv_broadcast(&ulp->ul_cv);
 491         mutex_exit(&ulp->ul_lock);
 492         vfs_unlock(vp->v_vfsp);

 494         /*
 495          * allow the delete thread to continue
 496          */
 497         ufs_thread_continue(&ufsvfsp->vfs_delete);
 498         return (error);
 499 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   43440 Mon Jul 28 07:44:21 2014
new/usr/src/uts/common/fs/ufs/ufs_lockfs.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 871 /* kernel-internal interface, also used by fix-on-panic */
 872 int
 873 ufs__fiolfs(
 874         struct vnode *vp,
 875         struct lockfs *lockfsp,
 876         int from_user,
 877         int from_log)
 878 {
 879         struct ulockfs  *ulp;
 880         struct lockfs   lfs;
 881         int             error;
 882         struct vfs      *vfsp;
 883         struct ufsvfs   *ufsvfsp;
 884         int              errlck         = NO_ERRLCK;
 885         int              poll_events    = POLLPRI;
 886         extern struct pollhead ufs_pollhd;
 887         ulockfs_info_t *head;
 888         ulockfs_info_t *info;
 889         int signal = 0;

 891         /* check valid lock type */
 892         if (!lockfsp || lockfsp->lf_lock > LOCKFS_MAXLOCK)
 893                 return (EINVAL);

 895         if (!vp || !vp->v_vfsp || !vp->v_vfsp->vfs_data)
 896                 return (EIO);

 898         vfsp = vp->v_vfsp;

 900         if (vfsp->vfs_flag & VFS_UNMOUNTED) /* has been unmounted */
 901                 return (EIO);

 903         /* take the lock and check again */
 904         vfs_lock_wait(vfsp);
 905         if (vfsp->vfs_flag & VFS_UNMOUNTED) {
 906                 vfs_unlock(vfsp);
 907                 return (EIO);
 908         }

 910         /*
 911          * Can't wlock or ro/elock fs with accounting or local swap file
 912          * We need to check for this before we grab the ul_lock to avoid
 913          * deadlocks with the accounting framework.
 914          */
 915         if ((LOCKFS_IS_WLOCK(lockfsp) || LOCKFS_IS_ELOCK(lockfsp) ||
 916             LOCKFS_IS_ROELOCK(lockfsp)) && !from_log) {
 917                 if (ufs_checkaccton(vp) || ufs_checkswapon(vp)) {
 918                         vfs_unlock(vfsp);
 919                         return (EDEADLK);
 920                 }
 921         }

 923         ufsvfsp = (struct ufsvfs *)vfsp->vfs_data;
 924         ulp = &ufsvfsp->vfs_ulockfs;
 925         head = (ulockfs_info_t *)tsd_get(ufs_lockfs_key);
 926         SEARCH_ULOCKFSP(head, ulp, info);

 928         /*
 929          * Suspend both the reclaim thread and the delete thread.
```



```
 930          * This must be done outside the lockfs locking protocol.
 931          */
 932         ufs_thread_suspend(&ufsvfsp->vfs_reclaim);
 933         ufs_thread_suspend(&ufsvfsp->vfs_delete);

 935         mutex_enter(&ulp->ul_lock);
 936         atomic_inc_ulong(&ufs_quiesce_pend);
 936         atomic_add_long(&ufs_quiesce_pend, 1);

 938         /*
 939          * Quit if there is another lockfs request in progress
 940          * that is waiting for existing ufs_vnops to complete.
 941          */
 942         if (ULOCKFS_IS_BUSY(ulp)) {
 943                 error = EBUSY;
 944                 goto errexit;
 945         }

 947         /* cannot ulocked or downgrade a hard-lock */
 948         if (ULOCKFS_IS_HLOCK(ulp)) {
 949                 error = EIO;
 950                 goto errexit;
 951         }

 953         /* an error lock may be unlocked or relocked, only */
 954         if (ULOCKFS_IS_ELOCK(ulp)) {
 955                 if (!LOCKFS_IS_ULOCK(lockfsp) && !LOCKFS_IS_ELOCK(lockfsp)) {
 956                         error = EBUSY;
 957                         goto errexit;
 958                 }
 959         }

 961         /*
 962          * a read-only error lock may only be upgraded to an
 963          * error lock or hard lock
 964          */
 965         if (ULOCKFS_IS_ROELOCK(ulp)) {
 966                 if (!LOCKFS_IS_HLOCK(lockfsp) && !LOCKFS_IS_ELOCK(lockfsp)) {
 967                         error = EBUSY;
 968                         goto errexit;
 969                 }
 970         }

 972         /*
 973          * until read-only error locks are fully implemented
 974          * just return EINVAL
 975          */
 976         if (LOCKFS_IS_ROELOCK(lockfsp)) {
 977                 error = EINVAL;
 978                 goto errexit;
 979         }

 981         /*
 982          * an error lock may only be applied if the file system is
 983          * unlocked or already error locked.
 984          * (this is to prevent the case where a fs gets changed out from
 985          * underneath a fs that is locked for backup,
 986          * that is, name/delete/write-locked.)
 987          */
 988         if ((!ULOCKFS_IS_ULOCK(ulp) && !ULOCKFS_IS_ELOCK(ulp) &&
 989             !ULOCKFS_IS_ROELOCK(ulp)) &&
 990             (LOCKFS_IS_ELOCK(lockfsp) || LOCKFS_IS_ROELOCK(lockfsp))) {
 991                 error = EBUSY;
 992                 goto errexit;
 993         }
```

```
 995         /* get and validate the input lockfs request */
 996         if (error = ufs_getlfd(lockfsp, &ulp->ul_lockfs))
 997                 goto errexit;

 999         /*
1000          * save current ulockfs struct
1001          */
1002         bcopy(&ulp->ul_lockfs, &lfs, sizeof (struct lockfs));

1004         /*
1005          * Freeze the file system (pend future accesses)
1006          */
1007         ufs_freeze(ulp, lockfsp);

1009         /*
1010          * Set locking in progress because ufs_quiesce may free the
1011          * ul_lock mutex.
1012          */
1013         ULOCKFS_SET_BUSY(ulp);
1014         /* update the ioctl copy */
1015         LOCKFS_SET_BUSY(&ulp->ul_lockfs);

1017         /*
1018          * We  need to unset FWLOCK status before we call ufs_quiesce
1019          * so that the thread doesnt get suspended. We do this only if
1020          * this (fallocate) thread requested an unlock operation.
1021          */
1022         if (info && (info->flags & ULOCK_INFO_FALLOCATE)) {
1023                 if (!ULOCKFS_IS_WLOCK(ulp))
1024                         ULOCKFS_CLR_FWLOCK(ulp);
1025         }

1027         /*
1028          * Quiesce (wait for outstanding accesses to finish)
1029          */
1030         if (error = ufs_quiesce(ulp)) {
1031                 /*
1032                  * Interrupted due to signal. There could still be
1033                  * pending vnops.
1034                  */
1035                 signal = 1;

1037                 /*
1038                  * We do broadcast because lock-status
1039                  * could be reverted to old status.
1040                  */
1041                 cv_broadcast(&ulp->ul_cv);
1042                 goto errout;
1043         }

1045         /*
1046          * If the fallocate thread requested a write fs lock operation
1047          * then we set fwlock status in the ulp.
1048          */
1049         if (info && (info->flags & ULOCK_INFO_FALLOCATE)) {
1050                 if (ULOCKFS_IS_WLOCK(ulp))
1051                         ULOCKFS_SET_FWLOCK(ulp);
1052         }

1054         /*
1055          * save error lock status to pass down to reconcilation
1056          * routines and for later cleanup
1057          */
1058         if (LOCKFS_IS_ELOCK(&lfs) && ULOCKFS_IS_ULOCK(ulp))
1059                 errlck = UN_ERRLCK;
```



```
1061         if (ULOCKFS_IS_ELOCK(ulp) || ULOCKFS_IS_ROELOCK(ulp)) {
1062                 int needs_unlock;
1063                 int needs_sbwrite;

1065                 poll_events |= POLLERR;
1066                 errlck = LOCKFS_IS_ELOCK(&lfs) || LOCKFS_IS_ROELOCK(&lfs) ?
1067                     RE_ERRLCK : SET_ERRLCK;

1069                 needs_unlock = !MUTEX_HELD(&ufsvfsp->vfs_lock);
1070                 if (needs_unlock)
1071                         mutex_enter(&ufsvfsp->vfs_lock);

1073                 /* disable delayed i/o */
1074                 needs_sbwrite = 0;

1076                 if (errlck == SET_ERRLCK) {
1077                         ufsvfsp->vfs_fs->fs_clean = FSBAD;
1078                         needs_sbwrite = 1;
1079                 }

1081                 needs_sbwrite |= ufsvfsp->vfs_dio;
1082                 ufsvfsp->vfs_dio = 0;

1084                 if (needs_unlock)
1085                         mutex_exit(&ufsvfsp->vfs_lock);

1087                 if (needs_sbwrite) {
1088                         ulp->ul_sbowner = curthread;
1089                         TRANS_SBWRITE(ufsvfsp, TOP_SBWRITE_STABLE);

1091                         if (needs_unlock)
1092                                 mutex_enter(&ufsvfsp->vfs_lock);

1094                         ufsvfsp->vfs_fs->fs_fmod = 0;

1096                         if (needs_unlock)
1097                                 mutex_exit(&ufsvfsp->vfs_lock);
1098                 }
1099         }

1101         /*
1102          * reconcile superblock and inodes if was wlocked
1103          */
1104         if (LOCKFS_IS_WLOCK(&lfs) || LOCKFS_IS_ELOCK(&lfs)) {
1105                 if (error = ufs_reconcile(vfsp, ufsvfsp, errlck))
1106                         goto errout;
1107                 /*
1108                  * in case the fs grew; reset the metadata map for logging tests
1109                  */
1110                 TRANS_MATA_UMOUNT(ufsvfsp);
1111                 TRANS_MATA_MOUNT(ufsvfsp);
1112                 TRANS_MATA_SI(ufsvfsp, ufsvfsp->vfs_fs);
1113         }

1115         /*
1116          * At least everything *currently* dirty goes out.
1117          */

1119         if ((error = ufs_flush(vfsp)) != 0 && !ULOCKFS_IS_HLOCK(ulp) &&
1120             !ULOCKFS_IS_ELOCK(ulp))
1121                 goto errout;

1123         /*
1124          * thaw file system and wakeup pended processes
1125          */
1126         if (error = ufs_thaw(vfsp, ufsvfsp, ulp))
```

```
1127                 if (!ULOCKFS_IS_HLOCK(ulp) && !ULOCKFS_IS_ELOCK(ulp))
1128                         goto errout;

1130         /*
1131          * reset modified flag if not already write locked
1132          */
1133         if (!LOCKFS_IS_WLOCK(&lfs))
1134                 ULOCKFS_CLR_MOD(ulp);

1136         /*
1137          * idle the lock struct
1138          */
1139         ULOCKFS_CLR_BUSY(ulp);
1140         /* update the ioctl copy */
1141         LOCKFS_CLR_BUSY(&ulp->ul_lockfs);

1143         /*
1144          * free current comment
1145          */
1146         if (lfs.lf_comment && lfs.lf_comlen != 0) {
1147                 kmem_free(lfs.lf_comment, lfs.lf_comlen);
1148                 lfs.lf_comment = NULL;
1149                 lfs.lf_comlen = 0;
1150         }

1152         /* do error lock cleanup */
1153         if (errlck == UN_ERRLCK)
1154                 ufsfx_unlockfs(ufsvfsp);

1156         else if (errlck == RE_ERRLCK)
1157                 ufsfx_lockfs(ufsvfsp);

1159         /* don't allow error lock from user to invoke panic */
1160         else if (from_user && errlck == SET_ERRLCK &&
1161             !(ufsvfsp->vfs_fsfx.fx_flags & (UFSMNT_ONERROR_PANIC >> 4)))
1162                 (void) ufs_fault(ufsvfsp->vfs_root,
1163                     ulp->ul_lockfs.lf_comment && ulp->ul_lockfs.lf_comlen > 0 ?
1164                     ulp->ul_lockfs.lf_comment: "user-applied error lock");

1166         atomic_dec_ulong(&ufs_quiesce_pend);
1166         atomic_add_long(&ufs_quiesce_pend, -1);
1167         mutex_exit(&ulp->ul_lock);
1168         vfs_unlock(vfsp);

1170         if (ULOCKFS_IS_HLOCK(&ufsvfsp->vfs_ulockfs))
1171                 poll_events |= POLLERR;

1173         pollwakeup(&ufs_pollhd, poll_events);

1175         /*
1176          * Allow both the delete thread and the reclaim thread to
1177          * continue.
1178          */
1179         ufs_thread_continue(&ufsvfsp->vfs_delete);
1180         ufs_thread_continue(&ufsvfsp->vfs_reclaim);

1182         return (0);

1184 errout:
1185         /*
1186          * Lock failed. Reset the old lock in ufsvfs if not hard locked.
1187          */
1188         if (!LOCKFS_IS_HLOCK(&ulp->ul_lockfs)) {
1189                 bcopy(&lfs, &ulp->ul_lockfs, sizeof (struct lockfs));
1190                 ulp->ul_fs_lock = (1 << lfs.lf_lock);
1191         }
```



```
1193         /*
1194          * Don't call ufs_thaw() when there's a signal during
1195          * ufs quiesce operation as it can lead to deadlock
1196          * with getpage.
1197          */
1198         if (signal == 0)
1199                 (void) ufs_thaw(vfsp, ufsvfsp, ulp);

1201         ULOCKFS_CLR_BUSY(ulp);
1202         LOCKFS_CLR_BUSY(&ulp->ul_lockfs);

1204 errexit:
1205         atomic_dec_ulong(&ufs_quiesce_pend);
1205         atomic_add_long(&ufs_quiesce_pend, -1);
1206         mutex_exit(&ulp->ul_lock);
1207         vfs_unlock(vfsp);

1209         /*
1210          * Allow both the delete thread and the reclaim thread to
1211          * continue.
1212          */
1213         ufs_thread_continue(&ufsvfsp->vfs_delete);
1214         ufs_thread_continue(&ufsvfsp->vfs_reclaim);

1216         return (error);
1217 }
______unchanged_portion_omitted_

1254 /*
1255  * ufs_check_lockfs
1256  *       check whether a ufs_vnops conflicts with the file system lock
1257  */
1258 int
1259 ufs_check_lockfs(struct ufsvfs *ufsvfsp, struct ulockfs *ulp, ulong_t mask)
1260 {
1261         k_sigset_t      smask;
1262         int             sig, slock;

1264         ASSERT(MUTEX_HELD(&ulp->ul_lock));

1266         while (ulp->ul_fs_lock & mask) {
1267                 slock = (int)ULOCKFS_IS_SLOCK(ulp);
1268                 if ((curthread->t_flag & T_DONTPEND) && !slock) {
1269                         curthread->t_flag |= T_WOULDBLOCK;
1270                         return (EAGAIN);
1271                 }
1272                 curthread->t_flag &= ~T_WOULDBLOCK;

1274                 /*
1275                  * In the case of an onerr umount of the fs, threads could
1276                  * have blocked before coming into ufs_check_lockfs and
1277                  * need to check for the special case of ELOCK and
1278                  * vfs_dontblock being set which would indicate that the fs
1279                  * is on its way out and will not return therefore making
1280                  * EIO the appropriate response.
1281                  */
1282                 if (ULOCKFS_IS_HLOCK(ulp) ||
1283                     (ULOCKFS_IS_ELOCK(ulp) && ufsvfsp->vfs_dontblock))
1284                         return (EIO);

1286                 /*
1287                  * wait for lock status to change
1288                  */
1289                 if (slock || ufsvfsp->vfs_nointr) {
1290                         cv_wait(&ulp->ul_cv, &ulp->ul_lock);
```

```
1291                } else {
1292                        sigintr(&smask, 1);
1293                        sig = cv_wait_sig(&ulp->ul_cv, &ulp->ul_lock);
1294                        sigunintr(&smask);
1295                        if ((!sig && (ulp->ul_fs_lock & mask)) ||
1296                            ufsvfsp->vfs_dontblock)
1297                                return (EINTR);
1298                }
1299        }

1301        if (mask & ULOCKFS_FWLOCK) {
1302                atomic_inc_ulong(&ulp->ul_falloc_cnt);
1302                atomic_add_long(&ulp->ul_falloc_cnt, 1);
1303                ULOCKFS_SET_FALLOC(ulp);
1304        } else {
1305                atomic_inc_ulong(&ulp->ul_vnops_cnt);
1305                atomic_add_long(&ulp->ul_vnops_cnt, 1);
1306        }

1308        return (0);
1309 }
______unchanged_portion_omitted_

1323 /*
1324  * ufs_lockfs_begin - start the lockfs locking protocol
1325  */
1326 int
1327 ufs_lockfs_begin(struct ufsvfs *ufsvfsp, struct ulockfs **ulpp, ulong_t mask)
1328 {
1329        int             error;
1330        int             rec_vop;
1331        ushort_t        op_cnt_incremented = 0;
1332        ulong_t         *ctr;
1333        struct ulockfs *ulp;
1334        ulockfs_info_t  *ulockfs_info;
1335        ulockfs_info_t  *ulockfs_info_free;
1336        ulockfs_info_t  *ulockfs_info_temp;

1338        /*
1339         * file system has been forcibly unmounted
1340         */
1341        if (ufsvfsp == NULL)
1342                return (EIO);

1344        *ulpp = ulp = &ufsvfsp->vfs_ulockfs;

1346        /*
1347         * Do lockfs protocol
1348         */
1349        ulockfs_info = (ulockfs_info_t *)tsd_get(ufs_lockfs_key);
1350        IS_REC_VOP(rec_vop, ulockfs_info, ulp, ulockfs_info_free);

1352        /*
1353         * Detect recursive VOP call or handcrafted internal lockfs protocol
1354         * path and bail out in that case.
1355         */
1356        if (rec_vop || ufs_lockfs_is_under_rawlockfs(ulp)) {
1357                *ulpp = NULL;
1358                return (0);
1359        } else {
1360                if (ulockfs_info_free == NULL) {
1361                        if ((ulockfs_info_temp = (ulockfs_info_t *)
1362                            kmem_zalloc(sizeof (ulockfs_info_t),
1363                            KM_NOSLEEP)) == NULL) {
1364                                *ulpp = NULL;
1365                                return (ENOMEM);
```



```
1366                        }
1367                }
1368        }

1370        /*
1371         * First time VOP call
1372         *
1373         * Increment the ctr irrespective of the lockfs state. If the lockfs
1374         * state is not ULOCKFS_ULOCK, we can decrement it later. However,
1375         * before incrementing we need to check if there is a pending quiesce
1376         * request because if we have a continuous stream of ufs_lockfs_begin
1377         * requests pounding on a few cpu's then the ufs_quiesce thread might
1378         * never see the value of zero for ctr - a livelock kind of scenario.
1379         */
1380        ctr = (mask & ULOCKFS_FWLOCK) ?
1381            &ulp->ul_falloc_cnt : &ulp->ul_vnops_cnt;
1382        if (!ULOCKFS_IS_SLOCK(ulp)) {
1383                atomic_inc_ulong(ctr);
1383                atomic_add_long(ctr, 1);
1384                op_cnt_incremented++;
1385        }

1387        /*
1388         * If the lockfs state (indicated by ul_fs_lock) is not just
1389         * ULOCKFS_ULOCK, then we will be routed through ufs_check_lockfs
1390         * where there is a check with an appropriate mask to selectively allow
1391         * operations permitted for that kind of lockfs state.
1392         *
1393         * Even these selective operations should not be allowed to go through
1394         * if a lockfs request is in progress because that could result in inode
1395         * modifications during a quiesce and could hence result in inode
1396         * reconciliation failures. ULOCKFS_SLOCK alone would not be sufficient,
1397         * so make use of ufs_quiesce_pend to disallow vnode operations when a
1398         * quiesce is in progress.
1399         */
1400        if (!ULOCKFS_IS_JUSTULOCK(ulp) || ufs_quiesce_pend) {
1401                if (op_cnt_incremented)
1402                        if (!atomic_dec_ulong_nv(ctr))
1402                        if (!atomic_add_long_nv(ctr, -1))
1403                                cv_broadcast(&ulp->ul_cv);
1404                mutex_enter(&ulp->ul_lock);
1405                error = ufs_check_lockfs(ufsvfsp, ulp, mask);
1406                mutex_exit(&ulp->ul_lock);
1407                if (error) {
1408                        if (ulockfs_info_free == NULL)
1409                                kmem_free(ulockfs_info_temp,
1410                                    sizeof (ulockfs_info_t));
1411                        return (error);
1412                }
1413        } else {
1414                /*
1415                 * This is the common case of file system in a unlocked state.
1416                 *
1417                 * If a file system is unlocked, we would expect the ctr to have
1418                 * been incremented by now. But this will not be true when a
1419                 * quiesce is winding up - SLOCK was set when we checked before
1420                 * incrementing the ctr, but by the time we checked for
1421                 * ULOCKFS_IS_JUSTULOCK, the quiesce thread was gone. It is okay
1422                 * to take ul_lock and go through the slow path in this uncommon
1423                 * case.
1424                 */
1425                if (op_cnt_incremented == 0) {
1426                        mutex_enter(&ulp->ul_lock);
1427                        error = ufs_check_lockfs(ufsvfsp, ulp, mask);
1428                        if (error) {
1429                                mutex_exit(&ulp->ul_lock);
```

```
1430                                if (ulockfs_info_free == NULL)
1431                                        kmem_free(ulockfs_info_temp,
1432                                            sizeof (ulockfs_info_t));
1433                                return (error);
1434                        }
1435                        if (mask & ULOCKFS_FWLOCK)
1436                                ULOCKFS_SET_FALLOC(ulp);
1437                        mutex_exit(&ulp->ul_lock);
1438                } else if (mask & ULOCKFS_FWLOCK) {
1439                        mutex_enter(&ulp->ul_lock);
1440                        ULOCKFS_SET_FALLOC(ulp);
1441                        mutex_exit(&ulp->ul_lock);
1442                }
1443        }

1445        if (ulockfs_info_free != NULL) {
1446                ulockfs_info_free->ulp = ulp;
1447                if (mask & ULOCKFS_FWLOCK)
1448                        ulockfs_info_free->flags |= ULOCK_INFO_FALLOCATE;
1449        } else {
1450                ulockfs_info_temp->ulp = ulp;
1451                ulockfs_info_temp->next = ulockfs_info;
1452                if (mask & ULOCKFS_FWLOCK)
1453                        ulockfs_info_temp->flags |= ULOCK_INFO_FALLOCATE;
1454                ASSERT(ufs_lockfs_key != 0);
1455                (void) tsd_set(ufs_lockfs_key, (void *)ulockfs_info_temp);
1456        }

1458        curthread->t_flag |= T_DONTBLOCK;
1459        return (0);
1460 }
______unchanged_portion_omitted_

1481 /*
1482  * ufs_lockfs_end - terminate the lockfs locking protocol
1483  */
1484 void
1485 ufs_lockfs_end(struct ulockfs *ulp)
1486 {
1487        ulockfs_info_t *info;
1488        ulockfs_info_t *head;

1490        /*
1491         * end-of-VOP protocol
1492         */
1493        if (ulp == NULL)
1494                return;

1496        head = (ulockfs_info_t *)tsd_get(ufs_lockfs_key);
1497        SEARCH_ULOCKFSP(head, ulp, info);

1499        /*
1500         * If we're called from a first level VOP, we have to have a
1501         * valid ulockfs record in the TSD.
1502         */
1503        ASSERT(info != NULL);

1505        /*
1506         * Invalidate the ulockfs record.
1507         */
1508        info->ulp = NULL;

1510        if (ufs_lockfs_top_vop_return(head))
1511                curthread->t_flag &= ~T_DONTBLOCK;

1513        /* fallocate thread */
```



```
1514        if (ULOCKFS_IS_FALLOC(ulp) && info->flags & ULOCK_INFO_FALLOCATE) {
1515                /* Clear the thread's fallocate state */
1516                info->flags &= ~ULOCK_INFO_FALLOCATE;
1517                if (!atomic_dec_ulong_nv(&ulp->ul_falloc_cnt)) {
1517                if (!atomic_add_long_nv(&ulp->ul_falloc_cnt, -1)) {
1518                        mutex_enter(&ulp->ul_lock);
1519                        ULOCKFS_CLR_FALLOC(ulp);
1520                        cv_broadcast(&ulp->ul_cv);
1521                        mutex_exit(&ulp->ul_lock);
1522                }
1523        } else  { /* normal thread */
1524                if (!atomic_dec_ulong_nv(&ulp->ul_vnops_cnt))
1524                if (!atomic_add_long_nv(&ulp->ul_vnops_cnt, -1))
1525                        cv_broadcast(&ulp->ul_cv);
1526        }
1527 }

1529 /*
1530  * ufs_lockfs_trybegin - try to start the lockfs locking protocol without
1531  * blocking.
1532  */
1533 int
1534 ufs_lockfs_trybegin(struct ufsvfs *ufsvfsp, struct ulockfs **ulpp, ulong_t mask)
1535 {
1536        int             error = 0;
1537        int             rec_vop;
1538        ushort_t        op_cnt_incremented = 0;
1539        ulong_t         *ctr;
1540        struct ulockfs *ulp;
1541        ulockfs_info_t  *ulockfs_info;
1542        ulockfs_info_t  *ulockfs_info_free;
1543        ulockfs_info_t  *ulockfs_info_temp;

1545        /*
1546         * file system has been forcibly unmounted
1547         */
1548        if (ufsvfsp == NULL)
1549                return (EIO);

1551        *ulpp = ulp = &ufsvfsp->vfs_ulockfs;

1553        /*
1554         * Do lockfs protocol
1555         */
1556        ulockfs_info = (ulockfs_info_t *)tsd_get(ufs_lockfs_key);
1557        IS_REC_VOP(rec_vop, ulockfs_info, ulp, ulockfs_info_free);

1559        /*
1560         * Detect recursive VOP call or handcrafted internal lockfs protocol
1561         * path and bail out in that case.
1562         */
1563        if (rec_vop || ufs_lockfs_is_under_rawlockfs(ulp)) {
1564                *ulpp = NULL;
1565                return (0);
1566        } else {
1567                if (ulockfs_info_free == NULL) {
1568                        if ((ulockfs_info_temp = (ulockfs_info_t *)
1569                            kmem_zalloc(sizeof (ulockfs_info_t),
1570                            KM_NOSLEEP)) == NULL) {
1571                                *ulpp = NULL;
1572                                return (ENOMEM);
1573                        }
1574                }
1575        }

1577        /*
```

```
1578          * First time VOP call
1579          *
1580          * Increment the ctr irrespective of the lockfs state. If the lockfs
1581          * state is not ULOCKFS_ULOCK, we can decrement it later. However,
1582          * before incrementing we need to check if there is a pending quiesce
1583          * request because if we have a continuous stream of ufs_lockfs_begin
1584          * requests pounding on a few cpu's then the ufs_quiesce thread might
1585          * never see the value of zero for ctr - a livelock kind of scenario.
1586          */
1587         ctr = (mask & ULOCKFS_FWLOCK) ?
1588             &ulp->ul_falloc_cnt : &ulp->ul_vnops_cnt;
1589         if (!ULOCKFS_IS_SLOCK(ulp)) {
1590                 atomic_inc_ulong(ctr);
1590                 atomic_add_long(ctr, 1);
1591                 op_cnt_incremented++;
1592         }
1593
1594         if (!ULOCKFS_IS_JUSTULOCK(ulp) || ufs_quiesce_pend) {
1595                 /*
1596                  * Non-blocking version of ufs_check_lockfs() code.
1597                  *
1598                  * If the file system is not hard locked or error locked
1599                  * and if ulp->ul_fs_lock allows this operation, increment
1600                  * the appropriate counter and proceed (For eg., In case the
1601                  * file system is delete locked, a mmap can still go through).
1602                  */
1603                 if (op_cnt_incremented)
1604                         if (!atomic_dec_ulong_nv(ctr))
1604                         if (!atomic_add_long_nv(ctr, -1))
1605                                 cv_broadcast(&ulp->ul_cv);
1606                 mutex_enter(&ulp->ul_lock);
1607                 if (ULOCKFS_IS_HLOCK(ulp) ||
1608                     (ULOCKFS_IS_ELOCK(ulp) && ufsvfsp->vfs_dontblock))
1609                         error = EIO;
1610                 else if (ulp->ul_fs_lock & mask)
1611                         error = EAGAIN;
1612
1613                 if (error) {
1614                         mutex_exit(&ulp->ul_lock);
1615                         if (ulockfs_info_free == NULL)
1616                                 kmem_free(ulockfs_info_temp,
1617                                     sizeof (ulockfs_info_t));
1618                         return (error);
1619                 }
1620                 atomic_inc_ulong(ctr);
1620                 atomic_add_long(ctr, 1);
1621                 if (mask & ULOCKFS_FWLOCK)
1622                         ULOCKFS_SET_FALLOC(ulp);
1623                 mutex_exit(&ulp->ul_lock);
1624         } else {
1625                 /*
1626                  * This is the common case of file system in a unlocked state.
1627                  *
1628                  * If a file system is unlocked, we would expect the ctr to have
1629                  * been incremented by now. But this will not be true when a
1630                  * quiesce is winding up - SLOCK was set when we checked before
1631                  * incrementing the ctr, but by the time we checked for
1632                  * ULOCKFS_IS_JUSTULOCK, the quiesce thread was gone. Take
1633                  * ul_lock and go through the non-blocking version of
1634                  * ufs_check_lockfs() code.
1635                  */
1636                 if (op_cnt_incremented == 0) {
1637                         mutex_enter(&ulp->ul_lock);
1638                         if (ULOCKFS_IS_HLOCK(ulp) ||
1639                             (ULOCKFS_IS_ELOCK(ulp) && ufsvfsp->vfs_dontblock))
1640                                 error = EIO;
```



```
1641                         else if (ulp->ul_fs_lock & mask)
1642                                 error = EAGAIN;
1643
1644                         if (error) {
1645                                 mutex_exit(&ulp->ul_lock);
1646                                 if (ulockfs_info_free == NULL)
1647                                         kmem_free(ulockfs_info_temp,
1648                                             sizeof (ulockfs_info_t));
1649                                 return (error);
1650                         }
1651                         atomic_inc_ulong(ctr);
1651                         atomic_add_long(ctr, 1);
1652                         if (mask & ULOCKFS_FWLOCK)
1653                                 ULOCKFS_SET_FALLOC(ulp);
1654                         mutex_exit(&ulp->ul_lock);
1655                 } else if (mask & ULOCKFS_FWLOCK) {
1656                         mutex_enter(&ulp->ul_lock);
1657                         ULOCKFS_SET_FALLOC(ulp);
1658                         mutex_exit(&ulp->ul_lock);
1659                 }
1660         }
1661
1662         if (ulockfs_info_free != NULL) {
1663                 ulockfs_info_free->ulp = ulp;
1664                 if (mask & ULOCKFS_FWLOCK)
1665                         ulockfs_info_free->flags |= ULOCK_INFO_FALLOCATE;
1666         } else {
1667                 ulockfs_info_temp->ulp = ulp;
1668                 ulockfs_info_temp->next = ulockfs_info;
1669                 if (mask & ULOCKFS_FWLOCK)
1670                         ulockfs_info_temp->flags |= ULOCK_INFO_FALLOCATE;
1671                 ASSERT(ufs_lockfs_key != 0);
1672                 (void) tsd_set(ufs_lockfs_key, (void *)ulockfs_info_temp);
1673         }
1674
1675         curthread->t_flag |= T_DONTBLOCK;
1676         return (0);
1677 }
1678
1679 /*
1680  * specialized version of ufs_lockfs_begin() called by ufs_getpage().
1681  */
1682 int
1683 ufs_lockfs_begin_getpage(
1684         struct ufsvfs   *ufsvfsp,
1685         struct ulockfs  **ulpp,
1686         struct seg      *seg,
1687         int             read_access,
1688         uint_t          *protp)
1689 {
1690         ulong_t                 mask;
1691         int                     error;
1692         int                     rec_vop;
1693         struct ulockfs          *ulp;
1694         ulockfs_info_t          *ulockfs_info;
1695         ulockfs_info_t          *ulockfs_info_free;
1696         ulockfs_info_t          *ulockfs_info_temp;
1697
1698         /*
1699          * file system has been forcibly unmounted
1700          */
1701         if (ufsvfsp == NULL)
1702                 return (EIO);
1703
1704         *ulpp = ulp = &ufsvfsp->vfs_ulockfs;
```

```
1706         /*
1707          * Do lockfs protocol
1708          */
1709         ulockfs_info = (ulockfs_info_t *)tsd_get(ufs_lockfs_key);
1710         IS_REC_VOP(rec_vop, ulockfs_info, ulp, ulockfs_info_free);

1712         /*
1713          * Detect recursive VOP call or handcrafted internal lockfs protocol
1714          * path and bail out in that case.
1715          */
1716         if (rec_vop || ufs_lockfs_is_under_rawlockfs(ulp)) {
1717                 *ulpp = NULL;
1718                 return (0);
1719         } else {
1720                 if (ulockfs_info_free == NULL) {
1721                         if ((ulockfs_info_temp = (ulockfs_info_t *)
1722                             kmem_zalloc(sizeof (ulockfs_info_t),
1723                             KM_NOSLEEP)) == NULL) {
1724                                 *ulpp = NULL;
1725                                 return (ENOMEM);
1726                         }
1727                 }
1728         }

1730         /*
1731          * First time VOP call
1732          */
1733         atomic_inc_ulong(&ulp->ul_vnops_cnt);
1733         atomic_add_long(&ulp->ul_vnops_cnt, 1);
1734         if (!ULOCKFS_IS_JUSTULOCK(ulp) || ufs_quiesce_pend) {
1735                 if (!atomic_dec_ulong_nv(&ulp->ul_vnops_cnt))
1735                 if (!atomic_add_long_nv(&ulp->ul_vnops_cnt, -1))
1736                         cv_broadcast(&ulp->ul_cv);
1737                 mutex_enter(&ulp->ul_lock);
1738                 if (seg->s_ops == &segvn_ops &&
1739                     ((struct segvn_data *)seg->s_data)->type != MAP_SHARED) {
1740                         mask = (ulong_t)ULOCKFS_GETREAD_MASK;
1741                 } else if (protp && read_access) {
1742                         /*
1743                          * Restrict the mapping to readonly.
1744                          * Writes to this mapping will cause
1745                          * another fault which will then
1746                          * be suspended if fs is write locked
1747                          */
1748                         *protp &= ~PROT_WRITE;
1749                         mask = (ulong_t)ULOCKFS_GETREAD_MASK;
1750                 } else
1751                         mask = (ulong_t)ULOCKFS_GETWRITE_MASK;

1753                 /*
1754                  * will sleep if this fs is locked against this VOP
1755                  */
1756                 error = ufs_check_lockfs(ufsvfsp, ulp, mask);
1757                 mutex_exit(&ulp->ul_lock);
1758                 if (error) {
1759                         if (ulockfs_info_free == NULL)
1760                                 kmem_free(ulockfs_info_temp,
1761                                     sizeof (ulockfs_info_t));
1762                         return (error);
1763                 }
1764         }

1766         if (ulockfs_info_free != NULL) {
1767                 ulockfs_info_free->ulp = ulp;
1768         } else {
1769                 ulockfs_info_temp->ulp = ulp;
```



```
1770                 ulockfs_info_temp->next = ulockfs_info;
1771                 ASSERT(ufs_lockfs_key != 0);
1772                 (void) tsd_set(ufs_lockfs_key, (void *)ulockfs_info_temp);
1773         }

1775         curthread->t_flag |= T_DONTBLOCK;
1776         return (0);
1777 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   59843 Mon Jul 28 07:44:21 2014
new/usr/src/uts/common/fs/ufs/ufs_vfsops.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 575 static int
 576 remountfs(struct vfs *vfsp, dev_t dev, void *raw_argsp, int args_len)
 577 {
 578         struct ufsvfs *ufsvfsp = (struct ufsvfs *)vfsp->vfs_data;
 579         struct ulockfs *ulp = &ufsvfsp->vfs_ulockfs;
 580         struct buf *bp = ufsvfsp->vfs_bufp;
 581         struct fs *fsp = (struct fs *)bp->b_un.b_addr;
 582         struct fs *fspt;
 583         struct buf *tpt = 0;
 584         int error = 0;
 585         int flags = 0;

 587         if (args_len == sizeof (struct ufs_args) && raw_argsp)
 588                 flags = ((struct ufs_args *)raw_argsp)->flags;

 590         /* cannot remount to RDONLY */
 591         if (vfsp->vfs_flag & VFS_RDONLY)
 592                 return (ENOTSUP);

 594         /* whoops, wrong dev */
 595         if (vfsp->vfs_dev != dev)
 596                 return (EINVAL);

 598         /*
 599          * synchronize w/ufs ioctls
 600          */
 601         mutex_enter(&ulp->ul_lock);
 602         atomic_inc_ulong(&ufs_quiesce_pend);
 602         atomic_add_long(&ufs_quiesce_pend, 1);

 604         /*
 605          * reset options
 606          */
 607         ufsvfsp->vfs_nointr  = flags & UFSMNT_NOINTR;
 608         ufsvfsp->vfs_syncdir = flags & UFSMNT_SYNCDIR;
 609         ufsvfsp->vfs_nosetsec = flags & UFSMNT_NOSETSEC;
 610         ufsvfsp->vfs_noatime = flags & UFSMNT_NOATIME;
 611         if ((flags & UFSMNT_NODFRATIME) || ufsvfsp->vfs_noatime)
 612                 ufsvfsp->vfs_dfritime &= ~UFS_DFRATIME;
 613         else    /* dfratime, default behavior */
 614                 ufsvfsp->vfs_dfritime |= UFS_DFRATIME;
 615         if (flags & UFSMNT_FORCEDIRECTIO)
 616                 ufsvfsp->vfs_forcedirectio = 1;
 617         else    /* default is no direct I/O */
 618                 ufsvfsp->vfs_forcedirectio = 0;
 619         ufsvfsp->vfs_iotstamp = ddi_get_lbolt();

 621         /*
 622          * set largefiles flag in ufsvfs equal to the
 623          * value passed in by the mount command. If
 624          * it is "nolargefiles", and the flag is set
 625          * in the superblock, the mount fails.
 626          */
 627         if (!(flags & UFSMNT_LARGEFILES)) {  /* "nolargefiles" */
 628                 if (fsp->fs_flags & FSLARGEFILES) {
 629                         error = EFBIG;
 630                         goto remounterr;
 631                 }
 632                 ufsvfsp->vfs_lfflags &= ~UFS_LARGEFILES;
```



```
 633         } else  /* "largefiles" */
 634                 ufsvfsp->vfs_lfflags |= UFS_LARGEFILES;
 635         /*
 636          * read/write to read/write; all done
 637          */
 638         if (fsp->fs_ronly == 0)
 639                 goto remounterr;

 641         /*
 642          * fix-on-panic assumes RO->RW remount implies system-critical fs
 643          * if it is shortly after boot; so, don't attempt to lock and fix
 644          * (unless the user explicitly asked for another action on error)
 645          * XXX UFSMNT_ONERROR_RDONLY rather than UFSMNT_ONERROR_PANIC
 646          */
 647 #define BOOT_TIME_LIMIT (180*hz)
 648         if (!(flags & UFSMNT_ONERROR_FLGMASK) &&
 649             ddi_get_lbolt() < BOOT_TIME_LIMIT) {
 650                 cmn_err(CE_WARN, "%s is required to be mounted onerror=%s",
 651                     ufsvfsp->vfs_fs->fs_fsmnt, UFSMNT_ONERROR_PANIC_STR);
 652                 flags |= UFSMNT_ONERROR_PANIC;
 653         }

 655         if ((error = ufsfx_mount(ufsvfsp, flags)) != 0)
 656                 goto remounterr;

 658         /*
 659          * quiesce the file system
 660          */
 661         error = ufs_quiesce(ulp);
 662         if (error)
 663                 goto remounterr;

 665         tpt = UFS_BREAD(ufsvfsp, ufsvfsp->vfs_dev, SBLOCK, SBSIZE);
 666         if (tpt->b_flags & B_ERROR) {
 667                 error = EIO;
 668                 goto remounterr;
 669         }
 670         fspt = (struct fs *)tpt->b_un.b_addr;
 671         if (((fspt->fs_magic != FS_MAGIC) &&
 672             (fspt->fs_magic != MTB_UFS_MAGIC)) ||
 673             (fspt->fs_magic == FS_MAGIC &&
 674             (fspt->fs_version != UFS_EFISTYLE4NONEFI_VERSION_2 &&
 675             fspt->fs_version != UFS_VERSION_MIN)) ||
 676             (fspt->fs_magic == MTB_UFS_MAGIC &&
 677             (fspt->fs_version > MTB_UFS_VERSION_1 ||
 678             fspt->fs_version < MTB_UFS_VERSION_MIN)) ||
 679             fspt->fs_bsize > MAXBSIZE || fspt->fs_frag > MAXFRAG ||
 680             fspt->fs_bsize < sizeof (struct fs) || fspt->fs_bsize < PAGESIZE) {
 681                 tpt->b_flags |= B_STALE | B_AGE;
 682                 error = EINVAL;
 683                 goto remounterr;
 684         }

 686         if (ufsvfsp->vfs_log && (ufsvfsp->vfs_log->un_flags & LDL_NOROLL)) {
 687                 ufsvfsp->vfs_log->un_flags &= ~LDL_NOROLL;
 688                 logmap_start_roll(ufsvfsp->vfs_log);
 689         }

 691         if (TRANS_ISERROR(ufsvfsp))
 692                 goto remounterr;
 693         TRANS_DOMATAMAP(ufsvfsp);

 695         if ((fspt->fs_state + fspt->fs_time == FSOKAY) &&
 696             fspt->fs_clean == FSLOG && !TRANS_ISTRANS(ufsvfsp)) {
 697                 ufsvfsp->vfs_log = NULL;
 698                 ufsvfsp->vfs_domatamap = 0;
```

```
699                 error = ENOSPC;
700                 goto remounterr;
701         }

703         if (fspt->fs_state + fspt->fs_time == FSOKAY &&
704             (fspt->fs_clean == FSCLEAN ||
705             fspt->fs_clean == FSSTABLE ||
706             fspt->fs_clean == FSLOG)) {

708                 /*
709                  * Ensure that ufs_getsummaryinfo doesn't reconstruct
710                  * the summary info.
711                  */
712                 error = ufs_getsummaryinfo(vfsp->vfs_dev, ufsvfsp, fspt);
713                 if (error)
714                         goto remounterr;

716                 /* preserve mount name */
717                 (void) strncpy(fspt->fs_fsmnt, fsp->fs_fsmnt, MAXMNTLEN);
718                 /* free the old cg space */
719                 kmem_free(fsp->fs_u.fs_csp, fsp->fs_cssize);
720                 /* switch in the new superblock */
721                 fspt->fs_rolled = FS_NEED_ROLL;
722                 bcopy(tpt->b_un.b_addr, bp->b_un.b_addr, fspt->fs_sbsize);

724                 fsp->fs_clean = FSSTABLE;
725         } /* superblock updated in memory */
726         tpt->b_flags |= B_STALE | B_AGE;
727         brelse(tpt);
728         tpt = 0;

730         if (fsp->fs_clean != FSSTABLE) {
731                 error = ENOSPC;
732                 goto remounterr;
733         }


736         if (TRANS_ISTRANS(ufsvfsp)) {
737                 fsp->fs_clean = FSLOG;
738                 ufsvfsp->vfs_dio = 0;
739         } else
740                 if (ufsvfsp->vfs_dio)
741                         fsp->fs_clean = FSSUSPEND;

743         TRANS_MATA_MOUNT(ufsvfsp);

745         fsp->fs_fmod = 0;
746         fsp->fs_ronly = 0;

748         atomic_dec_ulong(&ufs_quiesce_pend);
748         atomic_add_long(&ufs_quiesce_pend, -1);
749         cv_broadcast(&ulp->ul_cv);
750         mutex_exit(&ulp->ul_lock);

752         if (TRANS_ISTRANS(ufsvfsp)) {

754                 /*
755                  * start the delete thread
756                  */
757                 ufs_thread_start(&ufsvfsp->vfs_delete, ufs_thread_delete, vfsp);

759                 /*
760                  * start the reclaim thread
761                  */
762                 if (fsp->fs_reclaim & (FS_RECLAIM|FS_RECLAIMING)) {
763                         fsp->fs_reclaim &= ~FS_RECLAIM;
```



```
764                         fsp->fs_reclaim |=  FS_RECLAIMING;
765                         ufs_thread_start(&ufsvfsp->vfs_reclaim,
766                             ufs_thread_reclaim, vfsp);
767                 }
768         }

770         TRANS_SBWRITE(ufsvfsp, TOP_MOUNT);

772         return (0);

774 remounterr:
775         if (tpt)
776                 brelse(tpt);
777         atomic_dec_ulong(&ufs_quiesce_pend);
777         atomic_add_long(&ufs_quiesce_pend, -1);
778         cv_broadcast(&ulp->ul_cv);
779         mutex_exit(&ulp->ul_lock);
780         return (error);
781 }
_____unchanged_portion_omitted_

1370 /*
1371  * vfs operations
1372  */
1373 static int
1374 ufs_unmount(struct vfs *vfsp, int fflag, struct cred *cr)
1375 {
1376         dev_t           dev             = vfsp->vfs_dev;
1377         struct ufsvfs   *ufsvfsp        = (struct ufsvfs *)vfsp->vfs_data;
1378         struct fs       *fs             = ufsvfsp->vfs_fs;
1379         struct ulockfs  *ulp            = &ufsvfsp->vfs_ulockfs;
1380         struct vnode    *bvp, *vp;
1381         struct buf      *bp;
1382         struct inode    *ip, *inext, *rip;
1383         union ihead     *ih;
1384         int             error, flag, i;
1385         struct lockfs   lockfs;
1386         int             poll_events = POLLPRI;
1387         extern struct pollhead ufs_pollhd;
1388         refstr_t        *mountpoint;

1390         ASSERT(vfs_lock_held(vfsp));

1392         if (secpolicy_fs_unmount(cr, vfsp) != 0)
1393                 return (EPERM);
1394         /*
1395          * Forced unmount is now supported through the
1396          * lockfs protocol.
1397          */
1398         if (fflag & MS_FORCE) {
1399                 /*
1400                  * Mark the filesystem as being unmounted now in
1401                  * case of a forcible umount before we take any
1402                  * locks inside UFS to prevent racing with a VFS_VGET()
1403                  * request. Throw these VFS_VGET() requests away for
1404                  * the duration of the forcible umount so they won't
1405                  * use stale or even freed data later on when we're done.
1406                  * It may happen that the VFS has had a additional hold
1407                  * placed on it by someone other than UFS and thus will
1408                  * not get freed immediately once we're done with the
1409                  * umount by dounmount() - use VFS_UNMOUNTED to inform
1410                  * users of this still-alive VFS that its corresponding
1411                  * filesystem being gone so they can detect that and error
1412                  * out.
1413                  */
1414                 vfsp->vfs_flag |= VFS_UNMOUNTED;
```

```
1416                ufs_thread_suspend(&ufsvfsp->vfs_delete);
1417                mutex_enter(&ulp->ul_lock);
1418                /*
1419                 * If file system is already hard locked,
1420                 * unmount the file system, otherwise
1421                 * hard lock it before unmounting.
1422                 */
1423                if (!ULOCKFS_IS_HLOCK(ulp)) {
1424                        atomic_inc_ulong(&ufs_quiesce_pend);
1424                        atomic_add_long(&ufs_quiesce_pend, 1);
1425                        lockfs.lf_lock = LOCKFS_HLOCK;
1426                        lockfs.lf_flags = 0;
1427                        lockfs.lf_key = ulp->ul_lockfs.lf_key + 1;
1428                        lockfs.lf_comlen = 0;
1429                        lockfs.lf_comment = NULL;
1430                        ufs_freeze(ulp, &lockfs);
1431                        ULOCKFS_SET_BUSY(ulp);
1432                        LOCKFS_SET_BUSY(&ulp->ul_lockfs);
1433                        (void) ufs_quiesce(ulp);
1434                        (void) ufs_flush(vfsp);
1435                        (void) ufs_thaw(vfsp, ufsvfsp, ulp);
1436                        atomic_dec_ulong(&ufs_quiesce_pend);
1436                        atomic_add_long(&ufs_quiesce_pend, -1);
1437                        ULOCKFS_CLR_BUSY(ulp);
1438                        LOCKFS_CLR_BUSY(&ulp->ul_lockfs);
1439                        poll_events |= POLLERR;
1440                        pollwakeup(&ufs_pollhd, poll_events);
1441                }
1442                ufs_thread_continue(&ufsvfsp->vfs_delete);
1443                mutex_exit(&ulp->ul_lock);
1444        }

1446        /* let all types of writes go through */
1447        ufsvfsp->vfs_iotstamp = ddi_get_lbolt();

1449        /* coordinate with global hlock thread */
1450        if (TRANS_ISTRANS(ufsvfsp) && (ufsvfsp->vfs_validfs == UT_HLOCKING)) {
1451                /*
1452                 * last possibility for a forced umount to fail hence clear
1453                 * VFS_UNMOUNTED if appropriate.
1454                 */
1455                if (fflag & MS_FORCE)
1456                        vfsp->vfs_flag &= ~VFS_UNMOUNTED;
1457                return (EAGAIN);
1458        }

1460        ufsvfsp->vfs_validfs = UT_UNMOUNTED;

1462        /* kill the reclaim thread */
1463        ufs_thread_exit(&ufsvfsp->vfs_reclaim);

1465        /* suspend the delete thread */
1466        ufs_thread_suspend(&ufsvfsp->vfs_delete);

1468        /*
1469         * drain the delete and idle queues
1470         */
1471        ufs_delete_drain(vfsp, -1, 1);
1472        ufs_idle_drain(vfsp);

1474        /*
1475         * use the lockfs protocol to prevent new ops from starting
1476         * a forcible umount can not fail beyond this point as
1477         * we hard-locked the filesystem and drained all current consumers
1478         * before.
```



```
1479         */
1480        mutex_enter(&ulp->ul_lock);

1482        /*
1483         * if the file system is busy; return EBUSY
1484         */
1485        if (ulp->ul_vnops_cnt || ulp->ul_falloc_cnt || ULOCKFS_IS_SLOCK(ulp)) {
1486                error = EBUSY;
1487                goto out;
1488        }

1490        /*
1491         * if this is not a forced unmount (!hard/error locked), then
1492         * get rid of every inode except the root and quota inodes
1493         * also, commit any outstanding transactions
1494         */
1495        if (!ULOCKFS_IS_HLOCK(ulp) && !ULOCKFS_IS_ELOCK(ulp))
1496                if (error = ufs_flush(vfsp))
1497                        goto out;

1499        /*
1500         * ignore inodes in the cache if fs is hard locked or error locked
1501         */
1502        rip = VTOI(ufsvfsp->vfs_root);
1503        if (!ULOCKFS_IS_HLOCK(ulp) && !ULOCKFS_IS_ELOCK(ulp)) {
1504                /*
1505                 * Otherwise, only the quota and root inodes are in the cache.
1506                 *
1507                 * Avoid racing with ufs_update() and ufs_sync().
1508                 */
1509                mutex_enter(&ufs_scan_lock);

1511                for (i = 0, ih = ihead; i < inohsz; i++, ih++) {
1512                        mutex_enter(&ih_lock[i]);
1513                        for (ip = ih->ih_chain[0];
1514                            ip != (struct inode *)ih;
1515                            ip = ip->i_forw) {
1516                                if (ip->i_ufsvfs != ufsvfsp)
1517                                        continue;
1518                                if (ip == ufsvfsp->vfs_qinod)
1519                                        continue;
1520                                if (ip == rip && ITOV(ip)->v_count == 1)
1521                                        continue;
1522                                mutex_exit(&ih_lock[i]);
1523                                mutex_exit(&ufs_scan_lock);
1524                                error = EBUSY;
1525                                goto out;
1526                        }
1527                        mutex_exit(&ih_lock[i]);
1528                }
1529                mutex_exit(&ufs_scan_lock);
1530        }

1532        /*
1533         * if a snapshot exists and this is a forced unmount, then delete
1534         * the snapshot.  Otherwise return EBUSY.  This will insure the
1535         * snapshot always belongs to a valid file system.
1536         */
1537        if (ufsvfsp->vfs_snapshot) {
1538                if (ULOCKFS_IS_HLOCK(ulp) || ULOCKFS_IS_ELOCK(ulp)) {
1539                        (void) fssnap_delete(&ufsvfsp->vfs_snapshot);
1540                } else {
1541                        error = EBUSY;
1542                        goto out;
1543                }
1544        }
```

```
1546         /*
1547          * Close the quota file and invalidate anything left in the quota
1548          * cache for this file system.  Pass kcred to allow all quota
1549          * manipulations.
1550          */
1551         (void) closedq(ufsvfsp, kcred);
1552         invalidatedq(ufsvfsp);
1553         /*
1554          * drain the delete and idle queues
1555          */
1556         ufs_delete_drain(vfsp, -1, 0);
1557         ufs_idle_drain(vfsp);

1559         /*
1560          * discard the inodes for this fs (including root, shadow, and quota)
1561          */
1562         for (i = 0, ih = ihead; i < inohsz; i++, ih++) {
1563                 mutex_enter(&ih_lock[i]);
1564                 for (inext = 0, ip = ih->ih_chain[0];
1565                     ip != (struct inode *)ih;
1566                     ip = inext) {
1567                         inext = ip->i_forw;
1568                         if (ip->i_ufsvfs != ufsvfsp)
1569                                 continue;

1571                         /*
1572                          * We've found the inode in the cache and as we
1573                          * hold the hash mutex the inode can not
1574                          * disappear from underneath us.
1575                          * We also know it must have at least a vnode
1576                          * reference count of 1.
1577                          * We perform an additional VN_HOLD so the VN_RELE
1578                          * in case we take the inode off the idle queue
1579                          * can not be the last one.
1580                          * It is safe to grab the writer contents lock here
1581                          * to prevent a race with ufs_iinactive() putting
1582                          * inodes into the idle queue while we operate on
1583                          * this inode.
1584                          */
1585                         rw_enter(&ip->i_contents, RW_WRITER);

1587                         vp = ITOV(ip);
1588                         VN_HOLD(vp)
1589                         remque(ip);
1590                         if (ufs_rmidle(ip))
1591                                 VN_RELE(vp);
1592                         ufs_si_del(ip);
1593                         /*
1594                          * rip->i_ufsvfsp is needed by bflush()
1595                          */
1596                         if (ip != rip)
1597                                 ip->i_ufsvfs = NULL;
1598                         /*
1599                          * Set vnode's vfsops to dummy ops, which return
1600                          * EIO. This is needed to forced unmounts to work
1601                          * with lofs/nfs properly.
1602                          */
1603                         if (ULOCKFS_IS_HLOCK(ulp) || ULOCKFS_IS_ELOCK(ulp))
1604                                 vp->v_vfsp = &EIO_vfs;
1605                         else
1606                                 vp->v_vfsp = NULL;
1607                         vp->v_type = VBAD;

1609                         rw_exit(&ip->i_contents);
```



```
1611                         VN_RELE(vp);
1612                 }
1613                 mutex_exit(&ih_lock[i]);
1614         }
1615         ufs_si_cache_flush(dev);

1617         /*
1618          * kill the delete thread and drain the idle queue
1619          */
1620         ufs_thread_exit(&ufsvfsp->vfs_delete);
1621         ufs_idle_drain(vfsp);

1623         bp = ufsvfsp->vfs_bufp;
1624         bvp = ufsvfsp->vfs_devvp;
1625         flag = !fs->fs_ronly;
1626         if (flag) {
1627                 bflush(dev);
1628                 if (fs->fs_clean != FSBAD) {
1629                         if (fs->fs_clean == FSSTABLE)
1630                                 fs->fs_clean = FSCLEAN;
1631                         fs->fs_reclaim &= ~FS_RECLAIM;
1632                 }
1633                 if (TRANS_ISTRANS(ufsvfsp) &&
1634                     !TRANS_ISERROR(ufsvfsp) &&
1635                     !ULOCKFS_IS_HLOCK(ulp) &&
1636                     (fs->fs_rolled == FS_NEED_ROLL)) {
1637                         /*
1638                          * ufs_flush() above has flushed the last Moby.
1639                          * This is needed to ensure the following superblock
1640                          * update really is the last metadata update
1641                          */
1642                         error = ufs_putsummaryinfo(dev, ufsvfsp, fs);
1643                         if (error == 0) {
1644                                 fs->fs_rolled = FS_ALL_ROLLED;
1645                         }
1646                 }
1647                 TRANS_SBUPDATE(ufsvfsp, vfsp, TOP_SBUPDATE_UNMOUNT);
1648                 /*
1649                  * push this last transaction
1650                  */
1651                 curthread->t_flag |= T_DONTBLOCK;
1652                 TRANS_BEGIN_SYNC(ufsvfsp, TOP_COMMIT_UNMOUNT, TOP_COMMIT_SIZE,
1653                     error);
1654                 if (!error)
1655                         TRANS_END_SYNC(ufsvfsp, error, TOP_COMMIT_UNMOUNT,
1656                             TOP_COMMIT_SIZE);
1657                 curthread->t_flag &= ~T_DONTBLOCK;
1658         }

1660         TRANS_MATA_UMOUNT(ufsvfsp);
1661         lufs_unsnarf(ufsvfsp);          /* Release the in-memory structs */
1662         ufsfx_unmount(ufsvfsp);         /* fix-on-panic bookkeeping */
1663         kmem_free(fs->fs_u.fs_csp, fs->fs_cssize);

1665         bp->b_flags |= B_STALE|B_AGE;
1666         ufsvfsp->vfs_bufp = NULL;       /* don't point at freed buf */
1667         brelse(bp);                     /* free the superblock buf */

1669         (void) VOP_PUTPAGE(common_specvp(bvp), (offset_t)0, (size_t)0,
1670             B_INVAL, cr, NULL);
1671         (void) VOP_CLOSE(bvp, flag, 1, (offset_t)0, cr, NULL);
1672         bflush(dev);
1673         (void) bfinval(dev, 1);
1674         VN_RELE(bvp);

1676         /*
```

```
1677          * It is now safe to NULL out the ufsvfs pointer and discard
1678          * the root inode.
1679          */
1680         rip->i_ufsvfs = NULL;
1681         VN_RELE(ITOV(rip));

1683         /* free up lockfs comment structure, if any */
1684         if (ulp->ul_lockfs.lf_comlen && ulp->ul_lockfs.lf_comment)
1685                 kmem_free(ulp->ul_lockfs.lf_comment, ulp->ul_lockfs.lf_comlen);

1687         /*
1688          * Remove from instance list.
1689          */
1690         ufs_vfs_remove(ufsvfsp);

1692         /*
1693          * For a forcible unmount, threads may be asleep in
1694          * ufs_lockfs_begin/ufs_check_lockfs.  These threads will need
1695          * the ufsvfs structure so we don't free it, yet.  ufs_update
1696          * will free it up after awhile.
1697          */
1698         if (ULOCKFS_IS_HLOCK(ulp) || ULOCKFS_IS_ELOCK(ulp)) {
1699                 extern kmutex_t         ufsvfs_mutex;
1700                 extern struct ufsvfs    *ufsvfslist;

1702                 mutex_enter(&ufsvfs_mutex);
1703                 ufsvfsp->vfs_dontblock = 1;
1704                 ufsvfsp->vfs_next = ufsvfslist;
1705                 ufsvfslist = ufsvfsp;
1706                 mutex_exit(&ufsvfs_mutex);
1707                 /* wakeup any suspended threads */
1708                 cv_broadcast(&ulp->ul_cv);
1709                 mutex_exit(&ulp->ul_lock);
1710         } else {
1711                 mutex_destroy(&ufsvfsp->vfs_lock);
1712                 kmem_free(ufsvfsp, sizeof (struct ufsvfs));
1713         }

1715         /*
1716          * Now mark the filesystem as unmounted since we're done with it.
1717          */
1718         vfsp->vfs_flag |= VFS_UNMOUNTED;

1720         return (0);
1721 out:
1722         /* open the fs to new ops */
1723         cv_broadcast(&ulp->ul_cv);
1724         mutex_exit(&ulp->ul_lock);

1726         if (TRANS_ISTRANS(ufsvfsp)) {
1727                 /* allow the delete thread to continue */
1728                 ufs_thread_continue(&ufsvfsp->vfs_delete);
1729                 /* restart the reclaim thread */
1730                 ufs_thread_start(&ufsvfsp->vfs_reclaim, ufs_thread_reclaim,
1731                     vfsp);
1732                 /* coordinate with global hlock thread */
1733                 ufsvfsp->vfs_validfs = UT_MOUNTED;
1734                 /* check for trans errors during umount */
1735                 ufs_trans_onerror();

1737                 /*
1738                  * if we have a separate /usr it will never unmount
1739                  * when halting. In order to not re-read all the
1740                  * cylinder group summary info on mounting after
1741                  * reboot the logging of summary info is re-enabled
1742                  * and the super block written out.
```



```
1743                  */
1744                 mountpoint = vfs_getmntpoint(vfsp);
1745                 if ((fs->fs_si == FS_SI_OK) &&
1746                     (strcmp("/usr", refstr_value(mountpoint)) == 0)) {
1747                         ufsvfsp->vfs_nolog_si = 0;
1748                         UFS_BWRITE2(NULL, ufsvfsp->vfs_bufp);
1749                 }
1750                 refstr_rele(mountpoint);
1751         }

1753         return (error);
1754 }
______unchanged_portion_omitted_

2104 #ifdef __sparc

2106 /*
2107  * Mounting a mirrored SVM volume is only supported on ufs,
2108  * this is special-case boot code to support that configuration.
2109  * At this point, we have booted and mounted root on a
2110  * single component of the mirror.  Complete the boot
2111  * by configuring SVM and converting the root to the
2112  * dev_t of the mirrored root device.  This dev_t conversion
2113  * only works because the underlying device doesn't change.
2114  */
2115 int
2116 ufs_remountroot(struct vfs *vfsp)
2117 {
2118         struct ufsvfs *ufsvfsp;
2119         struct ulockfs *ulp;
2120         dev_t new_rootdev;
2121         dev_t old_rootdev;
2122         struct vnode *old_rootvp;
2123         struct vnode *new_rootvp;
2124         int error, sberror = 0;
2125         struct inode    *ip;
2126         union ihead     *ih;
2127         struct buf      *bp;
2128         int i;

2130         old_rootdev = rootdev;
2131         old_rootvp = rootvp;

2133         new_rootdev = getrootdev();
2134         if (new_rootdev == (dev_t)NODEV) {
2135                 return (ENODEV);
2136         }

2138         new_rootvp = makespecvp(new_rootdev, VBLK);

2140         error = VOP_OPEN(&new_rootvp,
2141             (vfsp->vfs_flag & VFS_RDONLY) ? FREAD : FREAD|FWRITE, CRED(), NULL);
2142         if (error) {
2143                 cmn_err(CE_CONT,
2144                     "Cannot open mirrored root device, error %d\n", error);
2145                 return (error);
2146         }

2148         if (vfs_lock(vfsp) != 0) {
2149                 return (EBUSY);
2150         }

2152         ufsvfsp = (struct ufsvfs *)vfsp->vfs_data;
2153         ulp = &ufsvfsp->vfs_ulockfs;

2155         mutex_enter(&ulp->ul_lock);
```

```
2156         atomic_inc_ulong(&ufs_quiesce_pend);
2156         atomic_add_long(&ufs_quiesce_pend, 1);

2158         (void) ufs_quiesce(ulp);
2159         (void) ufs_flush(vfsp);

2161         /*
2162          * Convert root vfs to new dev_t, including vfs hash
2163          * table and fs id.
2164          */
2165         vfs_root_redev(vfsp, new_rootdev, ufsfstype);

2167         ufsvfsp->vfs_devvp = new_rootvp;
2168         ufsvfsp->vfs_dev = new_rootdev;

2170         bp = ufsvfsp->vfs_bufp;
2171         bp->b_edev = new_rootdev;
2172         bp->b_dev = cmpdev(new_rootdev);

2174         /*
2175          * The buffer for the root inode does not contain a valid b_vp
2176          */
2177         (void) bfinval(new_rootdev, 0);

2179         /*
2180          * Here we hand-craft inodes with old root device
2181          * references to refer to the new device instead.
2182          */
2183         mutex_enter(&ufs_scan_lock);

2185         for (i = 0, ih = ihead; i < inohsz; i++, ih++) {
2186                 mutex_enter(&ih_lock[i]);
2187                 for (ip = ih->ih_chain[0];
2188                     ip != (struct inode *)ih;
2189                     ip = ip->i_forw) {
2190                         if (ip->i_ufsvfs != ufsvfsp)
2191                                 continue;
2192                         if (ip == ufsvfsp->vfs_qinod)
2193                                 continue;
2194                         if (ip->i_dev == old_rootdev) {
2195                                 ip->i_dev = new_rootdev;
2196                         }

2198                         if (ip->i_devvp == old_rootvp) {
2199                                 ip->i_devvp = new_rootvp;
2200                         }
2201                 }
2202                 mutex_exit(&ih_lock[i]);
2203         }

2205         mutex_exit(&ufs_scan_lock);

2207         /*
2208          * Make Sure logging structures are using the new device
2209          * if logging is enabled.  Also start any logging thread that
2210          * needs to write to the device and couldn't earlier.
2211          */
2212         if (ufsvfsp->vfs_log) {
2213                 buf_t           *bp, *tbp;
2214                 ml_unit_t       *ul = ufsvfsp->vfs_log;
2215                 struct fs       *fsp = ufsvfsp->vfs_fs;

2217                 /*
2218                  * Update the main logging structure.
2219                  */
2220                 ul->un_dev = new_rootdev;
```



```
2222                 /*
2223                  * Get a new bp for the on disk structures.
2224                  */
2225                 bp = ul->un_bp;
2226                 tbp = ngeteblk(dbtob(LS_SECTORS));
2227                 tbp->b_edev = new_rootdev;
2228                 tbp->b_dev = cmpdev(new_rootdev);
2229                 tbp->b_blkno = bp->b_blkno;
2230                 bcopy(bp->b_un.b_addr, tbp->b_un.b_addr, DEV_BSIZE);
2231                 bcopy(bp->b_un.b_addr, tbp->b_un.b_addr + DEV_BSIZE, DEV_BSIZE);
2232                 bp->b_flags |= (B_STALE | B_AGE);
2233                 brelse(bp);
2234                 ul->un_bp = tbp;

2236                 /*
2237                  * Allocate new circular buffers.
2238                  */
2239                 alloc_rdbuf(&ul->un_rdbuf, MAPBLOCKSIZE, MAPBLOCKSIZE);
2240                 alloc_wrbuf(&ul->un_wrbuf, ldl_bufsize(ul));

2242                 /*
2243                  * Clear the noroll bit which indicates that logging
2244                  * can't roll the log yet and start the logmap roll thread
2245                  * unless the filesystem is still read-only in which case
2246                  * remountfs() will do it when going to read-write.
2247                  */
2248                 ASSERT(ul->un_flags & LDL_NOROLL);

2250                 if (!fsp->fs_ronly) {
2251                         ul->un_flags &= ~LDL_NOROLL;
2252                         logmap_start_roll(ul);
2253                 }

2255                 /*
2256                  * Start the reclaim thread if needed.
2257                  */
2258                 if (!fsp->fs_ronly && (fsp->fs_reclaim &
2259                     (FS_RECLAIM|FS_RECLAIMING))) {
2260                         fsp->fs_reclaim &= ~FS_RECLAIM;
2261                         fsp->fs_reclaim |= FS_RECLAIMING;
2262                         ufs_thread_start(&ufsvfsp->vfs_reclaim,
2263                             ufs_thread_reclaim, vfsp);
2264                         TRANS_SBWRITE(ufsvfsp, TOP_SBUPDATE_UPDATE);
2265                         if (sberror = geterror(ufsvfsp->vfs_bufp)) {
2266                                 refstr_t        *mntpt;
2267                                 mntpt = vfs_getmntpoint(vfsp);
2268                                 cmn_err(CE_WARN,
2269                                     "Remountroot failed to update Reclaim"
2270                                     "state for filesystem %s "
2271                                     "Error writing SuperBlock %d",
2272                                     refstr_value(mntpt), error);
2273                                 refstr_rele(mntpt);
2274                         }
2275                 }
2276         }

2278         rootdev = new_rootdev;
2279         rootvp = new_rootvp;

2281         atomic_dec_ulong(&ufs_quiesce_pend);
2281         atomic_add_long(&ufs_quiesce_pend, -1);
2282         cv_broadcast(&ulp->ul_cv);
2283         mutex_exit(&ulp->ul_lock);

2285         vfs_unlock(vfsp);
```

```
2287         error = VOP_CLOSE(old_rootvp, FREAD, 1, (offset_t)0, CRED(), NULL);
2288         if (error) {
2289                 cmn_err(CE_CONT,
2290                     "close of root device component failed, error %d\n",
2291                     error);
2292         }
2293         VN_RELE(old_rootvp);

2295         return (sberror ? sberror : error);
2296 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  170819 Mon Jul 28 07:44:21 2014
new/usr/src/uts/common/fs/ufs/ufs_vnops.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

5921 int ufs_pageio_writes, ufs_pageio_reads;

5923 /*ARGSUSED*/
5924 static int
5925 ufs_pageio(struct vnode *vp, page_t *pp, u_offset_t io_off, size_t io_len,
5926         int flags, struct cred *cr, caller_context_t *ct)
5927 {
5928         struct inode *ip = VTOI(vp);
5929         struct ufsvfs *ufsvfsp;
5930         page_t *npp = NULL, *opp = NULL, *cpp = pp;
5931         struct buf *bp;
5932         daddr_t bn;
5933         size_t done_len = 0, cur_len = 0;
5934         int err = 0;
5935         int contig = 0;
5936         int dolock;
5937         int vmpss = 0;
5938         struct ulockfs *ulp;

5940         if ((flags & B_READ) && pp != NULL && pp->p_vnode == vp &&
5941             vp->v_mpssdata != NULL) {
5942                 vmpss = 1;
5943         }

5945         dolock = (rw_owner(&ip->i_contents) != curthread);
5946         /*
5947          * We need a better check.  Ideally, we would use another
5948          * vnodeops so that hlocked and forcibly unmounted file
5949          * systems would return EIO where appropriate and w/o the
5950          * need for these checks.
5951          */
5952         if ((ufsvfsp = ip->i_ufsvfs) == NULL)
5953                 return (EIO);

5955         /*
5956          * For vmpss (pp can be NULL) case respect the quiesce protocol.
5957          * ul_lock must be taken before locking pages so we can't use it here
5958          * if pp is non NULL because segvn already locked pages
5959          * SE_EXCL. Instead we rely on the fact that a forced umount or
5960          * applying a filesystem lock via ufs_fiolfs() will block in the
5961          * implicit call to ufs_flush() until we unlock the pages after the
5962          * return to segvn. Other ufs_quiesce() callers keep ufs_quiesce_pend
5963          * above 0 until they are done. We have to be careful not to increment
5964          * ul_vnops_cnt here after forceful unmount hlocks the file system.
5965          *
5966          * If pp is NULL use ul_lock to make sure we don't increment
5967          * ul_vnops_cnt after forceful unmount hlocks the file system.
5968          */
5969         if (vmpss || pp == NULL) {
5970                 ulp = &ufsvfsp->vfs_ulockfs;
5971                 if (pp == NULL)
5972                         mutex_enter(&ulp->ul_lock);
5973                 if (ulp->ul_fs_lock & ULOCKFS_GETREAD_MASK) {
5974                         if (pp == NULL) {
5975                                 mutex_exit(&ulp->ul_lock);
5976                         }
5977                         return (vmpss ? EIO : EINVAL);
5978                 }
5979                 atomic_inc_ulong(&ulp->ul_vnops_cnt);
```



```
5979                 atomic_add_long(&ulp->ul_vnops_cnt, 1);
5980                 if (pp == NULL)
5981                         mutex_exit(&ulp->ul_lock);
5982                 if (ufs_quiesce_pend) {
5983                         if (!atomic_dec_ulong_nv(&ulp->ul_vnops_cnt))
5983                         if (!atomic_add_long_nv(&ulp->ul_vnops_cnt, -1))
5984                                 cv_broadcast(&ulp->ul_cv);
5985                         return (vmpss ? EIO : EINVAL);
5986                 }
5987         }

5989         if (dolock) {
5990                 /*
5991                  * segvn may call VOP_PAGEIO() instead of VOP_GETPAGE() to
5992                  * handle a fault against a segment that maps vnode pages with
5993                  * large mappings.  Segvn creates pages and holds them locked
5994                  * SE_EXCL during VOP_PAGEIO() call. In this case we have to
5995                  * use rw_tryenter() to avoid a potential deadlock since in
5996                  * lock order i_contents needs to be taken first.
5997                  * Segvn will retry via VOP_GETPAGE() if VOP_PAGEIO() fails.
5998                  */
5999                 if (!vmpss) {
6000                         rw_enter(&ip->i_contents, RW_READER);
6001                 } else if (!rw_tryenter(&ip->i_contents, RW_READER)) {
6002                         if (!atomic_dec_ulong_nv(&ulp->ul_vnops_cnt))
6002                         if (!atomic_add_long_nv(&ulp->ul_vnops_cnt, -1))
6003                                 cv_broadcast(&ulp->ul_cv);
6004                         return (EDEADLK);
6005                 }
6006         }

6008         /*
6009          * Return an error to segvn because the pagefault request is beyond
6010          * PAGESIZE rounded EOF.
6011          */
6012         if (vmpss && btopr(io_off + io_len) > btopr(ip->i_size)) {
6013                 if (dolock)
6014                         rw_exit(&ip->i_contents);
6015                 if (!atomic_dec_ulong_nv(&ulp->ul_vnops_cnt))
6015                 if (!atomic_add_long_nv(&ulp->ul_vnops_cnt, -1))
6016                         cv_broadcast(&ulp->ul_cv);
6017                 return (EFAULT);
6018         }

6020         if (pp == NULL) {
6021                 if (bmap_has_holes(ip)) {
6022                         err = ENOSYS;
6023                 } else {
6024                         err = EINVAL;
6025                 }
6026                 if (dolock)
6027                         rw_exit(&ip->i_contents);
6028                 if (!atomic_dec_ulong_nv(&ulp->ul_vnops_cnt))
6028                 if (!atomic_add_long_nv(&ulp->ul_vnops_cnt, -1))
6029                         cv_broadcast(&ulp->ul_cv);
6030                 return (err);
6031         }

6033         /*
6034          * Break the io request into chunks, one for each contiguous
6035          * stretch of disk blocks in the target file.
6036          */
6037         while (done_len < io_len) {
6038                 ASSERT(cpp);
6039                 contig = 0;
6040                 if (err = bmap_read(ip, (u_offset_t)(io_off + done_len),
```

```
6041                    &bn, &contig))
6042                        break;
6043
6044                if (bn == UFS_HOLE) {   /* No holey swapfiles */
6045                        if (vmpss) {
6046                                err = EFAULT;
6047                                break;
6048                        }
6049                        err = ufs_fault(ITOV(ip), "ufs_pageio: bn == UFS_HOLE");
6050                        break;
6051                }
6052
6053                cur_len = MIN(io_len - done_len, contig);
6054                /*
6055                 * Zero out a page beyond EOF, when the last block of
6056                 * a file is a UFS fragment so that ufs_pageio() can be used
6057                 * instead of ufs_getpage() to handle faults against
6058                 * segvn segments that use large pages.
6059                 */
6060                page_list_break(&cpp, &npp, btopr(cur_len));
6061                if ((flags & B_READ) && (cur_len & PAGEOFFSET)) {
6062                        size_t xlen = cur_len & PAGEOFFSET;
6063                        pagezero(cpp->p_prev, xlen, PAGESIZE - xlen);
6064                }
6065
6066                bp = pageio_setup(cpp, cur_len, ip->i_devvp, flags);
6067                ASSERT(bp != NULL);
6068
6069                bp->b_edev = ip->i_dev;
6070                bp->b_dev = cmpdev(ip->i_dev);
6071                bp->b_blkno = bn;
6072                bp->b_un.b_addr = (caddr_t)0;
6073                bp->b_file = ip->i_vnode;
6074
6075                ufsvfsp->vfs_iotstamp = ddi_get_lbolt();
6076                ub.ub_pageios.value.ul++;
6077                if (ufsvfsp->vfs_snapshot)
6078                        fssnap_strategy(&(ufsvfsp->vfs_snapshot), bp);
6079                else
6080                        (void) bdev_strategy(bp);
6081
6082                if (flags & B_READ)
6083                        ufs_pageio_reads++;
6084                else
6085                        ufs_pageio_writes++;
6086                if (flags & B_READ)
6087                        lwp_stat_update(LWP_STAT_INBLK, 1);
6088                else
6089                        lwp_stat_update(LWP_STAT_OUBLK, 1);
6090                /*
6091                 * If the request is not B_ASYNC, wait for i/o to complete
6092                 * and re-assemble the page list to return to the caller.
6093                 * If it is B_ASYNC we leave the page list in pieces and
6094                 * cleanup() will dispose of them.
6095                 */
6096                if ((flags & B_ASYNC) == 0) {
6097                        err = biowait(bp);
6098                        pageio_done(bp);
6099                        if (err)
6100                                break;
6101                        page_list_concat(&opp, &cpp);
6102                }
6103                cpp = npp;
6104                npp = NULL;
6105                if (flags & B_READ)
6106                        cur_len = P2ROUNDUP_TYPED(cur_len, PAGESIZE, size_t);
```



```
6107                    done_len += cur_len;
6108            }
6109            ASSERT(err || (cpp == NULL && npp == NULL && done_len == io_len));
6110            if (err) {
6111                    if (flags & B_ASYNC) {
6112                            /* Cleanup unprocessed parts of list */
6113                            page_list_concat(&cpp, &npp);
6114                            if (flags & B_READ)
6115                                    pvn_read_done(cpp, B_ERROR);
6116                            else
6117                                    pvn_write_done(cpp, B_ERROR);
6118                    } else {
6119                            /* Re-assemble list and let caller clean up */
6120                            page_list_concat(&opp, &cpp);
6121                            page_list_concat(&opp, &npp);
6122                    }
6123            }
6124
6125            if (vmpss && !(ip->i_flag & IACC) && !ULOCKFS_IS_NOIACC(ulp) &&
6126                ufsvfsp->vfs_fs->fs_ronly == 0 && !ufsvfsp->vfs_noatime) {
6127                    mutex_enter(&ip->i_tlock);
6128                    ip->i_flag |= IACC;
6129                    ITIMES_NOLOCK(ip);
6130                    mutex_exit(&ip->i_tlock);
6131            }
6132
6133            if (dolock)
6134                    rw_exit(&ip->i_contents);
6135            if (vmpss && !atomic_dec_ulong_nv(&ulp->ul_vnops_cnt))
6135            if (vmpss && !atomic_add_long_nv(&ulp->ul_vnops_cnt, -1))
6136                    cv_broadcast(&ulp->ul_cv);
6137            return (err);
6138 }
```

*_____unchanged_portion_omitted_*

****
118239 Mon Jul 28 07:44:22 2014
new/usr/src/uts/common/fs/vfs.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
4328 /*
4329  * Increments the vfs reference count by one atomically.
4330  */
4331 void
4332 vfs_hold(vfs_t *vfsp)
4333 {
4334         atomic_inc_32(&vfsp->vfs_count);
4334         atomic_add_32(&vfsp->vfs_count, 1);
4335         ASSERT(vfsp->vfs_count != 0);
4336 }

4338 /*
4339  * Decrements the vfs reference count by one atomically. When
4340  * vfs reference count becomes zero, it calls the file system
4341  * specific vfs_freevfs() to free up the resources.
4342  */
4343 void
4344 vfs_rele(vfs_t *vfsp)
4345 {
4346         ASSERT(vfsp->vfs_count != 0);
4347         if (atomic_dec_32_nv(&vfsp->vfs_count) == 0) {
4347         if (atomic_add_32_nv(&vfsp->vfs_count, -1) == 0) {
4348                 VFS_FREEVFS(vfsp);
4349                 lofi_remove(vfsp);
4350                 if (vfsp->vfs_zone)
4351                         zone_rele_ref(&vfsp->vfs_implp->vi_zone_ref,
4352                             ZONE_REF_VFS);
4353                 vfs_freemnttab(vfsp);
4354                 vfs_free(vfsp);
4355         }
4356 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  105296 Mon Jul 28 07:44:22 2014
new/usr/src/uts/common/fs/vnode.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1228 /*
1229  * The following two accessor functions are for the NFSv4 server.  Since there
1230  * is no VOP_OPEN_UP/DOWNGRADE we need a way for the NFS server to keep the
1231  * vnode open counts correct when a client "upgrades" an open or does an
1232  * open_downgrade.  In NFS, an upgrade or downgrade can not only change the
1233  * open mode (add or subtract read or write), but also change the share/deny
1234  * modes.  However, share reservations are not integrated with OPEN, yet, so
1235  * we need to handle each separately.  These functions are cleaner than having
1236  * the NFS server manipulate the counts directly, however, nobody else should
1237  * use these functions.
1238  */
1239 void
1240 vn_open_upgrade(
1241         vnode_t *vp,
1242         int filemode)
1243 {
1244         ASSERT(vp->v_type == VREG);

1246         if (filemode & FREAD)
1247                 atomic_inc_32(&vp->v_rdcnt);
1247                 atomic_add_32(&(vp->v_rdcnt), 1);
1248         if (filemode & FWRITE)
1249                 atomic_inc_32(&vp->v_wrcnt);
1249                 atomic_add_32(&(vp->v_wrcnt), 1);

1251 }

1253 void
1254 vn_open_downgrade(
1255         vnode_t *vp,
1256         int filemode)
1257 {
1258         ASSERT(vp->v_type == VREG);

1260         if (filemode & FREAD) {
1261                 ASSERT(vp->v_rdcnt > 0);
1262                 atomic_dec_32(&vp->v_rdcnt);
1262                 atomic_add_32(&(vp->v_rdcnt), -1);
1263         }
1264         if (filemode & FWRITE) {
1265                 ASSERT(vp->v_wrcnt > 0);
1266                 atomic_dec_32(&vp->v_wrcnt);
1266                 atomic_add_32(&(vp->v_wrcnt), -1);
1267         }

1269 }
______unchanged_portion_omitted_

2910 /*
2911  * fs_new_caller_id() needs to return a unique ID on a given local system.
2912  * The IDs do not need to survive across reboots.  These are primarily
2913  * used so that (FEM) monitors can detect particular callers (such as
2914  * the NFS server) to a given vnode/vfs operation.
2915  */
2916 u_longlong_t
2917 fs_new_caller_id()
2918 {
2919         static uint64_t next_caller_id = 0LL; /* First call returns 1 */
```



```
2921         return ((u_longlong_t)atomic_inc_64_nv(&next_caller_id));
2921         return ((u_longlong_t)atomic_add_64_nv(&next_caller_id, 1));
2922 }
______unchanged_portion_omitted_

3122 /* VOP_XXX() macros call the corresponding fop_xxx() function */

3124 int
3125 fop_open(
3126         vnode_t **vpp,
3127         int mode,
3128         cred_t *cr,
3129         caller_context_t *ct)
3130 {
3131         int ret;
3132         vnode_t *vp = *vpp;

3134         VN_HOLD(vp);
3135         /*
3136          * Adding to the vnode counts before calling open
3137          * avoids the need for a mutex. It circumvents a race
3138          * condition where a query made on the vnode counts results in a
3139          * false negative. The inquirer goes away believing the file is
3140          * not open when there is an open on the file already under way.
3141          *
3142          * The counts are meant to prevent NFS from granting a delegation
3143          * when it would be dangerous to do so.
3144          *
3145          * The vnode counts are only kept on regular files
3146          */
3147         if ((*vpp)->v_type == VREG) {
3148                 if (mode & FREAD)
3149                         atomic_inc_32(&(*vpp)->v_rdcnt);
3149                         atomic_add_32(&((*vpp)->v_rdcnt), 1);
3150                 if (mode & FWRITE)
3151                         atomic_inc_32(&(*vpp)->v_wrcnt);
3151                         atomic_add_32(&((*vpp)->v_wrcnt), 1);
3152         }

3154         VOPXID_MAP_CR(vp, cr);

3156         ret = (*(*(vpp))->v_op->vop_open)(vpp, mode, cr, ct);

3158         if (ret) {
3159                 /*
3160                  * Use the saved vp just in case the vnode ptr got trashed
3161                  * by the error.
3162                  */
3163                 VOPSTATS_UPDATE(vp, open);
3164                 if ((vp->v_type == VREG) && (mode & FREAD))
3165                         atomic_dec_32(&vp->v_rdcnt);
3165                         atomic_add_32(&(vp->v_rdcnt), -1);
3166                 if ((vp->v_type == VREG) && (mode & FWRITE))
3167                         atomic_dec_32(&vp->v_wrcnt);
3167                         atomic_add_32(&(vp->v_wrcnt), -1);
3168         } else {
3169                 /*
3170                  * Some filesystems will return a different vnode,
3171                  * but the same path was still used to open it.
3172                  * So if we do change the vnode and need to
3173                  * copy over the path, do so here, rather than special
3174                  * casing each filesystem. Adjust the vnode counts to
3175                  * reflect the vnode switch.
3176                  */
3177                 VOPSTATS_UPDATE(*vpp, open);
3178                 if (*vpp != vp && *vpp != NULL) {
```

```
3179                         vn_copypath(vp, *vpp);
3180                         if (((*vpp)->v_type == VREG) && (mode & FREAD))
3181                                 atomic_inc_32(&(*vpp)->v_rdcnt);
3181                                 atomic_add_32(&((*vpp)->v_rdcnt), 1);
3182                         if ((vp->v_type == VREG) && (mode & FREAD))
3183                                 atomic_dec_32(&vp->v_rdcnt);
3183                                 atomic_add_32(&(vp->v_rdcnt), -1);
3184                         if (((*vpp)->v_type == VREG) && (mode & FWRITE))
3185                                 atomic_inc_32(&(*vpp)->v_wrcnt);
3185                                 atomic_add_32(&((*vpp)->v_wrcnt), 1);
3186                         if ((vp->v_type == VREG) && (mode & FWRITE))
3187                                 atomic_dec_32(&vp->v_wrcnt);
3187                                 atomic_add_32(&(vp->v_wrcnt), -1);
3188                 }
3189         }
3190         VN_RELE(vp);
3191         return (ret);
3192 }

3194 int
3195 fop_close(
3196         vnode_t *vp,
3197         int flag,
3198         int count,
3199         offset_t offset,
3200         cred_t *cr,
3201         caller_context_t *ct)
3202 {
3203         int err;

3205         VOPXID_MAP_CR(vp, cr);

3207         err = (*(vp)->v_op->vop_close)(vp, flag, count, offset, cr, ct);
3208         VOPSTATS_UPDATE(vp, close);
3209         /*
3210          * Check passed in count to handle possible dups. Vnode counts are only
3211          * kept on regular files
3212          */
3213         if ((vp->v_type == VREG) && (count == 1))  {
3214                 if (flag & FREAD) {
3215                         ASSERT(vp->v_rdcnt > 0);
3216                         atomic_dec_32(&vp->v_rdcnt);
3216                         atomic_add_32(&(vp->v_rdcnt), -1);
3217                 }
3218                 if (flag & FWRITE) {
3219                         ASSERT(vp->v_wrcnt > 0);
3220                         atomic_dec_32(&vp->v_wrcnt);
3220                         atomic_add_32(&(vp->v_wrcnt), -1);
3221                 }
3222         }
3223         return (err);
3224 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   77744 Mon Jul 28 07:44:22 2014
new/usr/src/uts/common/fs/zfs/dbuf.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 144 /*
 145  * Insert an entry into the hash table.  If there is already an element
 146  * equal to elem in the hash table, then the already existing element
 147  * will be returned and the new element will not be inserted.
 148  * Otherwise returns NULL.
 149  */
 150 static dmu_buf_impl_t *
 151 dbuf_hash_insert(dmu_buf_impl_t *db)
 152 {
 153         dbuf_hash_table_t *h = &dbuf_hash_table;
 154         objset_t *os = db->db_objset;
 155         uint64_t obj = db->db.db_object;
 156         int level = db->db_level;
 157         uint64_t blkid = db->db_blkid;
 158         uint64_t hv = DBUF_HASH(os, obj, level, blkid);
 159         uint64_t idx = hv & h->hash_table_mask;
 160         dmu_buf_impl_t *dbf;

 162         mutex_enter(DBUF_HASH_MUTEX(h, idx));
 163         for (dbf = h->hash_table[idx]; dbf != NULL; dbf = dbf->db_hash_next) {
 164                 if (DBUF_EQUAL(dbf, os, obj, level, blkid)) {
 165                         mutex_enter(&dbf->db_mtx);
 166                         if (dbf->db_state != DB_EVICTING) {
 167                                 mutex_exit(DBUF_HASH_MUTEX(h, idx));
 168                                 return (dbf);
 169                         }
 170                         mutex_exit(&dbf->db_mtx);
 171                 }
 172         }

 174         mutex_enter(&db->db_mtx);
 175         db->db_hash_next = h->hash_table[idx];
 176         h->hash_table[idx] = db;
 177         mutex_exit(DBUF_HASH_MUTEX(h, idx));
 178         atomic_inc_64(&dbuf_hash_count);
 178         atomic_add_64(&dbuf_hash_count, 1);

 180         return (NULL);
 181 }

 183 /*
 184  * Remove an entry from the hash table.  This operation will
 185  * fail if there are any existing holds on the db.
 186  */
 187 static void
 188 dbuf_hash_remove(dmu_buf_impl_t *db)
 189 {
 190         dbuf_hash_table_t *h = &dbuf_hash_table;
 191         uint64_t hv = DBUF_HASH(db->db_objset, db->db.db_object,
 192             db->db_level, db->db_blkid);
 193         uint64_t idx = hv & h->hash_table_mask;
 194         dmu_buf_impl_t *dbf, **dbp;

 196         /*
 197          * We musn't hold db_mtx to maintin lock ordering:
 198          * DBUF_HASH_MUTEX > db_mtx.
 199          */
 200         ASSERT(refcount_is_zero(&db->db_holds));
 201         ASSERT(db->db_state == DB_EVICTING);
```



```
 202         ASSERT(!MUTEX_HELD(&db->db_mtx));

 204         mutex_enter(DBUF_HASH_MUTEX(h, idx));
 205         dbp = &h->hash_table[idx];
 206         while ((dbf = *dbp) != db) {
 207                 dbp = &dbf->db_hash_next;
 208                 ASSERT(dbf != NULL);
 209         }
 210         *dbp = db->db_hash_next;
 211         db->db_hash_next = NULL;
 212         mutex_exit(DBUF_HASH_MUTEX(h, idx));
 213         atomic_dec_64(&dbuf_hash_count);
 213         atomic_add_64(&dbuf_hash_count, -1);
 214 }
______unchanged_portion_omitted_
```

```
**********************************************************
  178476 Mon Jul 28 07:44:23 2014
new/usr/src/uts/common/fs/zfs/spa.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1814 static void
1815 spa_load_verify_done(zio_t *zio)
1816 {
1817         blkptr_t *bp = zio->io_bp;
1818         spa_load_error_t *sle = zio->io_private;
1819         dmu_object_type_t type = BP_GET_TYPE(bp);
1820         int error = zio->io_error;
1821         spa_t *spa = zio->io_spa;

1823         if (error) {
1824                 if ((BP_GET_LEVEL(bp) != 0 || DMU_OT_IS_METADATA(type)) &&
1825                     type != DMU_OT_INTENT_LOG)
1826                         atomic_inc_64(&sle->sle_meta_count);
1826                         atomic_add_64(&sle->sle_meta_count, 1);
1827                 else
1828                         atomic_inc_64(&sle->sle_data_count);
1828                         atomic_add_64(&sle->sle_data_count, 1);
1829         }
1830         zio_data_buf_free(zio->io_data, zio->io_size);

1832         mutex_enter(&spa->spa_scrub_lock);
1833         spa->spa_scrub_inflight--;
1834         cv_broadcast(&spa->spa_scrub_io_cv);
1835         mutex_exit(&spa->spa_scrub_lock);
1836 }
______unchanged_portion_omitted_
```

****
3429 Mon Jul 28 07:44:23 2014
new/usr/src/uts/common/fs/zfs/sys/refcount.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
 84 #define refcount_create(rc) ((rc)->rc_count = 0)
 85 #define refcount_create_untracked(rc) ((rc)->rc_count = 0)
 86 #define refcount_destroy(rc) ((rc)->rc_count = 0)
 87 #define refcount_destroy_many(rc, number) ((rc)->rc_count = 0)
 88 #define refcount_is_zero(rc) ((rc)->rc_count == 0)
 89 #define refcount_count(rc) ((rc)->rc_count)
 90 #define refcount_add(rc, holder) atomic_inc_64_nv(&(rc)->rc_count)
 91 #define refcount_remove(rc, holder) atomic_dec_64_nv(&(rc)->rc_count)
 90 #define refcount_add(rc, holder) atomic_add_64_nv(&(rc)->rc_count, 1)
 91 #define refcount_remove(rc, holder) atomic_add_64_nv(&(rc)->rc_count, -1)
 92 #define refcount_add_many(rc, number, holder) \
 93         atomic_add_64_nv(&(rc)->rc_count, number)
 94 #define refcount_remove_many(rc, number, holder) \
 95         atomic_add_64_nv(&(rc)->rc_count, -number)
 96 #define refcount_transfer(dst, src) { \
 97         uint64_t __tmp = (src)->rc_count; \
 98         atomic_add_64(&(src)->rc_count, -__tmp); \
 99         atomic_add_64(&(dst)->rc_count, __tmp); \
100 }
```

_____***unchanged_portion_omitted_***

****
11381 Mon Jul 28 07:44:23 2014
new/usr/src/uts/common/fs/zfs/vdev_cache.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
105 #define VDCSTAT_BUMP(stat)      atomic_inc_64(&vdc_stats.stat.value.ui64);
105 #define VDCSTAT_BUMP(stat)      atomic_add_64(&vdc_stats.stat.value.ui64, 1);

107 static int
108 vdev_cache_offset_compare(const void *a1, const void *a2)
109 {
110         const vdev_cache_entry_t *ve1 = a1;
111         const vdev_cache_entry_t *ve2 = a2;

113         if (ve1->ve_offset < ve2->ve_offset)
114                 return (-1);
115         if (ve1->ve_offset > ve2->ve_offset)
116                 return (1);
117         return (0);
118 }
```

_____***unchanged_portion_omitted_***

****
37509 Mon Jul 28 07:44:23 2014
new/usr/src/uts/common/fs/zfs/vdev_label.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
 969 /*
 970  * On success, increment root zio's count of good writes.
 971  * We only get credit for writes to known-visible vdevs; see spa_vdev_add().
 972  */
 973 static void
 974 vdev_uberblock_sync_done(zio_t *zio)
 975 {
 976         uint64_t *good_writes = zio->io_private;

 978         if (zio->io_error == 0 && zio->io_vd->vdev_top->vdev_ms_array != 0)
 979                 atomic_inc_64(good_writes);
 979                 atomic_add_64(good_writes, 1);
 980 }
```

_____***unchanged_portion_omitted_***

```
1045 /*
1046  * On success, increment the count of good writes for our top-level vdev.
1047  */
1048 static void
1049 vdev_label_sync_done(zio_t *zio)
1050 {
1051         uint64_t *good_writes = zio->io_private;

1053         if (zio->io_error == 0)
1054                 atomic_inc_64(good_writes);
1054                 atomic_add_64(good_writes, 1);
1055 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   60827 Mon Jul 28 07:44:24 2014
new/usr/src/uts/common/fs/zfs/zfs_vfsops.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1154 static int
1155 zfs_domount(vfs_t *vfsp, char *osname)
1156 {
1157         dev_t mount_dev;
1158         uint64_t recordsize, fsid_guid;
1159         int error = 0;
1160         zfsvfs_t *zfsvfs;

1162         ASSERT(vfsp);
1163         ASSERT(osname);

1165         error = zfsvfs_create(osname, &zfsvfs);
1166         if (error)
1167                 return (error);
1168         zfsvfs->z_vfs = vfsp;

1170         /* Initialize the generic filesystem structure. */
1171         vfsp->vfs_bcount = 0;
1172         vfsp->vfs_data = NULL;

1174         if (zfs_create_unique_device(&mount_dev) == -1) {
1175                 error = SET_ERROR(ENODEV);
1176                 goto out;
1177         }
1178         ASSERT(vfs_devismounted(mount_dev) == 0);

1180         if (error = dsl_prop_get_integer(osname, "recordsize", &recordsize,
1181             NULL))
1182                 goto out;

1184         vfsp->vfs_dev = mount_dev;
1185         vfsp->vfs_fstype = zfsfstype;
1186         vfsp->vfs_bsize = recordsize;
1187         vfsp->vfs_flag |= VFS_NOTRUNC;
1188         vfsp->vfs_data = zfsvfs;

1190         /*
1191          * The fsid is 64 bits, composed of an 8-bit fs type, which
1192          * separates our fsid from any other filesystem types, and a
1193          * 56-bit objset unique ID.  The objset unique ID is unique to
1194          * all objsets open on this system, provided by unique_create().
1195          * The 8-bit fs type must be put in the low bits of fsid[1]
1196          * because that's where other Solaris filesystems put it.
1197          */
1198         fsid_guid = dmu_objset_fsid_guid(zfsvfs->z_os);
1199         ASSERT((fsid_guid & ~((1ULL<<56)-1)) == 0);
1200         vfsp->vfs_fsid.val[0] = fsid_guid;
1201         vfsp->vfs_fsid.val[1] = ((fsid_guid>>32) << 8) |
1202             zfsfstype & 0xFF;

1204         /*
1205          * Set features for file system.
1206          */
1207         zfs_set_fuid_feature(zfsvfs);
1208         if (zfsvfs->z_case == ZFS_CASE_INSENSITIVE) {
1209                 vfs_set_feature(vfsp, VFSFT_DIRENTFLAGS);
1210                 vfs_set_feature(vfsp, VFSFT_CASEINSENSITIVE);
1211                 vfs_set_feature(vfsp, VFSFT_NOCASESENSITIVE);
1212         } else if (zfsvfs->z_case == ZFS_CASE_MIXED) {
```



```
1213                 vfs_set_feature(vfsp, VFSFT_DIRENTFLAGS);
1214                 vfs_set_feature(vfsp, VFSFT_CASEINSENSITIVE);
1215         }
1216         vfs_set_feature(vfsp, VFSFT_ZEROCOPY_SUPPORTED);

1218         if (dmu_objset_is_snapshot(zfsvfs->z_os)) {
1219                 uint64_t pval;

1221                 atime_changed_cb(zfsvfs, B_FALSE);
1222                 readonly_changed_cb(zfsvfs, B_TRUE);
1223                 if (error = dsl_prop_get_integer(osname, "xattr", &pval, NULL))
1224                         goto out;
1225                 xattr_changed_cb(zfsvfs, pval);
1226                 zfsvfs->z_issnap = B_TRUE;
1227                 zfsvfs->z_os->os_sync = ZFS_SYNC_DISABLED;

1229                 mutex_enter(&zfsvfs->z_os->os_user_ptr_lock);
1230                 dmu_objset_set_user(zfsvfs->z_os, zfsvfs);
1231                 mutex_exit(&zfsvfs->z_os->os_user_ptr_lock);
1232         } else {
1233                 error = zfsvfs_setup(zfsvfs, B_TRUE);
1234         }

1236         if (!zfsvfs->z_issnap)
1237                 zfsctl_create(zfsvfs);
1238 out:
1239         if (error) {
1240                 dmu_objset_disown(zfsvfs->z_os, zfsvfs);
1241                 zfsvfs_free(zfsvfs);
1242         } else {
1243                 atomic_inc_32(&zfs_active_fs_count);
1243                 atomic_add_32(&zfs_active_fs_count, 1);
1244         }

1246         return (error);
1247 }
______unchanged_portion_omitted_

2145 static void
2146 zfs_freevfs(vfs_t *vfsp)
2147 {
2148         zfsvfs_t *zfsvfs = vfsp->vfs_data;

2150         /*
2151          * If this is a snapshot, we have an extra VFS_HOLD on our parent
2152          * from zfs_mount().  Release it here.  If we came through
2153          * zfs_mountroot() instead, we didn't grab an extra hold, so
2154          * skip the VFS_RELE for rootvfs.
2155          */
2156         if (zfsvfs->z_issnap && (vfsp != rootvfs))
2157                 VFS_RELE(zfsvfs->z_parent->z_vfs);

2159         zfsvfs_free(zfsvfs);

2161         atomic_dec_32(&zfs_active_fs_count);
2161         atomic_add_32(&zfs_active_fs_count, -1);
2162 }
______unchanged_portion_omitted_
```

```
**********************************************************
   13265 Mon Jul 28 07:44:24 2014
new/usr/src/uts/common/fs/zfs/zio_inject.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 391 /*
 392  * Create a new handler for the given record.  We add it to the list, adding
 393  * a reference to the spa_t in the process.  We increment zio_injection_enabled,
 394  * which is the switch to trigger all fault injection.
 395  */
 396 int
 397 zio_inject_fault(char *name, int flags, int *id, zinject_record_t *record)
 398 {
 399         inject_handler_t *handler;
 400         int error;
 401         spa_t *spa;

 403         /*
 404          * If this is pool-wide metadata, make sure we unload the corresponding
 405          * spa_t, so that the next attempt to load it will trigger the fault.
 406          * We call spa_reset() to unload the pool appropriately.
 407          */
 408         if (flags & ZINJECT_UNLOAD_SPA)
 409                 if ((error = spa_reset(name)) != 0)
 410                         return (error);

 412         if (!(flags & ZINJECT_NULL)) {
 413                 /*
 414                  * spa_inject_ref() will add an injection reference, which will
 415                  * prevent the pool from being removed from the namespace while
 416                  * still allowing it to be unloaded.
 417                  */
 418                 if ((spa = spa_inject_addref(name)) == NULL)
 419                         return (SET_ERROR(ENOENT));

 421                 handler = kmem_alloc(sizeof (inject_handler_t), KM_SLEEP);

 423                 rw_enter(&inject_lock, RW_WRITER);

 425                 *id = handler->zi_id = inject_next_id++;
 426                 handler->zi_spa = spa;
 427                 handler->zi_record = *record;
 428                 list_insert_tail(&inject_handlers, handler);
 429                 atomic_inc_32(&zio_injection_enabled);
 429                 atomic_add_32(&zio_injection_enabled, 1);

 431                 rw_exit(&inject_lock);
 432         }

 434         /*
 435          * Flush the ARC, so that any attempts to read this data will end up
 436          * going to the ZIO layer.  Note that this is a little overkill, but
 437          * we don't have the necessary ARC interfaces to do anything else, and
 438          * fault injection isn't a performance critical path.
 439          */
 440         if (flags & ZINJECT_FLUSH_ARC)
 441                 arc_flush(NULL);

 443         return (0);
 444 }
______unchanged_portion_omitted_

 480 /*
 481  * Clear the fault handler with the given identifier, or return ENOENT if none
```



```
 482  * exists.
 483  */
 484 int
 485 zio_clear_fault(int id)
 486 {
 487         inject_handler_t *handler;

 489         rw_enter(&inject_lock, RW_WRITER);

 491         for (handler = list_head(&inject_handlers); handler != NULL;
 492             handler = list_next(&inject_handlers, handler))
 493                 if (handler->zi_id == id)
 494                         break;

 496         if (handler == NULL) {
 497                 rw_exit(&inject_lock);
 498                 return (SET_ERROR(ENOENT));
 499         }

 501         list_remove(&inject_handlers, handler);
 502         rw_exit(&inject_lock);

 504         spa_inject_delref(handler->zi_spa);
 505         kmem_free(handler, sizeof (inject_handler_t));
 506         atomic_dec_32(&zio_injection_enabled);
 506         atomic_add_32(&zio_injection_enabled, -1);

 508         return (0);
 509 }
______unchanged_portion_omitted_
```

```
**********************************************************
   59184 Mon Jul 28 07:44:24 2014
new/usr/src/uts/common/inet/ilb/ilb.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 512 /*
 513  * Add an ILB rule.
 514  */
 515 int
 516 ilb_rule_add(ilb_stack_t *ilbs, zoneid_t zoneid, const ilb_rule_cmd_t *cmd)
 517 {
 518         ilb_rule_t *rule;
 519         netstackid_t stackid;
 520         int ret;
 521         in_port_t min_port, max_port;
 522         int64_t num_src;

 524         /* Sanity checks. */
 525         if (cmd->ip_ver != IPPROTO_IP && cmd->ip_ver != IPPROTO_IPV6)
 526                 return (EINVAL);

 528         /* Need to support SCTP... */
 529         if (cmd->proto != IPPROTO_TCP && cmd->proto != IPPROTO_UDP)
 530                 return (EINVAL);

 532         /* For full NAT, the NAT source must be supplied. */
 533         if (cmd->topo == ILB_TOPO_IMPL_NAT) {
 534                 if (IS_ADDR_UNSPEC(&cmd->nat_src_start) ||
 535                     IS_ADDR_UNSPEC(&cmd->nat_src_end)) {
 536                         return (EINVAL);
 537                 }
 538         }

 540         /* Check invalid mask */
 541         if ((cmd->flags & ILB_RULE_STICKY) &&
 542             IS_ADDR_UNSPEC(&cmd->sticky_mask)) {
 543                 return (EINVAL);
 544         }

 546         /* Port is passed in network byte order. */
 547         min_port = ntohs(cmd->min_port);
 548         max_port = ntohs(cmd->max_port);
 549         if (min_port > max_port)
 550                 return (EINVAL);

 552         /* min_port == 0 means "all ports". Make it so */
 553         if (min_port == 0) {
 554                 min_port = 1;
 555                 max_port = 65535;
 556         }

 558         /* Funny address checking. */
 559         if (cmd->ip_ver == IPPROTO_IP) {
 560                 in_addr_t v4_addr1, v4_addr2;

 562                 v4_addr1 = cmd->vip.s6_addr32[3];
 563                 if ((*(uchar_t *)&v4_addr1) == IN_LOOPBACKNET ||
 564                     CLASSD(v4_addr1) || v4_addr1 == INADDR_BROADCAST ||
 565                     v4_addr1 == INADDR_ANY ||
 566                     !IN6_IS_ADDR_V4MAPPED(&cmd->vip)) {
 567                         return (EINVAL);
 568                 }

 570                 if (cmd->topo == ILB_TOPO_IMPL_NAT) {
```



```
 571                         v4_addr1 = ntohl(cmd->nat_src_start.s6_addr32[3]);
 572                         v4_addr2 = ntohl(cmd->nat_src_end.s6_addr32[3]);
 573                         if ((*(uchar_t *)&v4_addr1) == IN_LOOPBACKNET ||
 574                             (*(uchar_t *)&v4_addr2) == IN_LOOPBACKNET ||
 575                             v4_addr1 == INADDR_BROADCAST ||
 576                             v4_addr2 == INADDR_BROADCAST ||
 577                             v4_addr1 == INADDR_ANY || v4_addr2 == INADDR_ANY ||
 578                             CLASSD(v4_addr1) || CLASSD(v4_addr2) ||
 579                             !IN6_IS_ADDR_V4MAPPED(&cmd->nat_src_start) ||
 580                             !IN6_IS_ADDR_V4MAPPED(&cmd->nat_src_end)) {
 581                                 return (EINVAL);
 582                         }

 584                         num_src = v4_addr2 - v4_addr1 + 1;
 585                         if (v4_addr1 > v4_addr2 || num_src > ILB_MAX_NAT_SRC)
 586                                 return (EINVAL);
 587                 }
 588         } else {
 589                 if (IN6_IS_ADDR_LOOPBACK(&cmd->vip) ||
 590                     IN6_IS_ADDR_MULTICAST(&cmd->vip) ||
 591                     IN6_IS_ADDR_UNSPECIFIED(&cmd->vip) ||
 592                     IN6_IS_ADDR_V4MAPPED(&cmd->vip)) {
 593                         return (EINVAL);
 594                 }

 596                 if (cmd->topo == ILB_TOPO_IMPL_NAT) {
 597                         if (IN6_IS_ADDR_LOOPBACK(&cmd->nat_src_start) ||
 598                             IN6_IS_ADDR_LOOPBACK(&cmd->nat_src_end) ||
 599                             IN6_IS_ADDR_MULTICAST(&cmd->nat_src_start) ||
 600                             IN6_IS_ADDR_MULTICAST(&cmd->nat_src_end) ||
 601                             IN6_IS_ADDR_UNSPECIFIED(&cmd->nat_src_start) ||
 602                             IN6_IS_ADDR_UNSPECIFIED(&cmd->nat_src_end) ||
 603                             IN6_IS_ADDR_V4MAPPED(&cmd->nat_src_start) ||
 604                             IN6_IS_ADDR_V4MAPPED(&cmd->nat_src_end)) {
 605                                 return (EINVAL);
 606                         }

 608                         if ((num_src = num_nat_src_v6(&cmd->nat_src_start,
 609                             &cmd->nat_src_end)) < 0 ||
 610                             num_src > ILB_MAX_NAT_SRC) {
 611                                 return (EINVAL);
 612                         }
 613                 }
 614         }

 616         mutex_enter(&ilbs->ilbs_g_lock);
 617         if (ilbs->ilbs_g_hash == NULL)
 618                 ilb_rule_hash_init(ilbs);
 619         if (ilbs->ilbs_c2s_conn_hash == NULL) {
 620                 ASSERT(ilbs->ilbs_s2c_conn_hash == NULL);
 621                 ilb_conn_hash_init(ilbs);
 622                 ilb_nat_src_init(ilbs);
 623         }

 625         /* Make sure that the new rule does not duplicate an existing one. */
 626         if (ilb_match_rule(ilbs, zoneid, cmd->name, cmd->ip_ver, cmd->proto,
 627             min_port, max_port, &cmd->vip)) {
 628                 mutex_exit(&ilbs->ilbs_g_lock);
 629                 return (EEXIST);
 630         }

 632         rule = kmem_zalloc(sizeof (ilb_rule_t), KM_NOSLEEP);
 633         if (rule == NULL) {
 634                 mutex_exit(&ilbs->ilbs_g_lock);
 635                 return (ENOMEM);
 636         }
```

```
638         /* ir_name is all 0 to begin with */
639         (void) memcpy(rule->ir_name, cmd->name, ILB_RULE_NAMESZ - 1);

641         rule->ir_ks_instance = atomic_inc_uint_nv(&ilb_kstat_instance);
641         rule->ir_ks_instance = atomic_add_int_nv(&ilb_kstat_instance, 1);
642         stackid = (netstackid_t)(uintptr_t)ilbs->ilbs_ksp->ks_private;
643         if ((rule->ir_ksp = ilb_rule_kstat_init(stackid, rule)) == NULL) {
644                 ret = ENOMEM;
645                 goto error;
646         }

648         if (cmd->topo == ILB_TOPO_IMPL_NAT) {
649                 rule->ir_nat_src_start = cmd->nat_src_start;
650                 rule->ir_nat_src_end = cmd->nat_src_end;
651         }

653         rule->ir_ipver = cmd->ip_ver;
654         rule->ir_proto = cmd->proto;
655         rule->ir_topo = cmd->topo;

657         rule->ir_min_port = min_port;
658         rule->ir_max_port = max_port;
659         if (rule->ir_min_port != rule->ir_max_port)
660                 rule->ir_port_range = B_TRUE;
661         else
662                 rule->ir_port_range = B_FALSE;

664         rule->ir_zoneid = zoneid;

666         rule->ir_target_v6 = cmd->vip;
667         rule->ir_servers = NULL;

669         /*
670          * The default connection drain timeout is indefinite (value 0),
671          * meaning we will wait for all connections to finish.  So we
672          * can assign cmd->conn_drain_timeout to it directly.
673          */
674         rule->ir_conn_drain_timeout = cmd->conn_drain_timeout;
675         if (cmd->nat_expiry != 0) {
676                 rule->ir_nat_expiry = cmd->nat_expiry;
677         } else {
678                 switch (rule->ir_proto) {
679                 case IPPROTO_TCP:
680                         rule->ir_nat_expiry = ilb_conn_tcp_expiry;
681                         break;
682                 case IPPROTO_UDP:
683                         rule->ir_nat_expiry = ilb_conn_udp_expiry;
684                         break;
685                 default:
686                         cmn_err(CE_PANIC, "data corruption: wrong ir_proto: %p",
687                             (void *)rule);
688                         break;
689                 }
690         }
691         if (cmd->sticky_expiry != 0)
692                 rule->ir_sticky_expiry = cmd->sticky_expiry;
693         else
694                 rule->ir_sticky_expiry = ilb_sticky_expiry;

696         if (cmd->flags & ILB_RULE_STICKY) {
697                 rule->ir_flags |= ILB_RULE_STICKY;
698                 rule->ir_sticky_mask = cmd->sticky_mask;
699                 if (ilbs->ilbs_sticky_hash == NULL)
700                         ilb_sticky_hash_init(ilbs);
701         }
```



```
702         if (cmd->flags & ILB_RULE_ENABLED)
703                 rule->ir_flags |= ILB_RULE_ENABLED;

705         mutex_init(&rule->ir_lock, NULL, MUTEX_DEFAULT, NULL);
706         cv_init(&rule->ir_cv, NULL, CV_DEFAULT, NULL);

708         rule->ir_refcnt = 1;

710         switch (cmd->algo) {
711         case ILB_ALG_IMPL_ROUNDROBIN:
712                 if ((rule->ir_alg = ilb_alg_rr_init(rule, NULL)) == NULL) {
713                         ret = ENOMEM;
714                         goto error;
715                 }
716                 rule->ir_alg_type = ILB_ALG_IMPL_ROUNDROBIN;
717                 break;
718         case ILB_ALG_IMPL_HASH_IP:
719         case ILB_ALG_IMPL_HASH_IP_SPORT:
720         case ILB_ALG_IMPL_HASH_IP_VIP:
721                 if ((rule->ir_alg = ilb_alg_hash_init(rule,
722                     &cmd->algo)) == NULL) {
723                         ret = ENOMEM;
724                         goto error;
725                 }
726                 rule->ir_alg_type = cmd->algo;
727                 break;
728         default:
729                 ret = EINVAL;
730                 goto error;
731         }

733         /* Add it to the global list and hash array at the end. */
734         ilb_rule_g_add(ilbs, rule);
735         ilb_rule_hash_add(ilbs, rule, &cmd->vip);

737         mutex_exit(&ilbs->ilbs_g_lock);

739         return (0);

741 error:
742         mutex_exit(&ilbs->ilbs_g_lock);
743         if (rule->ir_ksp != NULL) {
744                 /* stackid must be initialized if ir_ksp != NULL */
745                 kstat_delete_netstack(rule->ir_ksp, stackid);
746         }
747         kmem_free(rule, sizeof (ilb_rule_t));
748         return (ret);
749 }
```

_____***unchanged_portion_omitted_***

****
16877 Mon Jul 28 07:44:24 2014
new/usr/src/uts/common/inet/ilb/ilb_nat.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
 189 /* An arena name is "ilb_ns" + "_xxxxxxxxxx"  */
 190 #define ARENA_NAMESZ    18
 191 #define NAT_PORT_START  4096
 192 #define NAT_PORT_SIZE   65535 - NAT_PORT_START

 194 /*
 195  * Check if the NAT source and back end server pair ilb_nat_src_entry_t
 196  * exists.  If it does, increment the refcnt and return it.  If not, create
 197  * one and return it.
 198  */
 199 static ilb_nat_src_entry_t *
 200 ilb_find_nat_src(ilb_stack_t *ilbs, const in6_addr_t *nat_src,
 201     const in6_addr_t *serv_addr, in_port_t port)
 202 {
 203         ilb_nat_src_entry_t *tmp;
 204         uint32_t idx;
 205         char arena_name[ARENA_NAMESZ];
 206         list_t *head;

 208         ILB_NAT_SRC_HASH(idx, &nat_src->s6_addr32[3], &serv_addr->s6_addr32[3],
 209             ilbs->ilbs_nat_src_hash_size);
 210         mutex_enter(&ilbs->ilbs_nat_src[idx].nsh_lock);
 211         head = &ilbs->ilbs_nat_src[idx].nsh_head;
 212         for (tmp = list_head(head); tmp != NULL; tmp = list_next(head, tmp)) {
 213                 if (IN6_ARE_ADDR_EQUAL(&tmp->nse_src_addr, nat_src) &&
 214                     IN6_ARE_ADDR_EQUAL(&tmp->nse_serv_addr, serv_addr) &&
 215                     (port == tmp->nse_port || port == 0 ||
 216                     tmp->nse_port == 0)) {
 217                         break;
 218                 }
 219         }
 220         /* Found one, return it. */
 221         if (tmp != NULL) {
 222                 tmp->nse_refcnt++;
 223                 mutex_exit(&ilbs->ilbs_nat_src[idx].nsh_lock);
 224                 return (tmp);
 225         }

 227         tmp = kmem_alloc(sizeof (ilb_nat_src_entry_t), KM_NOSLEEP);
 228         if (tmp == NULL) {
 229                 mutex_exit(&ilbs->ilbs_nat_src[idx].nsh_lock);
 230                 return (NULL);
 231         }
 232         tmp->nse_src_addr = *nat_src;
 233         tmp->nse_serv_addr = *serv_addr;
 234         tmp->nse_port = port;
 235         tmp->nse_nsh_lock = &ilbs->ilbs_nat_src[idx].nsh_lock;
 236         tmp->nse_refcnt = 1;

 238         (void) snprintf(arena_name, ARENA_NAMESZ, "ilb_ns_%u",
 239             atomic_inc_32_nv(&ilb_nat_src_instance));
 239             atomic_add_32_nv(&ilb_nat_src_instance, 1));
 240         if ((tmp->nse_port_arena = vmem_create(arena_name,
 241             (void *)NAT_PORT_START, NAT_PORT_SIZE, 1, NULL, NULL, NULL, 1,
 242             VM_SLEEP | VMC_IDENTIFIER)) == NULL) {
 243                 kmem_free(tmp, sizeof (*tmp));
 244                 return (NULL);
 245         }
```



```
 247         list_insert_tail(head, tmp);
 248         mutex_exit(&ilbs->ilbs_nat_src[idx].nsh_lock);

 250         return (tmp);
 251 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  140768 Mon Jul 28 07:44:24 2014
new/usr/src/uts/common/inet/ip.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 722 #define IPLATCH_REFHOLD(ipl) { \
 723         atomic_inc_32(&(ipl)->ipl_refcnt);              \
 723         atomic_add_32(&(ipl)->ipl_refcnt, 1);           \
 724         ASSERT((ipl)->ipl_refcnt != 0);                 \
 725 }

 727 #define IPLATCH_REFRELE(ipl) {                          \
 728         ASSERT((ipl)->ipl_refcnt != 0);                         \
 729         membar_exit();                                  \
 730         if (atomic_dec_32_nv(&(ipl)->ipl_refcnt) == 0)  \
 730         if (atomic_add_32_nv(&(ipl)->ipl_refcnt, -1) == 0)      \
 731                 iplatch_free(ipl);                      \
 732 }
______unchanged_portion_omitted_
```

```
**********************************************************
   84852 Mon Jul 28 07:44:25 2014
new/usr/src/uts/common/inet/ip/igmp.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 475 static uint_t
 476 igmp_query_in(ipha_t *ipha, igmpa_t *igmpa, ill_t *ill)
 477 {
 478         ilm_t   *ilm;
 479         int     timer;
 480         uint_t  next, current;
 481         ip_stack_t      *ipst;

 483         ipst = ill->ill_ipst;
 484         ++ipst->ips_igmpstat.igps_rcv_queries;

 486         rw_enter(&ill->ill_mcast_lock, RW_WRITER);
 487         /*
 488          * In the IGMPv2 specification, there are 3 states and a flag.
 489          *
 490          * In Non-Member state, we simply don't have a membership record.
 491          * In Delaying Member state, our timer is running (ilm->ilm_timer
 492          * < INFINITY).  In Idle Member state, our timer is not running
 493          * (ilm->ilm_timer == INFINITY).
 494          *
 495          * The flag is ilm->ilm_state, it is set to IGMP_OTHERMEMBER if
 496          * we have heard a report from another member, or IGMP_IREPORTEDLAST
 497          * if I sent the last report.
 498          */
 499         if ((igmpa->igmpa_code == 0) ||
 500             (ipst->ips_igmp_max_version == IGMP_V1_ROUTER)) {
 501                 /*
 502                  * Query from an old router.
 503                  * Remember that the querier on this interface is old,
 504                  * and set the timer to the value in RFC 1112.
 505                  */
 506                 ill->ill_mcast_v1_time = 0;
 507                 ill->ill_mcast_v1_tset = 1;
 508                 if (ill->ill_mcast_type != IGMP_V1_ROUTER) {
 509                         ip1dbg(("Received IGMPv1 Query on %s, switching mode "
 510                             "to IGMP_V1_ROUTER\n", ill->ill_name));
 511                         atomic_inc_16(&ill->ill_ifptr->illif_mcast_v1);
 511                         atomic_add_16(&ill->ill_ifptr->illif_mcast_v1, 1);
 512                         ill->ill_mcast_type = IGMP_V1_ROUTER;
 513                 }

 515                 timer = SEC_TO_MSEC(IGMP_MAX_HOST_REPORT_DELAY);

 517                 if (ipha->ipha_dst != htonl(INADDR_ALLHOSTS_GROUP) ||
 518                     igmpa->igmpa_group != 0) {
 519                         ++ipst->ips_igmpstat.igps_rcv_badqueries;
 520                         rw_exit(&ill->ill_mcast_lock);
 521                         ill_mcast_timer_start(ill->ill_ipst);
 522                         return (0);
 523                 }

 525         } else {
 526                 in_addr_t group;

 528                 /*
 529                  * Query from a new router
 530                  * Simply do a validity check
 531                  */
 532                 group = igmpa->igmpa_group;
```



```
 533                 if (group != 0 && (!CLASSD(group))) {
 534                         ++ipst->ips_igmpstat.igps_rcv_badqueries;
 535                         rw_exit(&ill->ill_mcast_lock);
 536                         ill_mcast_timer_start(ill->ill_ipst);
 537                         return (0);
 538                 }

 540                 /*
 541                  * Switch interface state to v2 on receipt of a v2 query
 542                  * ONLY IF current state is v3.  Let things be if current
 543                  * state if v1 but do reset the v2-querier-present timer.
 544                  */
 545                 if (ill->ill_mcast_type == IGMP_V3_ROUTER) {
 546                         ip1dbg(("Received IGMPv2 Query on %s, switching mode "
 547                             "to IGMP_V2_ROUTER", ill->ill_name));
 548                         atomic_inc_16(&ill->ill_ifptr->illif_mcast_v2);
 548                         atomic_add_16(&ill->ill_ifptr->illif_mcast_v2, 1);
 549                         ill->ill_mcast_type = IGMP_V2_ROUTER;
 550                 }
 551                 ill->ill_mcast_v2_time = 0;
 552                 ill->ill_mcast_v2_tset = 1;

 554                 timer = DSEC_TO_MSEC((int)igmpa->igmpa_code);
 555         }

 557         if (ip_debug > 1) {
 558                 (void) mi_strlog(ill->ill_rq, 1, SL_TRACE,
 559                     "igmp_input: TIMER = igmp_code %d igmp_type 0x%x",
 560                     (int)ntohs(igmpa->igmpa_code),
 561                     (int)ntohs(igmpa->igmpa_type));
 562         }

 564         /*
 565          * -Start the timers in all of our membership records
 566          *  for the physical interface on which the query
 567          *  arrived, excluding those that belong to the "all
 568          *  hosts" group (224.0.0.1).
 569          *
 570          * -Restart any timer that is already running but has
 571          *  a value longer than the requested timeout.
 572          *
 573          * -Use the value specified in the query message as
 574          *  the maximum timeout.
 575          */
 576         next = (unsigned)INFINITY;

 578         current = CURRENT_MSTIME;
 579         for (ilm = ill->ill_ilm; ilm; ilm = ilm->ilm_next) {

 581                 /*
 582                  * A multicast router joins INADDR_ANY address
 583                  * to enable promiscuous reception of all
 584                  * mcasts from the interface. This INADDR_ANY
 585                  * is stored in the ilm_v6addr as V6 unspec addr
 586                  */
 587                 if (!IN6_IS_ADDR_V4MAPPED(&ilm->ilm_v6addr))
 588                         continue;
 589                 if (ilm->ilm_addr == htonl(INADDR_ANY))
 590                         continue;
 591                 if (ilm->ilm_addr != htonl(INADDR_ALLHOSTS_GROUP) &&
 592                     (igmpa->igmpa_group == 0) ||
 593                     (igmpa->igmpa_group == ilm->ilm_addr)) {
 594                         if (ilm->ilm_timer > timer) {
 595                                 MCAST_RANDOM_DELAY(ilm->ilm_timer, timer);
 596                                 if (ilm->ilm_timer < next)
 597                                         next = ilm->ilm_timer;
```

```
 598                                ilm->ilm_timer += current;
 599                        }
 600                }
 601        }
 602        rw_exit(&ill->ill_mcast_lock);
 603        /*
 604         * No packets have been sent above - no
 605         * ill_mcast_send_queued is needed.
 606         */
 607        ill_mcast_timer_start(ill->ill_ipst);

 609        return (next);
 610 }
```

*_____unchanged_portion_omitted_*

```
1639 /*
1640  * Calculate the Older Version Querier Present timeout value, in number
1641  * of slowtimo intervals, for the given ill.
1642  */
1643 #define OVQP(ill) \
1644         ((1000 * (((ill)->ill_mcast_rv * (ill)->ill_mcast_qi) \
1645         + MCAST_QUERY_RESP_INTERVAL)) / MCAST_SLOWTIMO_INTERVAL)

1647 /*
1648  * igmp_slowtimo:
1649  * - Resets to new router if we didnt we hear from the router
1650  *   in IGMP_AGE_THRESHOLD seconds.
1651  * - Resets slowtimeout.
1652  * Check for ips_igmp_max_version ensures that we don't revert to a higher
1653  * IGMP version than configured.
1654  */
1655 void
1656 igmp_slowtimo(void *arg)
1657 {
1658         ill_t   *ill;
1659         ill_if_t *ifp;
1660         avl_tree_t *avl_tree;
1661         ip_stack_t *ipst = (ip_stack_t *)arg;

1663         ASSERT(arg != NULL);

1665         /*
1666          * The ill_if_t list is circular, hence the odd loop parameters.
1667          *
1668          * We can't use the ILL_START_WALK and ill_next() wrappers for this
1669          * walk, as we need to check the illif_mcast_* fields in the ill_if_t
1670          * structure (allowing us to skip if none of the instances have timers
1671          * running).
1672          */
1673         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
1674         for (ifp = IP_V4_ILL_G_LIST(ipst);
1675             ifp != (ill_if_t *)&IP_V4_ILL_G_LIST(ipst);
1676             ifp = ifp->illif_next) {
1677                 /*
1678                  * illif_mcast_v[12] are set using atomics. If an ill hears
1679                  * a V1 or V2 query now and we miss seeing the count now,
1680                  * we will see it the next time igmp_slowtimo is called.
1681                  */
1682                 if (ifp->illif_mcast_v1 == 0 && ifp->illif_mcast_v2 == 0)
1683                         continue;

1685                 avl_tree = &ifp->illif_avl_by_ppa;
1686                 for (ill = avl_first(avl_tree); ill != NULL;
1687                     ill = avl_walk(avl_tree, ill, AVL_AFTER)) {
1688                         /* Make sure the ill isn't going away. */
1689                         if (!ill_check_and_refhold(ill))
```



```
1690                                 continue;
1691                         rw_exit(&ipst->ips_ill_g_lock);
1692                         rw_enter(&ill->ill_mcast_lock, RW_WRITER);
1693                         if (ill->ill_mcast_v1_tset == 1)
1694                                 ill->ill_mcast_v1_time++;
1695                         if (ill->ill_mcast_v2_tset == 1)
1696                                 ill->ill_mcast_v2_time++;
1697                         if ((ill->ill_mcast_type == IGMP_V1_ROUTER) &&
1698                             (ipst->ips_igmp_max_version >= IGMP_V2_ROUTER) &&
1699                             (ill->ill_mcast_v1_time >= OVQP(ill))) {
1700                                 if ((ill->ill_mcast_v2_tset > 0) ||
1701                                     (ipst->ips_igmp_max_version ==
1702                                     IGMP_V2_ROUTER)) {
1703                                         ip1dbg(("V1 query timer "
1704                                             "expired on %s; switching "
1705                                             "mode to IGMP_V2\n",
1706                                             ill->ill_name));
1707                                         ill->ill_mcast_type =
1708                                             IGMP_V2_ROUTER;
1709                                 } else {
1710                                         ip1dbg(("V1 query timer "
1711                                             "expired on %s; switching "
1712                                             "mode to IGMP_V3\n",
1713                                             ill->ill_name));
1714                                         ill->ill_mcast_type =
1715                                             IGMP_V3_ROUTER;
1716                                 }
1717                                 ill->ill_mcast_v1_time = 0;
1718                                 ill->ill_mcast_v1_tset = 0;
1719                                 atomic_dec_16(&ifp->illif_mcast_v1);
1719                                 atomic_add_16(&ifp->illif_mcast_v1, -1);
1720                         }
1721                         if ((ill->ill_mcast_type == IGMP_V2_ROUTER) &&
1722                             (ipst->ips_igmp_max_version >= IGMP_V3_ROUTER) &&
1723                             (ill->ill_mcast_v2_time >= OVQP(ill))) {
1724                                 ip1dbg(("V2 query timer expired on "
1725                                     "%s; switching mode to IGMP_V3\n",
1726                                     ill->ill_name));
1727                                 ill->ill_mcast_type = IGMP_V3_ROUTER;
1728                                 ill->ill_mcast_v2_time = 0;
1729                                 ill->ill_mcast_v2_tset = 0;
1730                                 atomic_dec_16(&ifp->illif_mcast_v2);
1730                                 atomic_add_16(&ifp->illif_mcast_v2, -1);
1731                         }
1732                         rw_exit(&ill->ill_mcast_lock);
1733                         ill_refrele(ill);
1734                         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
1735                 }
1736         }
1737         rw_exit(&ipst->ips_ill_g_lock);
1738         ill_mcast_timer_start(ipst);
1739         mutex_enter(&ipst->ips_igmp_slowtimeout_lock);
1740         ipst->ips_igmp_slowtimeout_id = timeout(igmp_slowtimo, (void *)ipst,
1741             MSEC_TO_TICK(MCAST_SLOWTIMO_INTERVAL));
1742         mutex_exit(&ipst->ips_igmp_slowtimeout_lock);
1743 }

1745 /*
1746  * mld_slowtimo:
1747  * - Resets to newer version if we didn't hear from the older version router
1748  *   in MLD_AGE_THRESHOLD seconds.
1749  * - Restarts slowtimeout.
1750  * Check for ips_mld_max_version ensures that we don't revert to a higher
1751  * IGMP version than configured.
1752  */
1753 void
```

```
1754 mld_slowtimo(void *arg)
1755 {
1756         ill_t *ill;
1757         ill_if_t *ifp;
1758         avl_tree_t *avl_tree;
1759         ip_stack_t *ipst = (ip_stack_t *)arg;

1761         ASSERT(arg != NULL);
1762         /* See comments in igmp_slowtimo() above... */
1763         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
1764         for (ifp = IP_V6_ILL_G_LIST(ipst);
1765             ifp != (ill_if_t *)&IP_V6_ILL_G_LIST(ipst);
1766             ifp = ifp->illif_next) {
1767                 if (ifp->illif_mcast_v1 == 0)
1768                         continue;

1770                 avl_tree = &ifp->illif_avl_by_ppa;
1771                 for (ill = avl_first(avl_tree); ill != NULL;
1772                     ill = avl_walk(avl_tree, ill, AVL_AFTER)) {
1773                         /* Make sure the ill isn't going away. */
1774                         if (!ill_check_and_refhold(ill))
1775                                 continue;
1776                         rw_exit(&ipst->ips_ill_g_lock);
1777                         rw_enter(&ill->ill_mcast_lock, RW_WRITER);
1778                         if (ill->ill_mcast_v1_tset == 1)
1779                                 ill->ill_mcast_v1_time++;
1780                         if ((ill->ill_mcast_type == MLD_V1_ROUTER) &&
1781                             (ipst->ips_mld_max_version >= MLD_V2_ROUTER) &&
1782                             (ill->ill_mcast_v1_time >= OVQP(ill))) {
1783                                 ip1dbg(("MLD query timer expired on"
1784                                     " %s; switching mode to MLD_V2\n",
1785                                     ill->ill_name));
1786                                 ill->ill_mcast_type = MLD_V2_ROUTER;
1787                                 ill->ill_mcast_v1_time = 0;
1788                                 ill->ill_mcast_v1_tset = 0;
1789                                 atomic_dec_16(&ifp->illif_mcast_v1);
1789                                 atomic_add_16(&ifp->illif_mcast_v1, -1);
1790                         }
1791                         rw_exit(&ill->ill_mcast_lock);
1792                         ill_refrele(ill);
1793                         rw_enter(&ipst->ips_ill_g_lock, RW_READER);
1794                 }
1795         }
1796         rw_exit(&ipst->ips_ill_g_lock);
1797         ill_mcast_timer_start(ipst);
1798         mutex_enter(&ipst->ips_mld_slowtimeout_lock);
1799         ipst->ips_mld_slowtimeout_id = timeout(mld_slowtimo, (void *)ipst,
1800             MSEC_TO_TICK(MCAST_SLOWTIMO_INTERVAL));
1801         mutex_exit(&ipst->ips_mld_slowtimeout_lock);
1802 }
```

*unchanged_portion_omitted_*

```
2179 /*
2180  * Handles an MLDv1 Listener Query.  Returns 0 on error, or the appropriate
2181  * (non-zero, unsigned) timer value to be set on success.
2182  */
2183 static uint_t
2184 mld_query_in(mld_hdr_t *mldh, ill_t *ill)
2185 {
2186         ilm_t   *ilm;
2187         int     timer;
2188         uint_t  next, current;
2189         in6_addr_t *v6group;

2191         BUMP_MIB(ill->ill_icmp6_mib, ipv6IfIcmpInGroupMembQueries);
```



```
2193         /*
2194          * In the MLD specification, there are 3 states and a flag.
2195          *
2196          * In Non-Listener state, we simply don't have a membership record.
2197          * In Delaying state, our timer is running (ilm->ilm_timer < INFINITY)
2198          * In Idle Member state, our timer is not running (ilm->ilm_timer ==
2199          * INFINITY)
2200          *
2201          * The flag is ilm->ilm_state, it is set to IGMP_OTHERMEMBER if
2202          * we have heard a report from another member, or IGMP_IREPORTEDLAST
2203          * if I sent the last report.
2204          */
2205         v6group = &mldh->mld_addr;
2206         if (!(IN6_IS_ADDR_UNSPECIFIED(v6group)) &&
2207             ((!IN6_IS_ADDR_MULTICAST(v6group)))) {
2208                 BUMP_MIB(ill->ill_icmp6_mib, ipv6IfIcmpInGroupMembBadQueries);
2209                 return (0);
2210         }

2212         /* Need to do compatibility mode checking */
2213         rw_enter(&ill->ill_mcast_lock, RW_WRITER);
2214         ill->ill_mcast_v1_time = 0;
2215         ill->ill_mcast_v1_tset = 1;
2216         if (ill->ill_mcast_type == MLD_V2_ROUTER) {
2217                 ip1dbg(("Received MLDv1 Query on %s, switching mode to "
2218                     "MLD_V1_ROUTER\n", ill->ill_name));
2219                 atomic_inc_16(&ill->ill_ifptr->illif_mcast_v1);
2219                 atomic_add_16(&ill->ill_ifptr->illif_mcast_v1, 1);
2220                 ill->ill_mcast_type = MLD_V1_ROUTER;
2221         }

2223         timer = (int)ntohs(mldh->mld_maxdelay);
2224         if (ip_debug > 1) {
2225                 (void) mi_strlog(ill->ill_rq, 1, SL_TRACE,
2226                     "mld_input: TIMER = mld_maxdelay %d mld_type 0x%x",
2227                     timer, (int)mldh->mld_type);
2228         }

2230         /*
2231          * -Start the timers in all of our membership records for
2232          * the physical interface on which the query arrived,
2233          * excl:
2234          *      1.  those that belong to the "all hosts" group,
2235          *      2.  those with 0 scope, or 1 node-local scope.
2236          *
2237          * -Restart any timer that is already running but has a value
2238          * longer that the requested timeout.
2239          * -Use the value specified in the query message as the
2240          * maximum timeout.
2241          */
2242         next = INFINITY;

2244         current = CURRENT_MSTIME;
2245         for (ilm = ill->ill_ilm; ilm != NULL; ilm = ilm->ilm_next) {
2246                 ASSERT(!IN6_IS_ADDR_V4MAPPED(&ilm->ilm_v6addr));

2248                 if (IN6_IS_ADDR_UNSPECIFIED(&ilm->ilm_v6addr) ||
2249                     IN6_IS_ADDR_MC_NODELOCAL(&ilm->ilm_v6addr) ||
2250                     IN6_IS_ADDR_MC_RESERVED(&ilm->ilm_v6addr))
2251                         continue;
2252                 if ((!IN6_ARE_ADDR_EQUAL(&ilm->ilm_v6addr,
2253                     &ipv6_all_hosts_mcast)) &&
2254                     (IN6_IS_ADDR_UNSPECIFIED(v6group)) ||
2255                     (IN6_ARE_ADDR_EQUAL(v6group, &ilm->ilm_v6addr))) {
2256                         if (timer == 0) {
2257                                 /* Respond immediately */
```

```
2258                                    ilm->ilm_timer = INFINITY;
2259                                    ilm->ilm_state = IGMP_IREPORTEDLAST;
2260                                    mld_sendpkt(ilm, MLD_LISTENER_REPORT, NULL);
2261                                    break;
2262                            }
2263                            if (ilm->ilm_timer > timer) {
2264                                    MCAST_RANDOM_DELAY(ilm->ilm_timer, timer);
2265                                    if (ilm->ilm_timer < next)
2266                                            next = ilm->ilm_timer;
2267                                    ilm->ilm_timer += current;
2268                            }
2269                            break;
2270                    }
2271            }
2272            rw_exit(&ill->ill_mcast_lock);
2273            /* Send any deferred/queued IP packets */
2274            ill_mcast_send_queued(ill);
2275            ill_mcast_timer_start(ill->ill_ipst);

2277            return (next);
2278 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   39651 Mon Jul 28 07:44:25 2014
new/usr/src/uts/common/inet/ip/ip_attr.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */

  22 /*
  23  * Copyright (c) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
  24  */
  25 /* Copyright (c) 1990 Mentat Inc. */

  27 #include <sys/types.h>
  28 #include <sys/stream.h>
  29 #include <sys/strsun.h>
  30 #include <sys/zone.h>
  31 #include <sys/ddi.h>
  32 #include <sys/sunddi.h>
  33 #include <sys/cmn_err.h>
  34 #include <sys/debug.h>
  35 #include <sys/atomic.h>

  37 #include <sys/systm.h>
  38 #include <sys/param.h>
  39 #include <sys/kmem.h>
  40 #include <sys/sdt.h>
  41 #include <sys/socket.h>
  42 #include <sys/mac.h>
  43 #include <net/if.h>
  44 #include <net/if_arp.h>
  45 #include <net/route.h>
  46 #include <sys/sockio.h>
  47 #include <netinet/in.h>
  48 #include <net/if_dl.h>

  50 #include <inet/common.h>
  51 #include <inet/mi.h>
  52 #include <inet/mib2.h>
  53 #include <inet/nd.h>
  54 #include <inet/arp.h>
  55 #include <inet/snmpcom.h>
  56 #include <inet/kstatcom.h>

  58 #include <netinet/igmp_var.h>
  59 #include <netinet/ip6.h>
  60 #include <netinet/icmp6.h>
  61 #include <netinet/sctp.h>
```



```
  63 #include <inet/ip.h>
  64 #include <inet/ip_impl.h>
  65 #include <inet/ip6.h>
  66 #include <inet/ip6_asp.h>
  67 #include <inet/tcp.h>
  68 #include <inet/ip_multi.h>
  69 #include <inet/ip_if.h>
  70 #include <inet/ip_ire.h>
  71 #include <inet/ip_ftable.h>
  72 #include <inet/ip_rts.h>
  73 #include <inet/optcom.h>
  74 #include <inet/ip_ndp.h>
  75 #include <inet/ip_listutils.h>
  76 #include <netinet/igmp.h>
  77 #include <netinet/ip_mroute.h>
  78 #include <inet/ipp_common.h>

  80 #include <net/pfkeyv2.h>
  81 #include <inet/sadb.h>
  82 #include <inet/ipsec_impl.h>
  83 #include <inet/ipdrop.h>
  84 #include <inet/ip_netinfo.h>
  85 #include <sys/squeue_impl.h>
  86 #include <sys/squeue.h>

  88 #include <inet/ipclassifier.h>
  89 #include <inet/sctp_ip.h>
  90 #include <inet/sctp/sctp_impl.h>
  91 #include <inet/udp_impl.h>
  92 #include <sys/sunddi.h>

  94 #include <sys/tsol/label.h>
  95 #include <sys/tsol/tnet.h>

  97 /*
  98  * Release a reference on ip_xmit_attr.
  99  * The reference is acquired by conn_get_ixa()
 100  */
 101 #define IXA_REFRELE(ixa)                                        \
 102 {                                                               \
 103         if (atomic_dec_32_nv(&(ixa)->ixa_refcnt) == 0)  \
 103         if (atomic_add_32_nv(&(ixa)->ixa_refcnt, -1) == 0)      \
 104                 ixa_inactive(ixa);                              \
 105 }

 107 #define IXA_REFHOLD(ixa)                                        \
 108 {                                                               \
 109         ASSERT((ixa)->ixa_refcnt != 0);                         \
 110         atomic_inc_32(&(ixa)->ixa_refcnt);                      \
 110         atomic_add_32(&(ixa)->ixa_refcnt, 1);                   \
 111 }
_____unchanged_portion_omitted_

 746 static ip_xmit_attr_t *
 747 conn_get_ixa_impl(conn_t *connp, boolean_t replace, int kmflag)
 748 {
 749         ip_xmit_attr_t  *ixa;
 750         ip_xmit_attr_t  *oldixa;

 752         mutex_enter(&connp->conn_lock);
 753         ixa = connp->conn_ixa;

 755         /* At least one references for the conn_t */
 756         ASSERT(ixa->ixa_refcnt >= 1);
 757         if (atomic_inc_32_nv(&ixa->ixa_refcnt) == 2) {
```

```
        if (atomic_add_32_nv(&ixa->ixa_refcnt, 1) == 2) {
                /* No other thread using conn_ixa */
                mutex_exit(&connp->conn_lock);
                return (ixa);
        }
        ixa = kmem_alloc(sizeof (*ixa), kmflag);
        if (ixa == NULL) {
                mutex_exit(&connp->conn_lock);
                ixa_refrele(connp->conn_ixa);
                return (NULL);
        }
        ixa_safe_copy(connp->conn_ixa, ixa);

        /* Make sure we drop conn_lock before any refrele */
        if (replace) {
                ixa->ixa_refcnt++;      /* No atomic needed - not visible */
                oldixa = connp->conn_ixa;
                connp->conn_ixa = ixa;
                mutex_exit(&connp->conn_lock);
                IXA_REFRELE(oldixa);    /* Undo refcnt from conn_t */
        } else {
                oldixa = connp->conn_ixa;
                mutex_exit(&connp->conn_lock);
        }
        IXA_REFRELE(oldixa);    /* Undo above atomic_add_32_nv */

        return (ixa);
}
```

unchanged_portion_omitted_

```
/*
 * Return a ip_xmit_attr_t to use with a conn_t that is based on but
 * separate from conn_ixa.
 *
 * This "safe" copy has the pointers set to NULL
 * (since the pointers might be changed by another thread using
 * conn_ixa). The caller needs to check for NULL pointers to see
 * if ip_set_destination needs to be called to re-establish the pointers.
 */
ip_xmit_attr_t *
conn_get_ixa_exclusive(conn_t *connp)
{
        ip_xmit_attr_t *ixa;

        mutex_enter(&connp->conn_lock);
        ixa = connp->conn_ixa;

        /* At least one references for the conn_t */
        ASSERT(ixa->ixa_refcnt >= 1);

        /* Make sure conn_ixa doesn't disappear while we copy it */
        atomic_inc_32(&ixa->ixa_refcnt);
        atomic_add_32(&ixa->ixa_refcnt, 1);

        ixa = kmem_alloc(sizeof (*ixa), KM_NOSLEEP);
        if (ixa == NULL) {
                mutex_exit(&connp->conn_lock);
                ixa_refrele(connp->conn_ixa);
                return (NULL);
        }
        ixa_safe_copy(connp->conn_ixa, ixa);
        mutex_exit(&connp->conn_lock);
        IXA_REFRELE(connp->conn_ixa);
        return (ixa);
}
```

unchanged_portion_omitted_

```
**********************************************************
   26238 Mon Jul 28 07:44:25 2014
new/usr/src/uts/common/inet/ip/ip_dce.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_
```

```
 481 /*
 482  * Atomically looks for a non-default DCE, and if not found tries to create one.
 483  * If there is no memory it returns NULL.
 484  * When an entry is created we increase the generation number on
 485  * the default DCE so that conn_ip_output will detect there is a new DCE.
 486  */
 487 dce_t *
 488 dce_lookup_and_add_v4(ipaddr_t dst, ip_stack_t *ipst)
 489 {
 490         uint_t          hash;
 491         dcb_t           *dcb;
 492         dce_t           *dce;

 494         hash = IRE_ADDR_HASH(dst, ipst->ips_dce_hashsize);
 495         dcb = &ipst->ips_dce_hash_v4[hash];
 496         /*
 497          * Assuming that we get fairly even distribution across all of the
 498          * buckets, once one bucket is overly full, prune the whole cache.
 499          */
 500         if (dcb->dcb_cnt > ipst->ips_ip_dce_reclaim_threshold)
 501                 atomic_or_uint(&ipst->ips_dce_reclaim_needed, 1);
 502         rw_enter(&dcb->dcb_lock, RW_WRITER);
 503         for (dce = dcb->dcb_dce; dce != NULL; dce = dce->dce_next) {
 504                 if (dce->dce_v4addr == dst) {
 505                         mutex_enter(&dce->dce_lock);
 506                         if (!DCE_IS_CONDEMNED(dce)) {
 507                                 dce_refhold(dce);
 508                                 mutex_exit(&dce->dce_lock);
 509                                 rw_exit(&dcb->dcb_lock);
 510                                 return (dce);
 511                         }
 512                         mutex_exit(&dce->dce_lock);
 513                 }
 514         }
 515         dce = kmem_cache_alloc(dce_cache, KM_NOSLEEP);
 516         if (dce == NULL) {
 517                 rw_exit(&dcb->dcb_lock);
 518                 return (NULL);
 519         }
 520         bzero(dce, sizeof (dce_t));
 521         dce->dce_ipst = ipst;   /* No netstack_hold */
 522         dce->dce_v4addr = dst;
 523         dce->dce_generation = DCE_GENERATION_INITIAL;
 524         dce->dce_ipversion = IPV4_VERSION;
 525         dce->dce_last_change_time = TICK_TO_SEC(ddi_get_lbolt64());
 526         dce_refhold(dce);       /* For the hash list */

 528         /* Link into list */
 529         if (dcb->dcb_dce != NULL)
 530                 dcb->dcb_dce->dce_ptpn = &dce->dce_next;
 531         dce->dce_next = dcb->dcb_dce;
 532         dce->dce_ptpn = &dcb->dcb_dce;
 533         dcb->dcb_dce = dce;
 534         dce->dce_bucket = dcb;
 535         atomic_inc_32(&dcb->dcb_cnt);
 535         atomic_add_32(&dcb->dcb_cnt, 1);
 536         dce_refhold(dce);       /* For the caller */
 537         rw_exit(&dcb->dcb_lock);
```



```
 539         /* Initialize dce_ident to be different than for the last packet */
 540         dce->dce_ident = ipst->ips_dce_default->dce_ident + 1;

 542         dce_increment_generation(ipst->ips_dce_default);
 543         return (dce);
 544 }

 546 /*
 547  * Atomically looks for a non-default DCE, and if not found tries to create one.
 548  * If there is no memory it returns NULL.
 549  * When an entry is created we increase the generation number on
 550  * the default DCE so that conn_ip_output will detect there is a new DCE.
 551  * ifindex should only be used with link-local addresses.
 552  */
 553 dce_t *
 554 dce_lookup_and_add_v6(const in6_addr_t *dst, uint_t ifindex, ip_stack_t *ipst)
 555 {
 556         uint_t          hash;
 557         dcb_t           *dcb;
 558         dce_t           *dce;

 560         /* We should not create entries for link-locals w/o an ifindex */
 561         ASSERT(!(IN6_IS_ADDR_LINKSCOPE(dst)) || ifindex != 0);

 563         hash = IRE_ADDR_HASH_V6(*dst, ipst->ips_dce_hashsize);
 564         dcb = &ipst->ips_dce_hash_v6[hash];
 565         /*
 566          * Assuming that we get fairly even distribution across all of the
 567          * buckets, once one bucket is overly full, prune the whole cache.
 568          */
 569         if (dcb->dcb_cnt > ipst->ips_ip_dce_reclaim_threshold)
 570                 atomic_or_uint(&ipst->ips_dce_reclaim_needed, 1);
 571         rw_enter(&dcb->dcb_lock, RW_WRITER);
 572         for (dce = dcb->dcb_dce; dce != NULL; dce = dce->dce_next) {
 573                 if (IN6_ARE_ADDR_EQUAL(&dce->dce_v6addr, dst) &&
 574                     dce->dce_ifindex == ifindex) {
 575                         mutex_enter(&dce->dce_lock);
 576                         if (!DCE_IS_CONDEMNED(dce)) {
 577                                 dce_refhold(dce);
 578                                 mutex_exit(&dce->dce_lock);
 579                                 rw_exit(&dcb->dcb_lock);
 580                                 return (dce);
 581                         }
 582                         mutex_exit(&dce->dce_lock);
 583                 }
 584         }

 586         dce = kmem_cache_alloc(dce_cache, KM_NOSLEEP);
 587         if (dce == NULL) {
 588                 rw_exit(&dcb->dcb_lock);
 589                 return (NULL);
 590         }
 591         bzero(dce, sizeof (dce_t));
 592         dce->dce_ipst = ipst;   /* No netstack_hold */
 593         dce->dce_v6addr = *dst;
 594         dce->dce_ifindex = ifindex;
 595         dce->dce_generation = DCE_GENERATION_INITIAL;
 596         dce->dce_ipversion = IPV6_VERSION;
 597         dce->dce_last_change_time = TICK_TO_SEC(ddi_get_lbolt64());
 598         dce_refhold(dce);       /* For the hash list */

 600         /* Link into list */
 601         if (dcb->dcb_dce != NULL)
 602                 dcb->dcb_dce->dce_ptpn = &dce->dce_next;
 603         dce->dce_next = dcb->dcb_dce;
 604         dce->dce_ptpn = &dcb->dcb_dce;
```

```
605         dcb->dcb_dce = dce;
606         dce->dce_bucket = dcb;
607         atomic_inc_32(&dcb->dcb_cnt);
607         atomic_add_32(&dcb->dcb_cnt, 1);
608         dce_refhold(dce);       /* For the caller */
609         rw_exit(&dcb->dcb_lock);

611         /* Initialize dce_ident to be different than for the last packet */
612         dce->dce_ident = ipst->ips_dce_default->dce_ident + 1;
613         dce_increment_generation(ipst->ips_dce_default);
614         return (dce);
615 }
______unchanged_portion_omitted_

724 static void
725 dce_make_condemned(dce_t *dce)
726 {
727         ip_stack_t      *ipst = dce->dce_ipst;

729         mutex_enter(&dce->dce_lock);
730         ASSERT(!DCE_IS_CONDEMNED(dce));
731         dce->dce_generation = DCE_GENERATION_CONDEMNED;
732         mutex_exit(&dce->dce_lock);
733         /* Count how many condemned dces for kmem_cache callback */
734         atomic_inc_32(&ipst->ips_num_dce_condemned);
734         atomic_add_32(&ipst->ips_num_dce_condemned, 1);
735 }
______unchanged_portion_omitted_

783 /*
784  * Caller needs to do a dce_refrele since we can't do the
785  * dce_refrele under dcb_lock.
786  */
787 static void
788 dce_delete_locked(dcb_t *dcb, dce_t *dce)
789 {
790         dce->dce_bucket = NULL;
791         *dce->dce_ptpn = dce->dce_next;
792         if (dce->dce_next != NULL)
793                 dce->dce_next->dce_ptpn = dce->dce_ptpn;
794         dce->dce_ptpn = NULL;
795         dce->dce_next = NULL;
796         atomic_dec_32(&dcb->dcb_cnt);
796         atomic_add_32(&dcb->dcb_cnt, -1);
797         dce_make_condemned(dce);
798 }

800 static void
801 dce_inactive(dce_t *dce)
802 {
803         ip_stack_t      *ipst = dce->dce_ipst;

805         ASSERT(!(dce->dce_flags & DCEF_DEFAULT));
806         ASSERT(dce->dce_ptpn == NULL);
807         ASSERT(dce->dce_bucket == NULL);

809         /* Count how many condemned dces for kmem_cache callback */
810         if (DCE_IS_CONDEMNED(dce))
811                 atomic_dec_32(&ipst->ips_num_dce_condemned);
811                 atomic_add_32(&ipst->ips_num_dce_condemned, -1);

813         kmem_cache_free(dce_cache, dce);
814 }

816 void
817 dce_refrele(dce_t *dce)
```



```
818 {
819         ASSERT(dce->dce_refcnt != 0);
820         if (atomic_dec_32_nv(&dce->dce_refcnt) == 0)
820         if (atomic_add_32_nv(&dce->dce_refcnt, -1) == 0)
821                 dce_inactive(dce);
822 }

824 void
825 dce_refhold(dce_t *dce)
826 {
827         atomic_inc_32(&dce->dce_refcnt);
827         atomic_add_32(&dce->dce_refcnt, 1);
828         ASSERT(dce->dce_refcnt != 0);
829 }

831 /* No tracing support yet hence the same as the above functions */
832 void
833 dce_refrele_notr(dce_t *dce)
834 {
835         ASSERT(dce->dce_refcnt != 0);
836         if (atomic_dec_32_nv(&dce->dce_refcnt) == 0)
836         if (atomic_add_32_nv(&dce->dce_refcnt, -1) == 0)
837                 dce_inactive(dce);
838 }

840 void
841 dce_refhold_notr(dce_t *dce)
842 {
843         atomic_inc_32(&dce->dce_refcnt);
843         atomic_add_32(&dce->dce_refcnt, 1);
844         ASSERT(dce->dce_refcnt != 0);
845 }
______unchanged_portion_omitted_
```

****
533313 Mon Jul 28 07:44:25 2014
new/usr/src/uts/common/inet/ip/ip_if.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
11770 /*
11771  * Assign a unique id for the ipif. This is used by sctp_addr.c
11772  * Note: remove if sctp_addr.c is redone to not shadow ill/ipif data structures.
11773  */
11774 static void
11775 ipif_assign_seqid(ipif_t *ipif)
11776 {
11777         ip_stack_t      *ipst = ipif->ipif_ill->ill_ipst;

11779         ipif->ipif_seqid = atomic_inc_64_nv(&ipst->ips_ipif_g_seqid);
11779         ipif->ipif_seqid = atomic_add_64_nv(&ipst->ips_ipif_g_seqid, 1);
11780 }
```

_____***unchanged_portion_omitted_***

```
12442 /*
12443  * Redo source address selection.  This makes IXAF_VERIFY_SOURCE take
12444  * a look again at valid source addresses.
12445  * This should be called each time after the set of source addresses has been
12446  * changed.
12447  */
12448 void
12449 ip_update_source_selection(ip_stack_t *ipst)
12450 {
12451         /* We skip past SRC_GENERATION_VERIFY */
12452         if (atomic_inc_32_nv(&ipst->ips_src_generation) ==
12452         if (atomic_add_32_nv(&ipst->ips_src_generation, 1) ==
12453             SRC_GENERATION_VERIFY)
12454                 atomic_inc_32(&ipst->ips_src_generation);
12454                 atomic_add_32(&ipst->ips_src_generation, 1);
12455 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  102172 Mon Jul 28 07:44:26 2014
new/usr/src/uts/common/inet/ip/ip_ire.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_


 327 /*
 328  * Bump up the reference count on the IRE. We cannot assert that the
 329  * bucket lock is being held as it is legal to bump up the reference
 330  * count after the first lookup has returned the IRE without
 331  * holding the lock.
 332  */
 333 void
 334 ire_refhold(ire_t *ire)
 335 {
 336         atomic_inc_32(&(ire)->ire_refcnt);
 336         atomic_add_32(&(ire)->ire_refcnt, 1);
 337         ASSERT((ire)->ire_refcnt != 0);
 338 #ifdef DEBUG
 339         ire_trace_ref(ire);
 340 #endif
 341 }

 343 void
 344 ire_refhold_notr(ire_t *ire)
 345 {
 346         atomic_inc_32(&(ire)->ire_refcnt);
 346         atomic_add_32(&(ire)->ire_refcnt, 1);
 347         ASSERT((ire)->ire_refcnt != 0);
 348 }
______unchanged_portion_omitted_

 359 /*
 360  * Release a ref on an IRE.
 361  *
 362  * Must not be called while holding any locks. Otherwise if this is
 363  * the last reference to be released there is a chance of recursive mutex
 364  * panic due to ire_refrele -> ipif_ill_refrele_tail -> qwriter_ip trying
 365  * to restart an ioctl. The one exception is when the caller is sure that
 366  * this is not the last reference to be released. Eg. if the caller is
 367  * sure that the ire has not been deleted and won't be deleted.
 368  *
 369  * In architectures e.g sun4u, where atomic_add_32_nv is just
 370  * a cas, we need to maintain the right memory barrier semantics
 371  * as that of mutex_exit i.e all the loads and stores should complete
 372  * before the cas is executed. membar_exit() does that here.
 373  */
 374 void
 375 ire_refrele(ire_t *ire)
 376 {
 377 #ifdef DEBUG
 378         ire_untrace_ref(ire);
 379 #endif
 380         ASSERT((ire)->ire_refcnt != 0);
 381         membar_exit();
 382         if (atomic_dec_32_nv(&(ire)->ire_refcnt) == 0)
 382         if (atomic_add_32_nv(&(ire)->ire_refcnt, -1) == 0)
 383                 ire_inactive(ire);
 384 }

 386 void
 387 ire_refrele_notr(ire_t *ire)
 388 {
 389         ASSERT((ire)->ire_refcnt != 0);
```



```
 390         membar_exit();
 391         if (atomic_dec_32_nv(&(ire)->ire_refcnt) == 0)
 391         if (atomic_add_32_nv(&(ire)->ire_refcnt, -1) == 0)
 392                 ire_inactive(ire);
 393 }
______unchanged_portion_omitted_

1180 /*
1181  * Add a fully initialized IPv4 IRE to the forwarding table.
1182  * This returns NULL on failure, or a held IRE on success.
1183  * Normally the returned IRE is the same as the argument. But a different
1184  * IRE will be returned if the added IRE is deemed identical to an existing
1185  * one. In that case ire_identical_ref will be increased.
1186  * The caller always needs to do an ire_refrele() on the returned IRE.
1187  */
1188 static ire_t *
1189 ire_add_v4(ire_t *ire)
1190 {
1191         ire_t   *ire1;
1192         irb_t   *irb_ptr;
1193         ire_t   **irep;
1194         int     match_flags;
1195         int     error;
1196         ip_stack_t      *ipst = ire->ire_ipst;

1198         if (ire->ire_ill != NULL)
1199                 ASSERT(!MUTEX_HELD(&ire->ire_ill->ill_lock));
1200         ASSERT(ire->ire_ipversion == IPV4_VERSION);

1202         /* Make sure the address is properly masked. */
1203         ire->ire_addr &= ire->ire_mask;

1205         match_flags = (MATCH_IRE_MASK | MATCH_IRE_TYPE | MATCH_IRE_GW);

1207         if (ire->ire_ill != NULL) {
1208                 match_flags |= MATCH_IRE_ILL;
1209         }
1210         irb_ptr = ire_get_bucket(ire);
1211         if (irb_ptr == NULL) {
1212                 printf("no bucket for %p\n", (void *)ire);
1213                 ire_delete(ire);
1214                 return (NULL);
1215         }

1217         /*
1218          * Start the atomic add of the ire. Grab the ill lock,
1219          * the bucket lock. Check for condemned.
1220          */
1221         error = ire_atomic_start(irb_ptr, ire);
1222         if (error != 0) {
1223                 printf("no ire_atomic_start for %p\n", (void *)ire);
1224                 ire_delete(ire);
1225                 irb_refrele(irb_ptr);
1226                 return (NULL);
1227         }
1228         /*
1229          * If we are creating a hidden IRE, make sure we search for
1230          * hidden IREs when searching for duplicates below.
1231          * Otherwise, we might find an IRE on some other interface
1232          * that's not marked hidden.
1233          */
1234         if (ire->ire_testhidden)
1235                 match_flags |= MATCH_IRE_TESTHIDDEN;

1237         /*
1238          * Atomically check for duplicate and insert in the table.
```

```
1239          */
1240         for (ire1 = irb_ptr->irb_ire; ire1 != NULL; ire1 = ire1->ire_next) {
1241                 if (IRE_IS_CONDEMNED(ire1))
1242                         continue;
1243                 /*
1244                  * Here we need an exact match on zoneid, i.e.,
1245                  * ire_match_args doesn't fit.
1246                  */
1247                 if (ire1->ire_zoneid != ire->ire_zoneid)
1248                         continue;

1250                 if (ire1->ire_type != ire->ire_type)
1251                         continue;

1253                 /*
1254                  * Note: We do not allow multiple routes that differ only
1255                  * in the gateway security attributes; such routes are
1256                  * considered duplicates.
1257                  * To change that we explicitly have to treat them as
1258                  * different here.
1259                  */
1260                 if (ire_match_args(ire1, ire->ire_addr, ire->ire_mask,
1261                     ire->ire_gateway_addr, ire->ire_type, ire->ire_ill,
1262                     ire->ire_zoneid, NULL, match_flags)) {
1263                         /*
1264                          * Return the old ire after doing a REFHOLD.
1265                          * As most of the callers continue to use the IRE
1266                          * after adding, we return a held ire. This will
1267                          * avoid a lookup in the caller again. If the callers
1268                          * don't want to use it, they need to do a REFRELE.
1269                          *
1270                          * We only allow exactly one IRE_IF_CLONE for any dst,
1271                          * so, if the is an IF_CLONE, return the ire without
1272                          * an identical_ref, but with an ire_ref held.
1273                          */
1274                         if (ire->ire_type != IRE_IF_CLONE) {
1275                                 atomic_inc_32(&ire1->ire_identical_ref);
1275                                 atomic_add_32(&ire1->ire_identical_ref, 1);
1276                                 DTRACE_PROBE2(ire__add__exist, ire_t *, ire1,
1277                                     ire_t *, ire);
1278                         }
1279                         ire_refhold(ire1);
1280                         ire_atomic_end(irb_ptr, ire);
1281                         ire_delete(ire);
1282                         irb_refrele(irb_ptr);
1283                         return (ire1);
1284                 }
1285         }

1287         /*
1288          * Normally we do head insertion since most things do not care about
1289          * the order of the IREs in the bucket. Note that ip_cgtp_bcast_add
1290          * assumes we at least do head insertion so that its IRE_BROADCAST
1291          * arrive ahead of existing IRE_HOST for the same address.
1292          * However, due to shared-IP zones (and restrict_interzone_loopback)
1293          * we can have an IRE_LOCAL as well as IRE_IF_CLONE for the same
1294          * address. For that reason we do tail insertion for IRE_IF_CLONE.
1295          * Due to the IRE_BROADCAST on cgtp0, which must be last in the bucket,
1296          * we do tail insertion of IRE_BROADCASTs that do not have RTF_MULTIRT
1297          * set.
1298          */
1299         irep = (ire_t **)irb_ptr;
1300         if ((ire->ire_type & IRE_IF_CLONE) ||
1301             ((ire->ire_type & IRE_BROADCAST) &&
1302             !(ire->ire_flags & RTF_MULTIRT))) {
1303                 while ((ire1 = *irep) != NULL)
```



```
1304                         irep = &ire1->ire_next;
1305         }
1306         /* Insert at *irep */
1307         ire1 = *irep;
1308         if (ire1 != NULL)
1309                 ire1->ire_ptpn = &ire->ire_next;
1310         ire->ire_next = ire1;
1311         /* Link the new one in. */
1312         ire->ire_ptpn = irep;

1314         /*
1315          * ire_walk routines de-reference ire_next without holding
1316          * a lock. Before we point to the new ire, we want to make
1317          * sure the store that sets the ire_next of the new ire
1318          * reaches global visibility, so that ire_walk routines
1319          * don't see a truncated list of ires i.e if the ire_next
1320          * of the new ire gets set after we do "*irep = ire" due
1321          * to re-ordering, the ire_walk thread will see a NULL
1322          * once it accesses the ire_next of the new ire.
1323          * membar_producer() makes sure that the following store
1324          * happens *after* all of the above stores.
1325          */
1326         membar_producer();
1327         *irep = ire;
1328         ire->ire_bucket = irb_ptr;
1329         /*
1330          * We return a bumped up IRE above. Keep it symmetrical
1331          * so that the callers will always have to release. This
1332          * helps the callers of this function because they continue
1333          * to use the IRE after adding and hence they don't have to
1334          * lookup again after we return the IRE.
1335          *
1336          * NOTE : We don't have to use atomics as this is appearing
1337          * in the list for the first time and no one else can bump
1338          * up the reference count on this yet.
1339          */
1340         ire_refhold_locked(ire);
1341         BUMP_IRE_STATS(ipst->ips_ire_stats_v4, ire_stats_inserted);

1343         irb_ptr->irb_ire_cnt++;
1344         if (irb_ptr->irb_marks & IRB_MARK_DYNAMIC)
1345                 irb_ptr->irb_nire++;

1347         if (ire->ire_ill != NULL) {
1348                 ire->ire_ill->ill_ire_cnt++;
1349                 ASSERT(ire->ire_ill->ill_ire_cnt != 0); /* Wraparound */
1350         }

1352         ire_atomic_end(irb_ptr, ire);

1354         /* Make any caching of the IREs be notified or updated */
1355         ire_flush_cache_v4(ire, IRE_FLUSH_ADD);

1357         if (ire->ire_ill != NULL)
1358                 ASSERT(!MUTEX_HELD(&ire->ire_ill->ill_lock));
1359         irb_refrele(irb_ptr);
1360         return (ire);
1361 }
```

unchanged_portion_omitted_

```
1481 /*
1482  * Delete the specified IRE.
1483  * We assume that if ire_bucket is not set then ire_ill->ill_ire_cnt was
1484  * not incremented i.e., that the insertion in the bucket and the increment
1485  * of that counter is done atomically.
1486  */
```

```
1487 void
1488 ire_delete(ire_t *ire)
1489 {
1490         ire_t   *ire1;
1491         ire_t   **ptpn;
1492         irb_t   *irb;
1493         ip_stack_t      *ipst = ire->ire_ipst;

1495         if ((irb = ire->ire_bucket) == NULL) {
1496                 /*
1497                  * It was never inserted in the list. Should call REFRELE
1498                  * to free this IRE.
1499                  */
1500                 ire_make_condemned(ire);
1501                 ire_refrele_notr(ire);
1502                 return;
1503         }

1505         /*
1506          * Move the use counts from an IRE_IF_CLONE to its parent
1507          * IRE_INTERFACE.
1508          * We need to do this before acquiring irb_lock.
1509          */
1510         if (ire->ire_type & IRE_IF_CLONE) {
1511                 ire_t *parent;

1513                 rw_enter(&ipst->ips_ire_dep_lock, RW_READER);
1514                 if ((parent = ire->ire_dep_parent) != NULL) {
1515                         parent->ire_ob_pkt_count += ire->ire_ob_pkt_count;
1516                         parent->ire_ib_pkt_count += ire->ire_ib_pkt_count;
1517                         ire->ire_ob_pkt_count = 0;
1518                         ire->ire_ib_pkt_count = 0;
1519                 }
1520                 rw_exit(&ipst->ips_ire_dep_lock);
1521         }

1523         rw_enter(&irb->irb_lock, RW_WRITER);
1524         if (ire->ire_ptpn == NULL) {
1525                 /*
1526                  * Some other thread has removed us from the list.
1527                  * It should have done the REFRELE for us.
1528                  */
1529                 rw_exit(&irb->irb_lock);
1530                 return;
1531         }

1533         if (!IRE_IS_CONDEMNED(ire)) {
1534                 /* Is this an IRE representing multiple duplicate entries? */
1535                 ASSERT(ire->ire_identical_ref >= 1);
1536                 if (atomic_dec_32_nv(&ire->ire_identical_ref) != 0) {
1536                 if (atomic_add_32_nv(&ire->ire_identical_ref, -1) != 0) {
1537                         /* Removed one of the identical parties */
1538                         rw_exit(&irb->irb_lock);
1539                         return;
1540                 }

1542                 irb->irb_ire_cnt--;
1543                 ire_make_condemned(ire);
1544         }

1546         if (irb->irb_refcnt != 0) {
1547                 /*
1548                  * The last thread to leave this bucket will
1549                  * delete this ire.
1550                  */
1551                 irb->irb_marks |= IRB_MARK_CONDEMNED;
```



```
1552                 rw_exit(&irb->irb_lock);
1553                 return;
1554         }

1556         /*
1557          * Normally to delete an ire, we walk the bucket. While we
1558          * walk the bucket, we normally bump up irb_refcnt and hence
1559          * we return from above where we mark CONDEMNED and the ire
1560          * gets deleted from ire_unlink. This case is where somebody
1561          * knows the ire e.g by doing a lookup, and wants to delete the
1562          * IRE. irb_refcnt would be 0 in this case if nobody is walking
1563          * the bucket.
1564          */
1565         ptpn = ire->ire_ptpn;
1566         ire1 = ire->ire_next;
1567         if (ire1 != NULL)
1568                 ire1->ire_ptpn = ptpn;
1569         ASSERT(ptpn != NULL);
1570         *ptpn = ire1;
1571         ire->ire_ptpn = NULL;
1572         ire->ire_next = NULL;
1573         if (ire->ire_ipversion == IPV6_VERSION) {
1574                 BUMP_IRE_STATS(ipst->ips_ire_stats_v6, ire_stats_deleted);
1575         } else {
1576                 BUMP_IRE_STATS(ipst->ips_ire_stats_v4, ire_stats_deleted);
1577         }
1578         rw_exit(&irb->irb_lock);

1580         /* Cleanup dependents and related stuff */
1581         if (ire->ire_ipversion == IPV6_VERSION) {
1582                 ire_delete_v6(ire);
1583         } else {
1584                 ire_delete_v4(ire);
1585         }
1586         /*
1587          * We removed it from the list. Decrement the
1588          * reference count.
1589          */
1590         ire_refrele_notr(ire);
1591 }
______unchanged_portion_omitted_

2604 /*
2605  * The caller should hold irb_lock as a writer if the ire is in a bucket.
2606  * This routine will clear ire_nce_cache, and we make sure that we can never
2607  * set ire_nce_cache after the ire is marked condemned.
2608  */
2609 void
2610 ire_make_condemned(ire_t *ire)
2611 {
2612         ip_stack_t      *ipst = ire->ire_ipst;
2613         nce_t           *nce;

2615         mutex_enter(&ire->ire_lock);
2616         ASSERT(ire->ire_bucket == NULL ||
2617             RW_WRITE_HELD(&ire->ire_bucket->irb_lock));
2618         ASSERT(!IRE_IS_CONDEMNED(ire));
2619         ire->ire_generation = IRE_GENERATION_CONDEMNED;
2620         /* Count how many condemned ires for kmem_cache callback */
2621         atomic_inc_32(&ipst->ips_num_ire_condemned);
2621         atomic_add_32(&ipst->ips_num_ire_condemned, 1);
2622         nce = ire->ire_nce_cache;
2623         ire->ire_nce_cache = NULL;
2624         mutex_exit(&ire->ire_lock);
2625         if (nce != NULL)
2626                 nce_refrele(nce);
```

```
2627 }
______unchanged_portion_omitted_
```

****
138909 Mon Jul 28 07:44:26 2014
new/usr/src/uts/common/inet/ip/ip_ndp.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
 443 /*
 444  * 1. Mark the entry CONDEMNED. This ensures that no new nce_lookup()
 445  *    will return this NCE. Also no new timeouts will
 446  *    be started (See nce_restart_timer).
 447  * 2. Cancel any currently running timeouts.
 448  * 3. If there is an ndp walker, return. The walker will do the cleanup.
 449  *    This ensures that walkers see a consistent list of NCEs while walking.
 450  * 4. Otherwise remove the NCE from the list of NCEs
 451  */
 452 void
 453 ncec_delete(ncec_t *ncec)
 454 {
 455         ncec_t  **ptpn;
 456         ncec_t  *ncec1;
 457         int     ipversion = ncec->ncec_ipversion;
 458         ndp_g_t *ndp;
 459         ip_stack_t      *ipst = ncec->ncec_ipst;

 461         if (ipversion == IPV4_VERSION)
 462                 ndp = ipst->ips_ndp4;
 463         else
 464                 ndp = ipst->ips_ndp6;

 466         /* Serialize deletes */
 467         mutex_enter(&ncec->ncec_lock);
 468         if (NCE_ISCONDEMNED(ncec)) {
 469                 /* Some other thread is doing the delete */
 470                 mutex_exit(&ncec->ncec_lock);
 471                 return;
 472         }
 473         /*
 474          * Caller has a refhold. Also 1 ref for being in the list. Thus
 475          * refcnt has to be >= 2
 476          */
 477         ASSERT(ncec->ncec_refcnt >= 2);
 478         ncec->ncec_flags |= NCE_F_CONDEMNED;
 479         mutex_exit(&ncec->ncec_lock);

 481         /* Count how many condemned ires for kmem_cache callback */
 482         atomic_inc_32(&ipst->ips_num_nce_condemned);
 482         atomic_add_32(&ipst->ips_num_nce_condemned, 1);
 483         nce_fastpath_list_delete(ncec->ncec_ill, ncec, NULL);

 485         /* Complete any waiting callbacks */
 486         ncec_cb_dispatch(ncec);

 488         /*
 489          * Cancel any running timer. Timeout can't be restarted
 490          * since CONDEMNED is set. Can't hold ncec_lock across untimeout.
 491          * Passing invalid timeout id is fine.
 492          */
 493         if (ncec->ncec_timeout_id != 0) {
 494                 (void) untimeout(ncec->ncec_timeout_id);
 495                 ncec->ncec_timeout_id = 0;
 496         }

 498         mutex_enter(&ndp->ndp_g_lock);
 499         if (ncec->ncec_ptpn == NULL) {
 500                 /*
```



```
 501                  * The last ndp walker has already removed this ncec from
 502                  * the list after we marked the ncec CONDEMNED and before
 503                  * we grabbed the global lock.
 504                  */
 505                 mutex_exit(&ndp->ndp_g_lock);
 506                 return;
 507         }
 508         if (ndp->ndp_g_walker > 0) {
 509                 /*
 510                  * Can't unlink. The walker will clean up
 511                  */
 512                 ndp->ndp_g_walker_cleanup = B_TRUE;
 513                 mutex_exit(&ndp->ndp_g_lock);
 514                 return;
 515         }

 517         /*
 518          * Now remove the ncec from the list. nce_restart_timer won't restart
 519          * the timer since it is marked CONDEMNED.
 520          */
 521         ptpn = ncec->ncec_ptpn;
 522         ncec1 = ncec->ncec_next;
 523         if (ncec1 != NULL)
 524                 ncec1->ncec_ptpn = ptpn;
 525         *ptpn = ncec1;
 526         ncec->ncec_ptpn = NULL;
 527         ncec->ncec_next = NULL;
 528         mutex_exit(&ndp->ndp_g_lock);

 530         /* Removed from ncec_ptpn/ncec_next list */
 531         ncec_refrele_notr(ncec);
 532 }
```

_____***unchanged_portion_omitted_***

****
74005 Mon Jul 28 07:44:27 2014
new/usr/src/uts/common/inet/ip/ip_output.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____unchanged_portion_omitted_

```
1547 /*
1548  * ire_sendfn for IREs with RTF_REJECT/RTF_BLACKHOLE, including IRE_NOROUTE
1549  */
1550 int
1551 ire_send_noroute_v4(ire_t *ire, mblk_t *mp, void *iph_arg,
1552     ip_xmit_attr_t *ixa, uint32_t *identp)
1553 {
1554         ip_stack_t      *ipst = ixa->ixa_ipst;
1555         ipha_t          *ipha = (ipha_t *)iph_arg;
1556         ill_t           *ill;
1557         ip_recv_attr_t  iras;
1558         boolean_t       dummy;

1560         /* We assign an IP ident for nice errors */
1561         ipha->ipha_ident = atomic_inc_32_nv(identp);
1561         ipha->ipha_ident = atomic_add_32_nv(identp, 1);

1563         BUMP_MIB(&ipst->ips_ip_mib, ipIfStatsOutNoRoutes);

1565         if (ire->ire_type & IRE_NOROUTE) {
1566                 /* A lack of a route as opposed to RTF_REJECT|BLACKHOLE */
1567                 ip_rts_change(RTM_MISS, ipha->ipha_dst, 0, 0, 0, 0, 0, 0,
1568                     RTA_DST, ipst);
1569         }

1571         if (ire->ire_flags & RTF_BLACKHOLE) {
1572                 ip_drop_output("ipIfStatsOutNoRoutes RTF_BLACKHOLE", mp, NULL);
1573                 freemsg(mp);
1574                 /* No error even for local senders - silent blackhole */
1575                 return (0);
1576         }
1577         ip_drop_output("ipIfStatsOutNoRoutes RTF_REJECT", mp, NULL);

1579         /*
1580          * We need an ill_t for the ip_recv_attr_t even though this packet
1581          * was never received and icmp_unreachable doesn't currently use
1582          * ira_ill.
1583          */
1584         ill = ill_lookup_on_name("lo0", B_FALSE,
1585             !(ixa->ixa_flags & IRAF_IS_IPV4), &dummy, ipst);
1586         if (ill == NULL) {
1587                 freemsg(mp);
1588                 return (EHOSTUNREACH);
1589         }

1591         bzero(&iras, sizeof (iras));
1592         /* Map ixa to ira including IPsec policies */
1593         ipsec_out_to_in(ixa, ill, &iras);

1595         if (ip_source_routed(ipha, ipst)) {
1596                 icmp_unreachable(mp, ICMP_SOURCE_ROUTE_FAILED, &iras);
1597         } else {
1598                 icmp_unreachable(mp, ICMP_HOST_UNREACHABLE, &iras);
1599         }
1600         /* We moved any IPsec refs from ixa to iras */
1601         ira_cleanup(&iras, B_FALSE);
1602         ill_refrele(ill);
1603         return (EHOSTUNREACH);
1604 }
```

_____unchanged_portion_omitted_

```
**********************************************************
  112643 Mon Jul 28 07:44:27 2014
new/usr/src/uts/common/inet/ip/ipsecah.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

2547 static boolean_t
2548 ah_finish_up(ah_t *phdr_ah, ah_t *inbound_ah, ipsa_t *assoc,
2549     int ah_data_sz, int ah_align_sz, ipsecah_stack_t *ahstack)
2550 {
2551         int i;

2553         /*
2554          * Padding :
2555          *
2556          * 1) Authentication data may have to be padded
2557          * before ICV calculation if ICV is not a multiple
2558          * of 64 bits. This padding is arbitrary and transmitted
2559          * with the packet at the end of the authentication data.
2560          * Payload length should include the padding bytes.
2561          *
2562          * 2) Explicit padding of the whole datagram may be
2563          * required by the algorithm which need not be
2564          * transmitted. It is assumed that this will be taken
2565          * care by the algorithm module.
2566          */
2567         bzero(phdr_ah + 1, ah_data_sz); /* Zero out ICV for pseudo-hdr. */

2569         if (inbound_ah == NULL) {
2570                 /* Outbound AH datagram. */

2572                 phdr_ah->ah_length = (ah_align_sz >> 2) + 1;
2573                 phdr_ah->ah_reserved = 0;
2574                 phdr_ah->ah_spi = assoc->ipsa_spi;

2576                 phdr_ah->ah_replay =
2577                     htonl(atomic_inc_32_nv(&assoc->ipsa_replay));
2577                     htonl(atomic_add_32_nv(&assoc->ipsa_replay, 1));
2578                 if (phdr_ah->ah_replay == 0 && assoc->ipsa_replay_wsize != 0) {
2579                         /*
2580                          * XXX We have replay counter wrapping.  We probably
2581                          * want to nuke this SA (and its peer).
2582                          */
2583                         ipsec_assocfailure(info.mi_idnum, 0, 0,
2584                             SL_ERROR | SL_CONSOLE | SL_WARN,
2585                             "Outbound AH SA (0x%x), dst %s has wrapped "
2586                             "sequence.\n", phdr_ah->ah_spi,
2587                             assoc->ipsa_dstaddr, assoc->ipsa_addrfam,
2588                             ahstack->ipsecah_netstack);

2590                         sadb_replay_delete(assoc);
2591                         /* Caller will free phdr_mp and return NULL. */
2592                         return (B_FALSE);
2593                 }

2595                 if (ah_data_sz != ah_align_sz) {
2596                         uchar_t *pad = ((uchar_t *)phdr_ah + sizeof (ah_t) +
2597                             ah_data_sz);

2599                         for (i = 0; i < (ah_align_sz - ah_data_sz); i++) {
2600                                 pad[i] = (uchar_t)i;    /* Fill the padding */
2601                         }
2602                 }
2603         } else {
2604                 /* Inbound AH datagram. */
```



```
2605                 phdr_ah->ah_nexthdr = inbound_ah->ah_nexthdr;
2606                 phdr_ah->ah_length = inbound_ah->ah_length;
2607                 phdr_ah->ah_reserved = 0;
2608                 ASSERT(inbound_ah->ah_spi == assoc->ipsa_spi);
2609                 phdr_ah->ah_spi = inbound_ah->ah_spi;
2610                 phdr_ah->ah_replay = inbound_ah->ah_replay;

2612                 if (ah_data_sz != ah_align_sz) {
2613                         uchar_t *opad = ((uchar_t *)inbound_ah +
2614                             sizeof (ah_t) + ah_data_sz);
2615                         uchar_t *pad = ((uchar_t *)phdr_ah + sizeof (ah_t) +
2616                             ah_data_sz);

2618                         for (i = 0; i < (ah_align_sz - ah_data_sz); i++) {
2619                                 pad[i] = opad[i];       /* Copy the padding */
2620                         }
2621                 }
2622         }

2624         return (B_TRUE);
2625 }
______unchanged_portion_omitted_
```

```
**********************************************************
  121360 Mon Jul 28 07:44:27 2014
new/usr/src/uts/common/inet/ip/ipsecesp.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

2599 /*
2600  * Handle outbound IPsec processing for IPv4 and IPv6
2601  *
2602  * Returns data_mp if successfully completed the request. Returns
2603  * NULL if it failed (and increments InDiscards) or if it is pending.
2604  */
2605 static mblk_t *
2606 esp_outbound(mblk_t *data_mp, ip_xmit_attr_t *ixa)
2607 {
2608         mblk_t *espmp, *tailmp;
2609         ipha_t *ipha;
2610         ip6_t *ip6h;
2611         esph_t *esph_ptr, *iv_ptr;
2612         uint_t af;
2613         uint8_t *nhp;
2614         uintptr_t divpoint, datalen, adj, padlen, i, alloclen;
2615         uintptr_t esplen = sizeof (esph_t);
2616         uint8_t protocol;
2617         ipsa_t *assoc;
2618         uint_t iv_len, block_size, mac_len = 0;
2619         uchar_t *icv_buf;
2620         udpha_t *udpha;
2621         boolean_t is_natt = B_FALSE;
2622         netstack_t      *ns = ixa->ixa_ipst->ips_netstack;
2623         ipsecesp_stack_t *espstack = ns->netstack_ipsecesp;
2624         ipsec_stack_t   *ipss = ns->netstack_ipsec;
2625         ill_t           *ill = ixa->ixa_nce->nce_ill;
2626         boolean_t       need_refrele = B_FALSE;

2628         ESP_BUMP_STAT(espstack, out_requests);

2630         /*
2631          * <sigh> We have to copy the message here, because TCP (for example)
2632          * keeps a dupb() of the message lying around for retransmission.
2633          * Since ESP changes the whole of the datagram, we have to create our
2634          * own copy lest we clobber TCP's data.  Since we have to copy anyway,
2635          * we might as well make use of msgpullup() and get the mblk into one
2636          * contiguous piece!
2637          */
2638         tailmp = msgpullup(data_mp, -1);
2639         if (tailmp == NULL) {
2640                 esp0dbg(("esp_outbound: msgpullup() failed, "
2641                     "dropping packet.\n"));
2642                 ip_drop_packet(data_mp, B_FALSE, ill,
2643                     DROPPER(ipss, ipds_esp_nomem),
2644                     &espstack->esp_dropper);
2645                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
2646                 return (NULL);
2647         }
2648         freemsg(data_mp);
2649         data_mp = tailmp;

2651         assoc = ixa->ixa_ipsec_esp_sa;
2652         ASSERT(assoc != NULL);

2654         /*
2655          * Get the outer IP header in shape to escape this system..
2656          */
2657         if (is_system_labeled() && (assoc->ipsa_otsl != NULL)) {
```



```
2658                 /*
2659                  * Need to update packet with any CIPSO option and update
2660                  * ixa_tsl to capture the new label.
2661                  * We allocate a separate ixa for that purpose.
2662                  */
2663                 ixa = ip_xmit_attr_duplicate(ixa);
2664                 if (ixa == NULL) {
2665                         ip_drop_packet(data_mp, B_FALSE, ill,
2666                             DROPPER(ipss, ipds_esp_nomem),
2667                             &espstack->esp_dropper);
2668                         return (NULL);
2669                 }
2670                 need_refrele = B_TRUE;

2672                 label_hold(assoc->ipsa_otsl);
2673                 ip_xmit_attr_replace_tsl(ixa, assoc->ipsa_otsl);

2675                 data_mp = sadb_whack_label(data_mp, assoc, ixa,
2676                     DROPPER(ipss, ipds_esp_nomem), &espstack->esp_dropper);
2677                 if (data_mp == NULL) {
2678                         /* Packet dropped by sadb_whack_label */
2679                         ixa_refrele(ixa);
2680                         return (NULL);
2681                 }
2682         }

2684         /*
2685          * Reality check....
2686          */
2687         ipha = (ipha_t *)data_mp->b_rptr;  /* So we can call esp_acquire(). */

2689         if (ixa->ixa_flags & IXAF_IS_IPV4) {
2690                 ASSERT(IPH_HDR_VERSION(ipha) == IPV4_VERSION);

2692                 af = AF_INET;
2693                 divpoint = IPH_HDR_LENGTH(ipha);
2694                 datalen = ntohs(ipha->ipha_length) - divpoint;
2695                 nhp = (uint8_t *)&ipha->ipha_protocol;
2696         } else {
2697                 ip_pkt_t ipp;

2699                 ASSERT(IPH_HDR_VERSION(ipha) == IPV6_VERSION);

2701                 af = AF_INET6;
2702                 ip6h = (ip6_t *)ipha;
2703                 bzero(&ipp, sizeof (ipp));
2704                 divpoint = ip_find_hdr_v6(data_mp, ip6h, B_FALSE, &ipp, NULL);
2705                 if (ipp.ipp_dstopts != NULL &&
2706                     ipp.ipp_dstopts->ip6d_nxt != IPPROTO_ROUTING) {
2707                         /*
2708                          * Destination options are tricky.  If we get in here,
2709                          * then we have a terminal header following the
2710                          * destination options.  We need to adjust backwards
2711                          * so we insert ESP BEFORE the destination options
2712                          * bag.  (So that the dstopts get encrypted!)
2713                          *
2714                          * Since this is for outbound packets only, we know
2715                          * that non-terminal destination options only precede
2716                          * routing headers.
2717                          */
2718                         divpoint -= ipp.ipp_dstoptslen;
2719                 }
2720                 datalen = ntohs(ip6h->ip6_plen) + sizeof (ip6_t) - divpoint;

2722                 if (ipp.ipp_rthdr != NULL) {
2723                         nhp = &ipp.ipp_rthdr->ip6r_nxt;
```

```
2724                } else if (ipp.ipp_hopopts != NULL) {
2725                        nhp = &ipp.ipp_hopopts->ip6h_nxt;
2726                } else {
2727                        ASSERT(divpoint == sizeof (ip6_t));
2728                        /* It's probably IP + ESP. */
2729                        nhp = &ip6h->ip6_nxt;
2730                }
2731        }

2733        mac_len = assoc->ipsa_mac_len;

2735        if (assoc->ipsa_flags & IPSA_F_NATT) {
2736                /* wedge in UDP header */
2737                is_natt = B_TRUE;
2738                esplen += UDPH_SIZE;
2739        }

2741        /*
2742         * Set up ESP header and encryption padding for ENCR PI request.
2743         */

2745        /* Determine the padding length.  Pad to 4-bytes for no-encryption. */
2746        if (assoc->ipsa_encr_alg != SADB_EALG_NULL) {
2747                iv_len = assoc->ipsa_iv_len;
2748                block_size = assoc->ipsa_datalen;

2750                /*
2751                 * Pad the data to the length of the cipher block size.
2752                 * Include the two additional bytes (hence the - 2) for the
2753                 * padding length and the next header.  Take this into account
2754                 * when calculating the actual length of the padding.
2755                 */
2756                ASSERT(ISP2(iv_len));
2757                padlen = ((unsigned)(block_size - datalen - 2)) &
2758                    (block_size - 1);
2759        } else {
2760                iv_len = 0;
2761                padlen = ((unsigned)(sizeof (uint32_t) - datalen - 2)) &
2762                    (sizeof (uint32_t) - 1);
2763        }

2765        /* Allocate ESP header and IV. */
2766        esplen += iv_len;

2768        /*
2769         * Update association byte-count lifetimes.  Don't forget to take
2770         * into account the padding length and next-header (hence the + 2).
2771         *
2772         * Use the amount of data fed into the "encryption algorithm".  This
2773         * is the IV, the data length, the padding length, and the final two
2774         * bytes (padlen, and next-header).
2775         *
2776         */

2778        if (!esp_age_bytes(assoc, datalen + padlen + iv_len + 2, B_FALSE)) {
2779                ip_drop_packet(data_mp, B_FALSE, ill,
2780                    DROPPER(ipss, ipds_esp_bytes_expire),
2781                    &espstack->esp_dropper);
2782                BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
2783                if (need_refrele)
2784                        ixa_refrele(ixa);
2785                return (NULL);
2786        }

2788        espmp = allocb(esplen, BPRI_HI);
2789        if (espmp == NULL) {
```



```
2790                ESP_BUMP_STAT(espstack, out_discards);
2791                esp1dbg(espstack, ("esp_outbound: can't allocate espmp.\n"));
2792                ip_drop_packet(data_mp, B_FALSE, ill,
2793                    DROPPER(ipss, ipds_esp_nomem),
2794                    &espstack->esp_dropper);
2795                BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
2796                if (need_refrele)
2797                        ixa_refrele(ixa);
2798                return (NULL);
2799        }
2800        espmp->b_wptr += esplen;
2801        esph_ptr = (esph_t *)espmp->b_rptr;

2803        if (is_natt) {
2804                esp3dbg(espstack, ("esp_outbound: NATT"));

2806                udpha = (udpha_t *)espmp->b_rptr;
2807                udpha->uha_src_port = (assoc->ipsa_local_nat_port != 0) ?
2808                    assoc->ipsa_local_nat_port : htons(IPPORT_IKE_NATT);
2809                udpha->uha_dst_port = (assoc->ipsa_remote_nat_port != 0) ?
2810                    assoc->ipsa_remote_nat_port : htons(IPPORT_IKE_NATT);
2811                /*
2812                 * Set the checksum to 0, so that the esp_prepare_udp() call
2813                 * can do the right thing.
2814                 */
2815                udpha->uha_checksum = 0;
2816                esph_ptr = (esph_t *)(udpha + 1);
2817        }

2819        esph_ptr->esph_spi = assoc->ipsa_spi;

2821        esph_ptr->esph_replay = htonl(atomic_inc_32_nv(&assoc->ipsa_replay));
2821        esph_ptr->esph_replay = htonl(atomic_add_32_nv(&assoc->ipsa_replay, 1));
2822        if (esph_ptr->esph_replay == 0 && assoc->ipsa_replay_wsize != 0) {
2823                /*
2824                 * XXX We have replay counter wrapping.
2825                 * We probably want to nuke this SA (and its peer).
2826                 */
2827                ipsec_assocfailure(info.mi_idnum, 0, 0,
2828                    SL_ERROR | SL_CONSOLE | SL_WARN,
2829                    "Outbound ESP SA (0x%x, %s) has wrapped sequence.\n",
2830                    esph_ptr->esph_spi, assoc->ipsa_dstaddr, af,
2831                    espstack->ipsecesp_netstack);

2833                ESP_BUMP_STAT(espstack, out_discards);
2834                sadb_replay_delete(assoc);
2835                ip_drop_packet(data_mp, B_FALSE, ill,
2836                    DROPPER(ipss, ipds_esp_replay),
2837                    &espstack->esp_dropper);
2838                BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
2839                if (need_refrele)
2840                        ixa_refrele(ixa);
2841                return (NULL);
2842        }

2844        iv_ptr = (esph_ptr + 1);
2845        /*
2846         * iv_ptr points to the mblk which will contain the IV once we have
2847         * written it there. This mblk will be part of a mblk chain that
2848         * will make up the packet.
2849         *
2850         * For counter mode algorithms, the IV is a 64 bit quantity, it
2851         * must NEVER repeat in the lifetime of the SA, otherwise an
2852         * attacker who had recorded enough packets might be able to
2853         * determine some clear text.
2854         *
```

```
2855          * To ensure this does not happen, the IV is stored in the SA and
2856          * incremented for each packet, the IV is then copied into the
2857          * "packet" for transmission to the receiving system. The IV will
2858          * also be copied into the nonce, when the packet is encrypted.
2859          *
2860          * CBC mode algorithms use a random IV for each packet. We do not
2861          * require the highest quality random bits, but for best security
2862          * with CBC mode ciphers, the value must be unlikely to repeat and
2863          * must not be known in advance to an adversary capable of influencing
2864          * the clear text.
2865          */
2866         if (!update_iv((uint8_t *)iv_ptr, espstack->esp_pfkey_q, assoc,
2867             espstack)) {
2868                 ip_drop_packet(data_mp, B_FALSE, ill,
2869                     DROPPER(ipss, ipds_esp_iv_wrap), &espstack->esp_dropper);
2870                 if (need_refrele)
2871                         ixa_refrele(ixa);
2872                 return (NULL);
2873         }

2875         /* Fix the IP header. */
2876         alloclen = padlen + 2 + mac_len;
2877         adj = alloclen + (espmp->b_wptr - espmp->b_rptr);

2879         protocol = *nhp;

2881         if (ixa->ixa_flags & IXAF_IS_IPV4) {
2882                 ipha->ipha_length = htons(ntohs(ipha->ipha_length) + adj);
2883                 if (is_natt) {
2884                         *nhp = IPPROTO_UDP;
2885                         udpha->uha_length = htons(ntohs(ipha->ipha_length) -
2886                             IPH_HDR_LENGTH(ipha));
2887                 } else {
2888                         *nhp = IPPROTO_ESP;
2889                 }
2890                 ipha->ipha_hdr_checksum = 0;
2891                 ipha->ipha_hdr_checksum = (uint16_t)ip_csum_hdr(ipha);
2892         } else {
2893                 ip6h->ip6_plen = htons(ntohs(ip6h->ip6_plen) + adj);
2894                 *nhp = IPPROTO_ESP;
2895         }

2897         /* I've got the two ESP mblks, now insert them. */

2899         esp2dbg(espstack, ("data_mp before outbound ESP adjustment:\n"));
2900         esp2dbg(espstack, (dump_msg(data_mp)));

2902         if (!esp_insert_esp(data_mp, espmp, divpoint, espstack)) {
2903                 ESP_BUMP_STAT(espstack, out_discards);
2904                 /* NOTE:  esp_insert_esp() only fails if there's no memory. */
2905                 ip_drop_packet(data_mp, B_FALSE, ill,
2906                     DROPPER(ipss, ipds_esp_nomem),
2907                     &espstack->esp_dropper);
2908                 freeb(espmp);
2909                 BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
2910                 if (need_refrele)
2911                         ixa_refrele(ixa);
2912                 return (NULL);
2913         }

2915         /* Append padding (and leave room for ICV). */
2916         for (tailmp = data_mp; tailmp->b_cont != NULL; tailmp = tailmp->b_cont)
2917                 ;
2918         if (tailmp->b_wptr + alloclen > tailmp->b_datap->db_lim) {
2919                 tailmp->b_cont = allocb(alloclen, BPRI_HI);
2920                 if (tailmp->b_cont == NULL) {
```



```
2921                         ESP_BUMP_STAT(espstack, out_discards);
2922                         esp0dbg(("esp_outbound:  Can't allocate tailmp.\n"));
2923                         ip_drop_packet(data_mp, B_FALSE, ill,
2924                             DROPPER(ipss, ipds_esp_nomem),
2925                             &espstack->esp_dropper);
2926                         BUMP_MIB(ill->ill_ip_mib, ipIfStatsOutDiscards);
2927                         if (need_refrele)
2928                                 ixa_refrele(ixa);
2929                         return (NULL);
2930                 }
2931                 tailmp = tailmp->b_cont;
2932         }

2934         /*
2935          * If there's padding, N bytes of padding must be of the form 0x1,
2936          * 0x2, 0x3... 0xN.
2937          */
2938         for (i = 0; i < padlen; ) {
2939                 i++;
2940                 *tailmp->b_wptr++ = i;
2941         }
2942         *tailmp->b_wptr++ = i;
2943         *tailmp->b_wptr++ = protocol;

2945         esp2dbg(espstack, ("data_Mp before encryption:\n"));
2946         esp2dbg(espstack, (dump_msg(data_mp)));

2948         /*
2949          * Okay.  I've set up the pre-encryption ESP.  Let's do it!
2950          */

2952         if (mac_len > 0) {
2953                 ASSERT(tailmp->b_wptr + mac_len <= tailmp->b_datap->db_lim);
2954                 icv_buf = tailmp->b_wptr;
2955                 tailmp->b_wptr += mac_len;
2956         } else {
2957                 icv_buf = NULL;
2958         }

2960         data_mp = esp_submit_req_outbound(data_mp, ixa, assoc, icv_buf,
2961             datalen + padlen + 2);
2962         if (need_refrele)
2963                 ixa_refrele(ixa);
2964         return (data_mp);
2965 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   64237 Mon Jul 28 07:44:28 2014
new/usr/src/uts/common/inet/ip/keysock.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 477 /*
 478  * Close routine for keysock.
 479  */
 480 static int
 481 keysock_close(queue_t *q)
 482 {
 483         keysock_t *ks;
 484         keysock_consumer_t *kc;
 485         void *ptr = q->q_ptr;
 486         int size;
 487         keysock_stack_t *keystack;


 490         qprocsoff(q);

 492         /* Safe assumption. */
 493         ASSERT(ptr != NULL);

 495         if (WR(q)->q_next) {
 496                 kc = (keysock_consumer_t *)ptr;
 497                 keystack = kc->kc_keystack;

 499                 ks1dbg(keystack, ("Module close, removing a consumer (%d).\n",
 500                     kc->kc_sa_type));
 501                 /*
 502                  * Because of PERMOD open/close exclusive perimeter, I
 503                  * can inspect KC_FLUSHING w/o locking down kc->kc_lock.
 504                  */
 505                 if (kc->kc_flags & KC_FLUSHING) {
 506                         /*
 507                          * If this decrement was the last one, send
 508                          * down the next pending one, if any.
 509                          *
 510                          * With a PERMOD perimeter, the mutexes ops aren't
 511                          * really necessary, but if we ever loosen up, we will
 512                          * have this bit covered already.
 513                          */
 514                         keystack->keystack_flushdump--;
 515                         if (keystack->keystack_flushdump == 0) {
 516                                 /*
 517                                  * The flush/dump terminated by having a
 518                                  * consumer go away.  I need to send up to the
 519                                  * appropriate keysock all of the relevant
 520                                  * information.  Unfortunately, I don't
 521                                  * have that handy.
 522                                  */
 523                                 ks0dbg(("Consumer went away while flushing or"
 524                                     " dumping.\n"));
 525                         }
 526                 }
 527                 size = sizeof (keysock_consumer_t);
 528                 mutex_enter(&keystack->keystack_consumers_lock);
 529                 keystack->keystack_consumers[kc->kc_sa_type] = NULL;
 530                 mutex_exit(&keystack->keystack_consumers_lock);
 531                 mutex_destroy(&kc->kc_lock);
 532                 netstack_rele(kc->kc_keystack->keystack_netstack);
 533         } else {
 534                 ks = (keysock_t *)ptr;
 535                 keystack = ks->keysock_keystack;
```



```
 537                 ks3dbg(keystack,
 538                     ("Driver close, PF_KEY socket is going away.\n"));
 539                 if ((ks->keysock_flags & KEYSOCK_EXTENDED) != 0)
 540                         atomic_dec_32(&keystack->keystack_num_extended);
 540                         atomic_add_32(&keystack->keystack_num_extended, -1);
 541                 size = sizeof (keysock_t);
 542                 mutex_enter(&keystack->keystack_list_lock);
 543                 *(ks->keysock_ptpn) = ks->keysock_next;
 544                 if (ks->keysock_next != NULL)
 545                         ks->keysock_next->keysock_ptpn = ks->keysock_ptpn;
 546                 mutex_exit(&keystack->keystack_list_lock);
 547                 mutex_destroy(&ks->keysock_lock);
 548                 vmem_free(keysock_vmem, (void *)(uintptr_t)ks->keysock_serial,
 549                     1);
 550                 netstack_rele(ks->keysock_keystack->keystack_netstack);
 551         }

 553         /* Now I'm free. */
 554         kmem_free(ptr, size);
 555         return (0);
 556 }
_____unchanged_portion_omitted_

1522 /*
1523  * Spew an extended REGISTER down to the relevant consumers.
1524  */
1525 static void
1526 keysock_extended_register(keysock_t *ks, mblk_t *mp, sadb_ext_t *extv[])
1527 {
1528         sadb_x_ereg_t *ereg = (sadb_x_ereg_t *)extv[SADB_X_EXT_EREG];
1529         uint8_t *satypes, *fencepost;
1530         mblk_t *downmp;
1531         sadb_ext_t *downextv[SADB_EXT_MAX + 1];
1532         keysock_stack_t *keystack = ks->keysock_keystack;

1534         if (ks->keysock_registered[0] != 0 || ks->keysock_registered[1] != 0 ||
1535             ks->keysock_registered[2] != 0 || ks->keysock_registered[3] != 0) {
1536                 keysock_error(ks, mp, EBUSY, 0);
1537         }

1539         ks->keysock_flags |= KEYSOCK_EXTENDED;
1540         if (ereg == NULL) {
1541                 keysock_error(ks, mp, EINVAL, SADB_X_DIAGNOSTIC_SATYPE_NEEDED);
1542         } else {
1543                 ASSERT(mp->b_rptr + msgdsize(mp) == mp->b_wptr);
1544                 fencepost = (uint8_t *)mp->b_wptr;
1545                 satypes = ereg->sadb_x_ereg_satypes;
1546                 while (*satypes != SADB_SATYPE_UNSPEC && satypes != fencepost) {
1547                         downmp = copymsg(mp);
1548                         if (downmp == NULL) {
1549                                 keysock_error(ks, mp, ENOMEM, 0);
1550                                 return;
1551                         }
1552                         /*
1553                          * Since we've made it here, keysock_get_ext will work!
1554                          */
1555                         (void) keysock_get_ext(downextv,
1556                             (sadb_msg_t *)downmp->b_rptr, msgdsize(downmp),
1557                             keystack);
1558                         keysock_passdown(ks, downmp, *satypes, downextv,
1559                             B_FALSE);
1560                         ++satypes;
1561                 }
1562                 freemsg(mp);
1563         }
```

```
1565         /*
1566          * Set global to indicate we prefer an extended ACQUIRE.
1567          */
1568         atomic_inc_32(&keystack->keystack_num_extended);
1568         atomic_add_32(&keystack->keystack_num_extended, 1);
1569 }
______unchanged_portion_omitted_

2270 /*
2271  * The read procedure should only be invoked by a keysock consumer, like
2272  * ESP, AH, etc.  I should only see KEYSOCK_OUT and KEYSOCK_HELLO_ACK
2273  * messages on my read queues.
2274  */
2275 static void
2276 keysock_rput(queue_t *q, mblk_t *mp)
2277 {
2278         keysock_consumer_t *kc = (keysock_consumer_t *)q->q_ptr;
2279         ipsec_info_t *ii;
2280         keysock_hello_ack_t *ksa;
2281         minor_t serial;
2282         mblk_t *mp1;
2283         sadb_msg_t *samsg;
2284         keysock_stack_t *keystack = kc->kc_keystack;

2286         /* Make sure I'm a consumer instance.  (i.e. something's below me) */
2287         ASSERT(WR(q)->q_next != NULL);

2289         if (mp->b_datap->db_type != M_CTL) {
2290                 /*
2291                  * Keysock should only see keysock consumer interface
2292                  * messages (see ipsec_info.h) on its read procedure.
2293                  * To be robust, however, putnext() up so the STREAM head can
2294                  * deal with it appropriately.
2295                  */
2296                 ks1dbg(keystack,
2297                     ("Hmmm, a non M_CTL (%d, 0x%x) on keysock_rput.\n",
2298                     mp->b_datap->db_type, mp->b_datap->db_type));
2299                 putnext(q, mp);
2300                 return;
2301         }

2303         ii = (ipsec_info_t *)mp->b_rptr;

2305         switch (ii->ipsec_info_type) {
2306         case KEYSOCK_OUT:
2307                 /*
2308                  * A consumer needs to pass a response message or an ACQUIRE
2309                  * UP.  I assume that the consumer has done the right
2310                  * thing w.r.t. message creation, etc.
2311                  */
2312                 serial = ((keysock_out_t *)mp->b_rptr)->ks_out_serial;
2313                 mp1 = mp->b_cont;       /* Get M_DATA portion. */
2314                 freeb(mp);
2315                 samsg = (sadb_msg_t *)mp1->b_rptr;
2316                 if (samsg->sadb_msg_type == SADB_FLUSH ||
2317                     (samsg->sadb_msg_type == SADB_DUMP &&
2318                     samsg->sadb_msg_len == SADB_8TO64(sizeof (*samsg)))) {
2319                         /*
2320                          * If I'm an end-of-FLUSH or an end-of-DUMP marker...
2321                          */
2322                         ASSERT(keystack->keystack_flushdump != 0);
2323                                         /* Am I flushing? */

2325                         mutex_enter(&kc->kc_lock);
2326                         kc->kc_flags &= ~KC_FLUSHING;
```



```
2327                         mutex_exit(&kc->kc_lock);

2329                         if (samsg->sadb_msg_errno != 0)
2330                                 keystack->keystack_flushdump_errno =
2331                                     samsg->sadb_msg_errno;

2333                         /*
2334                          * Lower the atomic "flushing" count.  If it's
2335                          * the last one, send up the end-of-{FLUSH,DUMP} to
2336                          * the appropriate PF_KEY socket.
2337                          */
2338                         if (atomic_dec_32_nv(&keystack->keystack_flushdump) !=
2339                             0) {
2338                         if (atomic_add_32_nv(&keystack->keystack_flushdump,
2339                             -1) != 0) {
2340                                 ks1dbg(keystack,
2341                                     ("One flush/dump message back from %d,"
2342                                     " more to go.\n", samsg->sadb_msg_satype));
2343                                 freemsg(mp1);
2344                                 return;
2345                         }

2347                         samsg->sadb_msg_errno =
2348                             (uint8_t)keystack->keystack_flushdump_errno;
2349                         if (samsg->sadb_msg_type == SADB_DUMP) {
2350                                 samsg->sadb_msg_seq = 0;
2351                         }
2352                 }
2353                 keysock_passup(mp1, samsg, serial, kc,
2354                     (samsg->sadb_msg_type == SADB_DUMP), keystack);
2355                 return;
2356         case KEYSOCK_HELLO_ACK:
2357                 /* Aha, now we can link in the consumer! */
2358                 ksa = (keysock_hello_ack_t *)ii;
2359                 keysock_link_consumer(ksa->ks_hello_satype, kc);
2360                 freemsg(mp);
2361                 return;
2362         default:
2363                 ks1dbg(keystack, ("Hmmm, an IPsec info I'm not used to, 0x%x\n",
2364                     ii->ipsec_info_type));
2365                 putnext(q, mp);
2366         }
2367 }
______unchanged_portion_omitted_

2380 uint32_t
2381 keysock_next_seq(netstack_t *ns)
2382 {
2383         keysock_stack_t *keystack = ns->netstack_keysock;

2385         return (atomic_dec_32_nv(&keystack->keystack_acquire_seq));
2385         return (atomic_add_32_nv(&keystack->keystack_acquire_seq, -1));
2386 }
______unchanged_portion_omitted_
```

****
190737 Mon Jul 28 07:44:28 2014
new/usr/src/uts/common/inet/ip/spd.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____unchanged_portion_omitted_

```
3247 /*
3248  * Called when refcount goes to 0, indicating that all references to this
3249  * node are gone.
3250  *
3251  * This does not unchain the action from the hash table.
3252  */
3253 void
3254 ipsec_action_free(ipsec_action_t *ap)
3255 {
3256         for (;;) {
3257                 ipsec_action_t *np = ap->ipa_next;
3258                 ASSERT(ap->ipa_refs == 0);
3259                 ASSERT(ap->ipa_hash.hash_pp == NULL);
3260                 kmem_cache_free(ipsec_action_cache, ap);
3261                 ap = np;
3262                 /* Inlined IPACT_REFRELE -- avoid recursion */
3263                 if (ap == NULL)
3264                         break;
3265                 membar_exit();
3266                 if (atomic_dec_32_nv(&(ap)->ipa_refs) != 0)
3266                 if (atomic_add_32_nv(&(ap)->ipa_refs, -1) != 0)
3267                         break;
3268                 /* End inlined IPACT_REFRELE */
3269         }
3270 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   14501 Mon Jul 28 07:44:28 2014
new/usr/src/uts/common/inet/ip_ire.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/* Copyright (c) 1990 Mentat Inc. */

#ifndef _INET_IP_IRE_H
#define _INET_IP_IRE_H

#ifdef  __cplusplus
extern "C" {
#endif

#define IPV6_LL_PREFIXLEN       10      /* Number of bits in link-local pref */

#define IP_CACHE_TABLE_SIZE     256
#define IP_MASK_TABLE_SIZE      (IP_ABITS + 1)          /* 33 ptrs */

#define IP6_FTABLE_HASH_SIZE    32      /* size of each hash table in ptrs */
#define IP6_CACHE_TABLE_SIZE    256
#define IP6_MASK_TABLE_SIZE     (IPV6_ABITS + 1)        /* 129 ptrs */

/*
 * We use the common modulo hash function.  In ip_ire_init(), we make
 * sure that the cache table size is always a power of 2.  That's why
 * we can use & instead of %.  Also note that we try hard to make sure
 * the lower bits of an address capture most info from the whole address.
 * The reason being that since our hash table is probably a lot smaller
 * than 2^32 buckets so the lower bits are the most important.
 */
#define IRE_ADDR_HASH(addr, table_size) \
        (((addr) ^ ((addr) >> 8) ^ ((addr) >> 16) ^ ((addr) >> 24)) &   \
        ((table_size) - 1))

/*
 * To make a byte-order neutral hash for IPv6, just take all the
 * bytes in the bottom 32 bits into account.
 */
#define IRE_ADDR_HASH_V6(addr, table_size)                              \
        IRE_ADDR_HASH((addr).s6_addr32[3], table_size)

/*
```



```
 * This assumes that the ftable size is a power of 2.
 * We include some high-order bytes to avoid all IRE_LOCALs in the same
 * bucket for performance reasons.
 */
#define IRE_ADDR_MASK_HASH_V6(addr, mask, table_size)                   \
        ((((addr).s6_addr8[0] & (mask).s6_addr8[0]) ^                   \
        ((addr).s6_addr8[1] & (mask).s6_addr8[1]) ^                     \
        ((addr).s6_addr8[6] & (mask).s6_addr8[6]) ^                     \
        ((addr).s6_addr8[7] & (mask).s6_addr8[7]) ^                     \
        ((addr).s6_addr8[8] & (mask).s6_addr8[8]) ^                     \
        ((addr).s6_addr8[9] & (mask).s6_addr8[9]) ^                     \
        ((addr).s6_addr8[10] & (mask).s6_addr8[10]) ^                   \
        ((addr).s6_addr8[13] & (mask).s6_addr8[13]) ^                   \
        ((addr).s6_addr8[14] & (mask).s6_addr8[14]) ^                   \
        ((addr).s6_addr8[15] & (mask).s6_addr8[15])) & ((table_size) - 1))

#define IRE_HIDDEN_TYPE(ire_type) ((ire_type) &                 \
        (IRE_HOST | IRE_PREFIX | IRE_DEFAULT | IRE_IF_ALL | IRE_BROADCAST))

/*
 * match parameter definitions for IRE lookup routines.
 */
#define MATCH_IRE_DSTONLY       0x0000  /* Match just the address */
#define MATCH_IRE_TYPE          0x0001  /* Match IRE type */
#define MATCH_IRE_MASK          0x0002  /* Match IRE mask */
#define MATCH_IRE_SHORTERMASK   0x0004  /* A mask shorter than the argument */
#define MATCH_IRE_GW            0x0008  /* Match IRE gateway */
#define MATCH_IRE_ILL           0x0010  /* Match IRE on the ill */
#define MATCH_IRE_ZONEONLY      0x0020  /* Match IREs in specified zone, ie */
                                        /* don't match IRE_LOCALs from other */
                                        /* zones or shared IREs */
#define MATCH_IRE_SECATTR       0x0040  /* Match gateway security attributes */
#define MATCH_IRE_TESTHIDDEN    0x0080  /* Match ire_testhidden IREs */
#define MATCH_IRE_SRC_ILL       0x0100  /* ire_ill uses a src address on ill */
#define MATCH_IRE_DIRECT        0x0200  /* Don't match indirect routes */

#define MAX_IRE_RECURSION       4       /* Max IREs in ire_route_recursive */


/*
 * We use atomics so that we get an accurate accounting on the ires.
 * Otherwise we can't determine leaks correctly.
 */
#define BUMP_IRE_STATS(ire_stats, x) atomic_inc_64(&(ire_stats).x)
#define BUMP_IRE_STATS(ire_stats, x) atomic_add_64(&(ire_stats).x, 1)

#ifdef _KERNEL
struct ts_label_s;
struct nce_s;
/*
 * structure for passing args between ire_ftable_lookup and ire_find_best_route
 */
typedef struct ire_ftable_args_s {
        in6_addr_t              ift_addr_v6;
        in6_addr_t              ift_mask_v6;
        in6_addr_t              ift_gateway_v6;
#define ift_addr                V4_PART_OF_V6(ift_addr_v6)
#define ift_mask                V4_PART_OF_V6(ift_mask_v6)
#define ift_gateway             V4_PART_OF_V6(ift_gateway_v6)
        int                     ift_type;
        const ill_t             *ift_ill;
        zoneid_t                ift_zoneid;
        const ts_label_t        *ift_tsl;
        int                     ift_flags;
        ire_t                   *ift_best_ire;
} ire_ftable_args_t;
______unchanged_portion_omitted_
```

```
**********************************************************
   70498 Mon Jul 28 07:44:28 2014
new/usr/src/uts/common/inet/ipf/netinet/ip_compat.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 221 extern void mb_copydata __P((mblk_t *, size_t , size_t, char *));
 222 extern void mb_copyback __P((mblk_t *, size_t , size_t, char *));
 223 # endif

 225 # if SOLARIS2 >= 6
 226 #  include <sys/atomic.h>
 227 typedef uint32_t        u_32_t;
 228 # else
 229 typedef unsigned int    u_32_t;
 230 # endif
 231 # define        U_32_T  1

 233 # ifdef _KERNEL
 234 #  define       KRWLOCK_T               krwlock_t
 235 #  define       KMUTEX_T                kmutex_t
 236 #  if SOLARIS2 >= 10
 237 #   include <sys/sdt.h>

 239 #   define IPF_IS_LOOPBACK(f)   ((f) & FI_NOCKSUM)
 240 #  endif /* SOLARIS2 >= 10 */
 241 #  if SOLARIS2 >= 6
 242 #   if SOLARIS2 == 6
 243 #    define     ATOMIC_INCL(x)          atomic_inc_ulong((uint32_t *)&(x))
 244 #    define     ATOMIC_DECL(x)          atomic_dec_ulong((uint32_t *)&(x))
 243 #    define     ATOMIC_INCL(x)          atomic_add_long((uint32_t*)&(x), 1)
 244 #    define     ATOMIC_DECL(x)          atomic_add_long((uint32_t*)&(x), -1)
 245 #   else
 246 #    define     ATOMIC_INCL(x)          atomic_inc_ulong(&(x))
 247 #    define     ATOMIC_DECL(x)          atomic_dec_ulong(&(x))
 246 #    define     ATOMIC_INCL(x)          atomic_add_long(&(x), 1)
 247 #    define     ATOMIC_DECL(x)          atomic_add_long(&(x), -1)
 248 #   endif /* SOLARIS2 == 6 */
 249 #   define      ATOMIC_INC64(x)         atomic_inc_64((uint64_t *)&(x))
 250 #   define      ATOMIC_INC32(x)         atomic_inc_32((uint32_t *)&(x))
 251 #   define      ATOMIC_INC16(x)         atomic_inc_16((uint16_t *)&(x))
 252 #   define      ATOMIC_DEC64(x)         atomic_dec_64((uint64_t *)&(x))
 253 #   define      ATOMIC_DEC32(x)         atomic_dec_32((uint32_t *)&(x))
 254 #   define      ATOMIC_DEC16(x)         atomic_dec_16((uint16_t *)&(x))
 249 #   define      ATOMIC_INC64(x)         atomic_add_64((uint64_t*)&(x), 1)
 250 #   define      ATOMIC_INC32(x)         atomic_add_32((uint32_t*)&(x), 1)
 251 #   define      ATOMIC_INC16(x)         atomic_add_16((uint16_t*)&(x), 1)
 252 #   define      ATOMIC_DEC64(x)         atomic_add_64((uint64_t*)&(x), -1)
 253 #   define      ATOMIC_DEC32(x)         atomic_add_32((uint32_t*)&(x), -1)
 254 #   define      ATOMIC_DEC16(x)         atomic_add_16((uint16_t*)&(x), -1)
 255 #  else
 256 #   define      ATOMIC_INC(x)           { mutex_enter(&ipf_rw); (x)++; \
 257                                           mutex_exit(&ipf_rw); }
 258 #   define      ATOMIC_DEC(x)           { mutex_enter(&ipf_rw); (x)--; \
 259                                           mutex_exit(&ipf_rw); }
 260 #  endif /* SOLARIS2 >= 6 */
 261 #  define       USE_MUTEXES
 262 #  define       MUTEX_ENTER(x)          mutex_enter(&(x)->ipf_lk)
 263 #  define       READ_ENTER(x)           rw_enter(&(x)->ipf_lk, RW_READER)
 264 #  define       WRITE_ENTER(x)          rw_enter(&(x)->ipf_lk, RW_WRITER)
 265 #  define       MUTEX_DOWNGRADE(x)      rw_downgrade(&(x)->ipf_lk)
 266 #  define       RWLOCK_INIT(x, y)       rw_init(&(x)->ipf_lk, (y),  \
 267                                                RW_DRIVER, NULL)
 268 #  define       RWLOCK_EXIT(x)          rw_exit(&(x)->ipf_lk)
 269 #  define       RW_DESTROY(x)           rw_destroy(&(x)->ipf_lk)
```



```
 270 #  define       MUTEX_INIT(x, y)        mutex_init(&(x)->ipf_lk, (y), \
 271                                                   MUTEX_DRIVER, NULL)
 272 #  define       MUTEX_DESTROY(x)        mutex_destroy(&(x)->ipf_lk)
 273 #  define       MUTEX_NUKE(x)           bzero((x), sizeof(*(x)))
 274 #  define       MUTEX_EXIT(x)           mutex_exit(&(x)->ipf_lk)
 275 #  define       COPYIN(a,b,c)   copyin((caddr_t)(a), (caddr_t)(b), (c))
 276 #  define       COPYOUT(a,b,c)  copyout((caddr_t)(a), (caddr_t)(b), (c))
 277 #  define       BCOPYIN(a,b,c)  copyin((caddr_t)(a), (caddr_t)(b), (c))
 278 #  define       BCOPYOUT(a,b,c) copyout((caddr_t)(a), (caddr_t)(b), (c))
 279 #  define       UIOMOVE(a,b,c,d)        uiomove((caddr_t)a,b,c,d)
 280 #  define       KFREE(x)        kmem_free((char *)(x), sizeof(*(x)))
 281 #  define       KFREES(x,s)     kmem_free((char *)(x), (s))
 282 #  define       SPL_NET(x)      ;
 283 #  define       SPL_IMP(x)      ;
 284 #  undef        SPL_X
 285 #  define       SPL_X(x)        ;
 286 #  ifdef sparc
 287 #   define      ntohs(x)        (x)
 288 #   define      ntohl(x)        (x)
 289 #   define      htons(x)        (x)
 290 #   define      htonl(x)        (x)
 291 #  endif /* sparc */
 292 #  define       KMALLOC(a,b)    (a) = (b)kmem_alloc(sizeof(*(a)), KM_NOSLEEP)
 293 #  define       KMALLOCS(a,b,c) (a) = (b)kmem_alloc((c), KM_NOSLEEP)
 294 #  define       GET_MINOR(x)    getminor(x)
 295 /*extern        phy_if_t        get_unit __P((char *, int, ipf_stack_t *));*/
 296 #  define       GETIFP(n, v, ifs)       (void *)get_unit(n, v, ifs)
 297 #  define       IFNAME(x)       ((ill_t *)x)->ill_name
 298 #  define       COPYIFNAME(x, b, v)     (void) net_getifname(((v) == 4) ? \
 299                                           ifs->ifs_ipf_ipv4 : ifs->ifs_ipf_ipv6,\
 300                                           (phy_if_t)(x), (b), sizeof(b))
 301 #  define       GETKTIME(x)     uniqtime((struct timeval *)x)
 302 #  define       MSGDSIZE(x)     msgdsize(x)
 303 #  define       M_LEN(x)        ((x)->b_wptr - (x)->b_rptr)
 304 #  define       M_DUPLICATE(x)  copymsg((x))
 305 #  define       MTOD(m,t)       ((t)((m)->b_rptr))
 306 #  define       MTYPE(m)        ((m)->b_datap->db_type)
 307 #  define       FREE_MB_T(m)    freemsg(m)
 308 #  define       m_next          b_cont
 309 #  define       CACHE_HASH(x)   (((phy_if_t)(x)->fin_ifp) & 7)
 310 #  define       IPF_PANIC(x,y)  if (x) { printf y; cmn_err(CE_PANIC, "ipf_panic"
 311 typedef mblk_t mb_t;
 312 # endif /* _KERNEL */

 314 # if (SOLARIS2 >= 7)
 315 #  ifdef lint
 316 #   define ALIGN32(ptr)    (ptr ? 0L : 0L)
 317 #   define ALIGN16(ptr)    (ptr ? 0L : 0L)
 318 #  else
 319 #   define ALIGN32(ptr)    (ptr)
 320 #   define ALIGN16(ptr)    (ptr)
 321 #  endif
 322 # endif

 324 # if SOLARIS2 < 6
 325 typedef struct uio      uio_t;
 326 # endif
 327 typedef int             ioctlcmd_t;
 328 typedef uint8_t         u_int8_t;

 330 # define OS_RECOGNISED 1

 332 #endif /* SOLARIS */

 334 /* ------------------------------------------------------------------------ */
 335 /*                                H P U X                                   */
```

```
 336 /* ------------------------------------------------------------------------ */
 337 #ifdef __hpux
 338 # define        MENTAT  1
 339 # include       <sys/sysmacros.h>
 340 # include       <sys/spinlock.h>
 341 # include       <sys/lock.h>
 342 # include       <sys/stream.h>
 343 # ifdef USE_INET6
 344 #  include      <netinet/if_ether.h>
 345 #  include      <netinet/ip6.h>
 346 #  include      <netinet/icmp6.h>
 347 typedef struct  ip6_hdr ip6_t;
 348 # endif

 350 # ifdef _KERNEL
 351 #  define SNPRINTF      sprintf
 352 #  if (HPUXREV >= 1111)
 353 #   define      IPL_SELECT
 354 #   ifdef       IPL_SELECT
 355 #    include    <machine/sys/user.h>
 356 #    include    <sys/kthread_iface.h>
 357 #    define     READ_COLLISION  0x01

 359 typedef struct  iplog_select_s {
 360         kthread_t       *read_waiter;
 361         int             state;
 362 } iplog_select_t;
_____unchanged_portion_omitted_

 745 typedef int             ioctlcmd_t;
 746 /*
 747  * Really, any arch where sizeof(long) != sizeof(int).
 748  */
 749 typedef unsigned int    u_32_t;
 750 # define        U_32_T  1

 752 # define OS_RECOGNISED 1
 753 #endif /* __osf__ */

 755 /* ------------------------------------------------------------------------ */
 756 /*                                  N E T B S D                             */
 757 /* ------------------------------------------------------------------------ */
 758 #ifdef __NetBSD__
 759 # if defined(_KERNEL) && !defined(IPFILTER_LKM)
 760 #  include "bpfilter.h"
 761 #  if defined(__NetBSD_Version__) && (__NetBSD_Version__ >= 104110000)
 762 #   include "opt_inet.h"
 763 #  endif
 764 #  ifdef INET6
 765 #   define USE_INET6
 766 #  endif
 767 #  if (__NetBSD_Version__ >= 105000000)
 768 #   define HAVE_M_PULLDOWN 1
 769 #  endif
 770 # endif

 772 # ifdef _KERNEL
 773 #  define       MSGDSIZE(x)     mbufchainlen(x)
 774 #  define       M_LEN(x)        (x)->m_len
 775 #  define       M_DUPLICATE(x)  m_copy((x), 0, M_COPYALL)
 776 #  define       GETKTIME(x)     microtime((struct timeval *)x)
 777 #  define       IPF_PANIC(x,y)  if (x) { printf y; panic("ipf_panic"); }
 778 #  define       COPYIN(a,b,c)   copyin((caddr_t)(a), (caddr_t)(b), (c))
 779 #  define       COPYOUT(a,b,c)  copyout((caddr_t)(a), (caddr_t)(b), (c))
 780 #  define       BCOPYIN(a,b,c)  bcopy((caddr_t)(a), (caddr_t)(b), (c))
 781 #  define       BCOPYOUT(a,b,c) bcopy((caddr_t)(a), (caddr_t)(b), (c))
```



```
 782 typedef struct mbuf mb_t;
 783 # endif /* _KERNEL */
 784 # if (NetBSD <= 1991011) && (NetBSD >= 199606)
 785 #  define       IFNAME(x)       ((struct ifnet *)x)->if_xname
 786 #  define       COPYIFNAME(x, b, v) \
 787                                   (void) strncpy(b, \
 788                                                  ((struct ifnet *)x)->if_xname, \
 789                                                  LIFNAMSIZ)
 790 #  define       CACHE_HASH(x)   ((((struct ifnet *)fin->fin_ifp)->if_index)&7)
 791 # else
 792 #  define       CACHE_HASH(x)   ((IFNAME(fin->fin_ifp)[0] + \
 793                                   ((struct ifnet *)fin->fin_ifp)->if_unit) & 7)
 794 # endif

 796 typedef struct uio      uio_t;
 797 typedef u_long          ioctlcmd_t;
 798 typedef int             minor_t;
 799 typedef u_int32_t       u_32_t;
 800 # define        U_32_T  1

 802 # define OS_RECOGNISED 1
 803 #endif /* __NetBSD__ */


 806 /* ------------------------------------------------------------------------ */
 807 /*                                  F R E E B S D                           */
 808 /* ------------------------------------------------------------------------ */
 809 #ifdef __FreeBSD__
 810 # if defined(_KERNEL)
 811 #  if (__FreeBSD_version >= 500000)
 812 #   include "opt_bpf.h"
 813 #  else
 814 #   include "bpf.h"
 815 #  endif
 816 #  if defined(__FreeBSD_version) && (__FreeBSD_version >= 400000)
 817 #   include "opt_inet6.h"
 818 #  endif
 819 #  if defined(INET6) && !defined(USE_INET6)
 820 #   define USE_INET6
 821 #  endif
 822 # endif

 824 # if defined(_KERNEL)
 825 #  if (__FreeBSD_version >= 400000)
 826 /*
 827  * When #define'd, the 5.2.1 kernel panics when used with the ftp proxy.
 828  * There may be other, safe, kernels but this is not extensively tested yet.
 829  */
 830 #   define HAVE_M_PULLDOWN
 831 #  endif
 832 #  if !defined(IPFILTER_LKM) && (__FreeBSD_version >= 300000)
 833 #   include "opt_ipfilter.h"
 834 #  endif
 835 #  define       COPYIN(a,b,c)   copyin((caddr_t)(a), (caddr_t)(b), (c))
 836 #  define       COPYOUT(a,b,c)  copyout((caddr_t)(a), (caddr_t)(b), (c))
 837 #  define       BCOPYIN(a,b,c)  bcopy((caddr_t)(a), (caddr_t)(b), (c))
 838 #  define       BCOPYOUT(a,b,c) bcopy((caddr_t)(a), (caddr_t)(b), (c))

 840 #  if (__FreeBSD_version >= 500043)
 841 #   define NETBSD_PF
 842 #  endif
 843 # endif /* _KERNEL */

 845 # if (__FreeBSD_version >= 500043)
 846 #  include <sys/mutex.h>
 847 #  include <sys/sx.h>
```

```
/*
 * Whilst the sx(9) locks on FreeBSD have the right semantics and interface
 * for what we want to use them for, despite testing showing they work -
 * with a WITNESS kernel, it generates LOR messages.
 */
#  define       KMUTEX_T                struct mtx
#  if 1
#   define      KRWLOCK_T               struct mtx
#  else
#   define      KRWLOCK_T               struct sx
#  endif
# endif

# if (__FreeBSD_version >= 501113)
#  include <net/if_var.h>
#  define       IFNAME(x)       ((struct ifnet *)x)->if_xname
#  define       COPYIFNAME(x, b) \
                                (void) strncpy(b, \
                                               ((struct ifnet *)x)->if_xname, \
                                               LIFNAMSIZ)
# endif
# if (__FreeBSD_version >= 500043)
#  define       CACHE_HASH(x)   ((((struct ifnet *)fin->fin_ifp)->if_index) & 7)
# else
#  define       CACHE_HASH(x)   ((IFNAME(fin->fin_ifp)[0] + \
                                  ((struct ifnet *)fin->fin_ifp)->if_unit) & 7)
# endif

# ifdef _KERNEL
#  define       GETKTIME(x)     microtime((struct timeval *)x)

#  if (__FreeBSD_version >= 500002)
#   include <netinet/in_systm.h>
#   include <netinet/ip.h>
#   include <machine/in_cksum.h>
#  endif

#  if (__FreeBSD_version >= 500043)
#   define      USE_MUTEXES
#   define      MUTEX_ENTER(x)          mtx_lock(&(x)->ipf_lk)
#   define      MUTEX_EXIT(x)           mtx_unlock(&(x)->ipf_lk)
#   define      MUTEX_INIT(x,y)         mtx_init(&(x)->ipf_lk, (y), NULL,\
                                                MTX_DEF)
#   define      MUTEX_DESTROY(x)        mtx_destroy(&(x)->ipf_lk)
#   define      MUTEX_NUKE(x)           bzero((x), sizeof(*(x)))
/*
 * Whilst the sx(9) locks on FreeBSD have the right semantics and interface
 * for what we want to use them for, despite testing showing they work -
 * with a WITNESS kernel, it generates LOR messages.
 */
#   if 1
#    define     READ_ENTER(x)           mtx_lock(&(x)->ipf_lk)
#    define     WRITE_ENTER(x)          mtx_lock(&(x)->ipf_lk)
#    define     RWLOCK_EXIT(x)          mtx_unlock(&(x)->ipf_lk)
#    define     MUTEX_DOWNGRADE(x)      ;
#    define     RWLOCK_INIT(x,y)        mtx_init(&(x)->ipf_lk, (y), NULL,\
                                                 MTX_DEF)
#    define     RW_DESTROY(x)           mtx_destroy(&(x)->ipf_lk)
#   else
#    define     READ_ENTER(x)           sx_slock(&(x)->ipf_lk)
#    define     WRITE_ENTER(x)          sx_xlock(&(x)->ipf_lk)
#    define     MUTEX_DOWNGRADE(x)      sx_downgrade(&(x)->ipf_lk)
#    define     RWLOCK_INIT(x, y)       sx_init(&(x)->ipf_lk, (y))
#    define     RW_DESTROY(x)           sx_destroy(&(x)->ipf_lk)
#    ifdef sx_unlock
#     define    RWLOCK_EXIT(x)          sx_unlock(x)
```



```
#    else
#     define    RWLOCK_EXIT(x)          do { \
                                             if ((x)->ipf_lk.sx_cnt < 0) \
                                                 sx_xunlock(&(x)->ipf_lk); \
                                             else \
                                                 sx_sunlock(&(x)->ipf_lk); \
                                         } while (0)
#    endif
#   endif
#   include <machine/atomic.h>
#   define      ATOMIC_INC(x)           { mtx_lock(&ipf_rw.ipf_lk); (x)++; \
                                           mtx_unlock(&ipf_rw.ipf_lk); }
#   define      ATOMIC_DEC(x)           { mtx_lock(&ipf_rw.ipf_lk); (x)--; \
                                           mtx_unlock(&ipf_rw.ipf_lk); }
#   define      ATOMIC_INCL(x)          atomic_inc_ulong(&(x))
#   define      ATOMIC_INCL(x)          atomic_add_long(&(x), 1)
#   define      ATOMIC_INC64(x)         ATOMIC_INC(x)
#   define      ATOMIC_INC32(x)         atomic_inc_32(&(x))
#   define      ATOMIC_INC16(x)         atomic_inc_16(&(x))
#   define      ATOMIC_DECL(x)          atomic_dec_ulong(&(x))
#   define      ATOMIC_INC32(x)         atomic_add_32(&(x), 1)
#   define      ATOMIC_INC16(x)         atomic_add_16(&(x), 1)
#   define      ATOMIC_DECL(x)          atomic_add_long(&(x), -1)
#   define      ATOMIC_DEC64(x)         ATOMIC_DEC(x)
#   define      ATOMIC_DEC32(x)         atomic_dec_32(&(x))
#   define      ATOMIC_DEC16(x)         atomic_dec_16(&(x))
#   define      ATOMIC_DEC32(x)         atomic_add_32(&(x), -1)
#   define      ATOMIC_DEC16(x)         atomic_add_16(&(x), -1)
#   define      SPL_X(x)        ;
#   define      SPL_NET(x)      ;
#   define      SPL_IMP(x)      ;
extern  int     in_cksum __P((struct mbuf *, int));
#  endif /* __FreeBSD_version >= 500043 */
#  define       MSGDSIZE(x)     mbufchainlen(x)
#  define       M_LEN(x)        (x)->m_len
#  define       M_DUPLICATE(x)  m_copy((x), 0, M_COPYALL)
#  define       IPF_PANIC(x,y)  if (x) { printf y; panic("ipf_panic"); }
typedef struct mbuf mb_t;
# endif /* _KERNEL */

# if __FreeBSD__ < 3
#  include <machine/spl.h>
# else
#  if __FreeBSD__ == 3
#   if defined(IPFILTER_LKM) && !defined(ACTUALLY_LKM_NOT_KERNEL)
#    define     ACTUALLY_LKM_NOT_KERNEL
#   endif
#  endif
# endif

# if (__FreeBSD_version >= 300000)
typedef u_long          ioctlcmd_t;
# else
typedef int             ioctlcmd_t;
# endif
typedef struct uio      uio_t;
typedef int             minor_t;
typedef u_int32_t       u_32_t;
# define        U_32_T  1

# define OS_RECOGNISED 1
#endif /* __FreeBSD__ */


/* ------------------------------------------------------------------------ */
/*                                O P E N B S D                             */
```

```
 974 /* ------------------------------------------------------------------------ */
 975 #ifdef __OpenBSD__
 976 # ifdef INET6
 977 #  define USE_INET6
 978 # endif

 980 # ifdef _KERNEL
 981 #  if !defined(IPFILTER_LKM)
 982 #   include "bpfilter.h"
 983 #  endif
 984 #  if (OpenBSD >= 200311)
 985 #   define SNPRINTF     snprintf
 986 #   if defined(USE_INET6)
 987 #    include "netinet6/in6_var.h"
 988 #    include "netinet6/nd6.h"
 989 #   endif
 990 #  endif
 991 #  if (OpenBSD >= 200012)
 992 #   define HAVE_M_PULLDOWN 1
 993 #  endif
 994 #  define       COPYIN(a,b,c)   copyin((caddr_t)(a), (caddr_t)(b), (c))
 995 #  define       COPYOUT(a,b,c)  copyout((caddr_t)(a), (caddr_t)(b), (c))
 996 #  define       BCOPYIN(a,b,c)  bcopy((caddr_t)(a), (caddr_t)(b), (c))
 997 #  define       BCOPYOUT(a,b,c) bcopy((caddr_t)(a), (caddr_t)(b), (c))
 998 #  define       GETKTIME(x)     microtime((struct timeval *)x)
 999 #  define       MSGDSIZE(x)     mbufchainlen(x)
1000 #  define       M_LEN(x)        (x)->m_len
1001 #  define       M_DUPLICATE(x)  m_copy((x), 0, M_COPYALL)
1002 #  define       IPF_PANIC(x,y)  if (x) { printf y; panic("ipf_panic"); }
1003 typedef struct mbuf mb_t;
1004 # endif /* _KERNEL */
1005 # if (OpenBSD >= 199603)
1006 #  define       IFNAME(x, b)    ((struct ifnet *)x)->if_xname
1007 #  define       COPYIFNAME(x, b, v) \
1008                                (void) strncpy(b, \
1009                                               ((struct ifnet *)x)->if_xname, \
1010                                               LIFNAMSIZ)
1011 #  define       CACHE_HASH(x)   ((((struct ifnet *)fin->fin_ifp)->if_index)&7)
1012 # else
1013 #  define       CACHE_HASH(x)   ((IFNAME(fin->fin_ifp)[0] + \
1014                                   ((struct ifnet *)fin->fin_ifp)->if_unit) & 7)
1015 # endif

1017 typedef struct uio      uio_t;
1018 typedef u_long          ioctlcmd_t;
1019 typedef int             minor_t;
1020 typedef u_int32_t       u_32_t;
1021 # define        U_32_T  1

1023 # define OS_RECOGNISED 1
1024 #endif /* __OpenBSD__ */


1027 /* ------------------------------------------------------------------------ */
1028 /*                                B S D O S                                 */
1029 /* ------------------------------------------------------------------------ */
1030 #ifdef _BSDI_VERSION
1031 # ifdef INET6
1032 #  define USE_INET6
1033 # endif

1035 # ifdef _KERNEL
1036 #  define       GETKTIME(x)     microtime((struct timeval *)x)
1037 #  define       MSGDSIZE(x)     mbufchainlen(x)
1038 #  define       M_LEN(x)        (x)->m_len
1039 #  define       M_DUPLICATE(x)  m_copy((x), 0, M_COPYALL)
```



```
1040 #  define       CACHE_HASH(x)   ((IFNAME(fin->fin_ifp)[0] + \
1041                                   ((struct ifnet *)fin->fin_ifp)->if_unit) & 7)
1042 typedef struct mbuf mb_t;
1043 # endif /* _KERNEL */

1045 # if (_BSDI_VERSION >= 199701)
1046 typedef u_long          ioctlcmd_t;
1047 # else
1048 typedef int             ioctlcmd_t;
1049 # endif
1050 typedef u_int32_t       u_32_t;
1051 # define        U_32_T  1

1053 #endif /* _BSDI_VERSION */


1056 /* ------------------------------------------------------------------------ */
1057 /*                                S U N O S 4                               */
1058 /* ------------------------------------------------------------------------ */
1059 #if defined(sun) && !defined(OS_RECOGNISED) /* SunOS4 */
1060 # ifdef _KERNEL
1061 #  include      <sys/kmem_alloc.h>
1062 #  define       GETKTIME(x)     uniqtime((struct timeval *)x)
1063 #  define       MSGDSIZE(x)     mbufchainlen(x)
1064 #  define       M_LEN(x)        (x)->m_len
1065 #  define       M_DUPLICATE(x)  m_copy((x), 0, M_COPYALL)
1066 #  define       CACHE_HASH(x)   ((IFNAME(fin->fin_ifp)[0] + \
1067                                   ((struct ifnet *)fin->fin_ifp)->if_unit) & 7)
1068 #  define       GETIFP(n, v, ifs)       ifunit(n, IFNAMSIZ)
1069 #  define       KFREE(x)        kmem_free((char *)(x), sizeof(*(x)))
1070 #  define       KFREES(x,s)     kmem_free((char *)(x), (s))
1071 #  define       SLEEP(id, n)    sleep((id), PZERO+1)
1072 #  define       WAKEUP(id,x)    wakeup(id + x)
1073 #  define       UIOMOVE(a,b,c,d)        uiomove((caddr_t)a,b,c,d)
1074 #  define       IPF_PANIC(x,y)  if (x) { printf y; panic("ipf_panic"); }

1076 extern  void    m_copydata __P((struct mbuf *, int, int, caddr_t));
1077 extern  void    m_copyback __P((struct mbuf *, int, int, caddr_t));

1079 typedef struct mbuf mb_t;
1080 # endif

1082 typedef struct uio      uio_t;
1083 typedef int             ioctlcmd_t;
1084 typedef int             minor_t;
1085 typedef unsigned int    u_32_t;
1086 # define        U_32_T  1

1088 # define OS_RECOGNISED 1

1090 #endif /* SunOS 4 */

1092 /* ------------------------------------------------------------------------ */
1093 /*                              L I N U X                                   */
1094 /* ------------------------------------------------------------------------ */
1095 #if defined(linux) && !defined(OS_RECOGNISED)
1096 #include <linux/config.h>
1097 #include <linux/version.h>
1098 # if LINUX >= 20600
1099 #  define        HDR_T_PRIVATE  1
1100 # endif
1101 # undef USE_INET6
1102 # ifdef USE_INET6
1103 struct ip6_ext {
1104         u_char  ip6e_nxt;
1105         u_char  ip6e_len;
```

```
1106 };
```

*unchanged_portion_omitted_*

```
**********************************************************
   32871 Mon Jul 28 07:44:29 2014
new/usr/src/uts/common/inet/ipsec_impl.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 300 #define IPACT_REFHOLD(ipa) {                    \
 301         atomic_inc_32(&(ipa)->ipa_refs);        \
 301         atomic_add_32(&(ipa)->ipa_refs, 1);     \
 302         ASSERT((ipa)->ipa_refs != 0);   \
 303 }
 304 #define IPACT_REFRELE(ipa) {                                    \
 305         ASSERT((ipa)->ipa_refs != 0);                           \
 306         membar_exit();                                          \
 307         if (atomic_dec_32_nv(&(ipa)->ipa_refs) == 0)    \
 307         if (atomic_add_32_nv(&(ipa)->ipa_refs, -1) == 0)        \
 308                 ipsec_action_free(ipa);                         \
 309         (ipa) = 0;                                              \
 310 }
______unchanged_portion_omitted_

 416 #define IPPOL_REFHOLD(ipp) {                    \
 417         atomic_inc_32(&(ipp)->ipsp_refs);       \
 417         atomic_add_32(&(ipp)->ipsp_refs, 1);    \
 418         ASSERT((ipp)->ipsp_refs != 0);          \
 419 }
 420 #define IPPOL_REFRELE(ipp) {                                    \
 421         ASSERT((ipp)->ipsp_refs != 0);                          \
 422         membar_exit();                                          \
 423         if (atomic_dec_32_nv(&(ipp)->ipsp_refs) == 0)   \
 423         if (atomic_add_32_nv(&(ipp)->ipsp_refs, -1) == 0)       \
 424                 ipsec_policy_free(ipp);                         \
 425         (ipp) = 0;                                              \
 426 }
______unchanged_portion_omitted_

 463 #define IPPH_REFHOLD(iph) {                     \
 464         atomic_inc_32(&(iph)->iph_refs);        \
 464         atomic_add_32(&(iph)->iph_refs, 1);     \
 465         ASSERT((iph)->iph_refs != 0);           \
 466 }
 467 #define IPPH_REFRELE(iph, ns) {                                 \
 468         ASSERT((iph)->iph_refs != 0);                           \
 469         membar_exit();                                          \
 470         if (atomic_dec_32_nv(&(iph)->iph_refs) == 0)    \
 470         if (atomic_add_32_nv(&(iph)->iph_refs, -1) == 0)        \
 471                 ipsec_polhead_free(iph, ns);                    \
 472         (iph) = 0;                                              \
 473 }
______unchanged_portion_omitted_
 520 /* NOTE - Callers (tun code) synchronize their own instances for these flags. */
 521 #define ITPF_P_ACTIVE 0x1       /* Are we using IPsec right now? */
 522 #define ITPF_P_TUNNEL 0x2       /* Negotiate tunnel-mode */
 523 /* Optimization -> Do we have per-port security entries in this polhead? */
 524 #define ITPF_P_PER_PORT_SECURITY 0x4
 525 #define ITPF_PFLAGS 0x7
 526 #define ITPF_SHIFT 3

 528 #define ITPF_I_ACTIVE 0x8       /* Is the inactive using IPsec right now? */
 529 #define ITPF_I_TUNNEL 0x10      /* Negotiate tunnel-mode (on inactive) */
 530 /* Optimization -> Do we have per-port security entries in this polhead? */
 531 #define ITPF_I_PER_PORT_SECURITY 0x20
 532 #define ITPF_IFLAGS 0x38

 534 /* NOTE:  f cannot be an expression. */
```



```
 535 #define ITPF_CLONE(f) (f) = (((f) & ITPF_PFLAGS) | \
 536             (((f) & ITPF_PFLAGS) << ITPF_SHIFT));
 537 #define ITPF_SWAP(f) (f) = ((((f) & ITPF_PFLAGS) << ITPF_SHIFT) | \
 538             (((f) & ITPF_IFLAGS) >> ITPF_SHIFT))

 540 #define ITP_P_ISACTIVE(itp, iph) ((itp)->itp_flags & \
 541         (((itp)->itp_policy == (iph)) ? ITPF_P_ACTIVE : ITPF_I_ACTIVE))

 543 #define ITP_P_ISTUNNEL(itp, iph) ((itp)->itp_flags & \
 544         (((itp)->itp_policy == (iph)) ? ITPF_P_TUNNEL : ITPF_I_TUNNEL))

 546 #define ITP_P_ISPERPORT(itp, iph) ((itp)->itp_flags & \
 547         (((itp)->itp_policy == (iph)) ? ITPF_P_PER_PORT_SECURITY : \
 548         ITPF_I_PER_PORT_SECURITY))

 550 #define ITP_REFHOLD(itp) { \
 551         atomic_inc_32(&((itp)->itp_refcnt));    \
 551         atomic_add_32(&((itp)->itp_refcnt), 1); \
 552         ASSERT((itp)->itp_refcnt != 0); \
 553 }

 555 #define ITP_REFRELE(itp, ns) { \
 556         ASSERT((itp)->itp_refcnt != 0); \
 557         membar_exit(); \
 558         if (atomic_dec_32_nv(&((itp)->itp_refcnt)) == 0) \
 558         if (atomic_add_32_nv(&((itp)->itp_refcnt), -1) == 0) \
 559                 itp_free(itp, ns); \
 560 }
______unchanged_portion_omitted_

 575 /*
 576  * ipsid_t reference hold/release macros, just like ipsa versions.
 577  */

 579 #define IPSID_REFHOLD(ipsid) {                  \
 580         atomic_inc_32(&(ipsid)->ipsid_refcnt);  \
 580         atomic_add_32(&(ipsid)->ipsid_refcnt, 1);       \
 581         ASSERT((ipsid)->ipsid_refcnt != 0);     \
 582 }

 584 /*
 585  * Decrement the reference count on the ID.  Someone else will clean up
 586  * after us later.
 587  */

 589 #define IPSID_REFRELE(ipsid) {                                  \
 590         membar_exit();                                          \
 591         atomic_dec_32(&(ipsid)->ipsid_refcnt);          \
 591         atomic_add_32(&(ipsid)->ipsid_refcnt, -1);              \
 592 }
______unchanged_portion_omitted_
```

****
6362 Mon Jul 28 07:44:29 2014
new/usr/src/uts/common/inet/kssl/ksslimpl.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____unchanged_portion_omitted_

```
 125 #define KSSL_ENTRY_REFHOLD(kssl_entry) {                                \
 126         atomic_inc_32(&(kssl_entry)->ke_refcnt);                        \
 126         atomic_add_32(&(kssl_entry)->ke_refcnt, 1);                     \
 127         ASSERT((kssl_entry)->ke_refcnt != 0);                           \
 128 }

 130 #define KSSL_ENTRY_REFRELE(kssl_entry) {                                \
 131         ASSERT((kssl_entry)->ke_refcnt != 0);                           \
 132         membar_exit();                                                  \
 133         if (atomic_dec_32_nv(&(kssl_entry)->ke_refcnt) == 0) {  \
 133         if (atomic_add_32_nv(&(kssl_entry)->ke_refcnt, -1) == 0) {      \
 134                 kssl_free_entry((kssl_entry));                          \
 135         }                                                               \
 136 }
```

_____unchanged_portion_omitted_

****
70048 Mon Jul 28 07:44:29 2014
new/usr/src/uts/common/inet/nca/nca.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
 959 #define DCB_RD_EXIT(cpu) {                                          \
 960         uint32_t *rp = &nca_gv[cpu].dcb_readers;                    \
 961                                                                     \
 962         if (atomic_dec_32_nv(rp) == DCB_COUNT_USELOCK) {            \
 962         if (atomic_add_32_nv(rp, -1) == DCB_COUNT_USELOCK) {        \
 963                 mutex_enter(&nca_dcb_lock);                         \
 964                 if (CV_HAS_WAITERS(&nca_dcb_wait)) {                \
 965                         /* May be the last reader for this CPU */   \
 966                         cv_signal(&nca_dcb_wait);                   \
 967                 }                                                   \
 968                 mutex_exit(&nca_dcb_lock);                          \
 969         }                                                           \
 970 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   34521 Mon Jul 28 07:44:29 2014
new/usr/src/uts/common/inet/sadb.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 323 /*
 324  * ipsa_t address handling macros.  We want these to be inlined, and deal
 325  * with 32-bit words to avoid bcmp/bcopy calls.
 326  *
 327  * Assume we only have AF_INET and AF_INET6 addresses for now.  Also assume
 328  * that we have 32-bit alignment on everything.
 329  */
 330 #define IPSA_IS_ADDR_UNSPEC(addr, fam) ((((uint32_t *)(addr))[0] == 0) && \
 331         (((fam) == AF_INET) || (((uint32_t *)(addr))[3] == 0 && \
 332         ((uint32_t *)(addr))[2] == 0 && ((uint32_t *)(addr))[1] == 0)))
 333 #define IPSA_ARE_ADDR_EQUAL(addr1, addr2, fam) \
 334         ((((uint32_t *)(addr1))[0] == ((uint32_t *)(addr2))[0]) && \
 335         (((fam) == AF_INET) || \
 336         (((uint32_t *)(addr1))[3] == ((uint32_t *)(addr2))[3] && \
 337         ((uint32_t *)(addr1))[2] == ((uint32_t *)(addr2))[2] && \
 338         ((uint32_t *)(addr1))[1] == ((uint32_t *)(addr2))[1])))
 339 #define IPSA_COPY_ADDR(dstaddr, srcaddr, fam) { \
 340         ((uint32_t *)(dstaddr))[0] = ((uint32_t *)(srcaddr))[0]; \
 341         if ((fam) == AF_INET6) {\
 342                 ((uint32_t *)(dstaddr))[1] = ((uint32_t *)(srcaddr))[1]; \
 343                 ((uint32_t *)(dstaddr))[2] = ((uint32_t *)(srcaddr))[2]; \
 344                 ((uint32_t *)(dstaddr))[3] = ((uint32_t *)(srcaddr))[3]; } }

 346 /*
 347  * ipsa_t reference hold/release macros.
 348  *
 349  * If you have a pointer, you REFHOLD.  If you are releasing a pointer, you
 350  * REFRELE.  An ipsa_t that is newly inserted into the table should have
 351  * a reference count of 1 (for the table's pointer), plus 1 more for every
 352  * pointer that is referencing the ipsa_t.
 353  */

 355 #define IPSA_REFHOLD(ipsa) {                    \
 356         atomic_inc_32(&(ipsa)->ipsa_refcnt);    \
 356         atomic_add_32(&(ipsa)->ipsa_refcnt, 1); \
 357         ASSERT((ipsa)->ipsa_refcnt != 0);       \
 358 }

 360 /*
 361  * Decrement the reference count on the SA.
 362  * In architectures e.g sun4u, where atomic_add_32_nv is just
 363  * a cas, we need to maintain the right memory barrier semantics
 364  * as that of mutex_exit i.e all the loads and stores should complete
 365  * before the cas is executed. membar_exit() does that here.
 366  */

 368 #define IPSA_REFRELE(ipsa) {                                    \
 369         ASSERT((ipsa)->ipsa_refcnt != 0);                       \
 370         membar_exit();                                          \
 371         if (atomic_dec_32_nv(&(ipsa)->ipsa_refcnt) == 0)        \
 371         if (atomic_add_32_nv(&(ipsa)->ipsa_refcnt, -1) == 0)    \
 372                 ((ipsa)->ipsa_freefunc)(ipsa);                  \
 373 }
_____unchanged_portion_omitted_
```

****
58942 Mon Jul 28 07:44:30 2014
new/usr/src/uts/common/inet/sctp/sctp.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____unchanged_portion_omitted_

```
1721 /* Add another taskq for a new ill. */
1722 void
1723 sctp_inc_taskq(sctp_stack_t *sctps)
1724 {
1725         taskq_t *tq;
1726         char tq_name[TASKQ_NAMELEN];
1727         int thrs;
1728         int max_tasks;

1730         thrs = MIN(sctp_recvq_tq_thr_max, MAX(sctp_recvq_tq_thr_min,
1731             MAX(ncpus, boot_ncpus)));
1732         /*
1733          * Make sure that the maximum number of tasks is at least thrice as
1734          * large as the number of threads.
1735          */
1736         max_tasks = MAX(sctp_recvq_tq_task_min, thrs) * 3;

1738         mutex_enter(&sctps->sctps_rq_tq_lock);
1739         if (sctps->sctps_recvq_tq_list_cur_sz + 1 >
1740             sctps->sctps_recvq_tq_list_max_sz) {
1741                 mutex_exit(&sctps->sctps_rq_tq_lock);
1742                 cmn_err(CE_NOTE, "Cannot create more SCTP recvq taskq");
1743                 return;
1744         }

1746         (void) snprintf(tq_name, sizeof (tq_name), "sctp_rq_taskq_%d_%u",
1747             sctps->sctps_netstack->netstack_stackid,
1748             sctps->sctps_recvq_tq_list_cur_sz);
1749         tq = taskq_create(tq_name, thrs, minclsyspri, sctp_recvq_tq_task_min,
1750             max_tasks, TASKQ_PREPOPULATE);
1751         if (tq == NULL) {
1752                 mutex_exit(&sctps->sctps_rq_tq_lock);
1753                 cmn_err(CE_NOTE, "SCTP recvq taskq creation failed");
1754                 return;
1755         }
1756         ASSERT(sctps->sctps_recvq_tq_list[
1757             sctps->sctps_recvq_tq_list_cur_sz] == NULL);
1758         sctps->sctps_recvq_tq_list[sctps->sctps_recvq_tq_list_cur_sz] = tq;
1759         atomic_inc_32(&sctps->sctps_recvq_tq_list_cur_sz);
1759         atomic_add_32(&sctps->sctps_recvq_tq_list_cur_sz, 1);
1760         mutex_exit(&sctps->sctps_rq_tq_lock);
1761 }

1763 #ifdef DEBUG
1764 uint32_t recvq_loop_cnt = 0;
1765 uint32_t recvq_call = 0;
1766 #endif

1768 /*
1769  * Find the next recvq_tq to use.  This routine will go thru all the
1770  * taskqs until it can dispatch a job for the sctp.  If this fails,
1771  * it will create a new taskq and try it.
1772  */
1773 static boolean_t
1774 sctp_find_next_tq(sctp_t *sctp)
1775 {
1776         int next_tq, try;
1777         taskq_t *tq;
1778         sctp_stack_t    *sctps = sctp->sctp_sctps;
```



```
1780         /*
1781          * Note that since we don't hold a lock on sctp_rq_tq_lock for
1782          * performance reason, recvq_ta_list_cur_sz can be changed during
1783          * this loop.  The problem this will create is that the loop may
1784          * not have tried all the recvq_tq.  This should be OK.
1785          */
1786         next_tq = atomic_inc_32_nv(&sctps->sctps_recvq_tq_list_cur) %
1786         next_tq = atomic_add_32_nv(&sctps->sctps_recvq_tq_list_cur, 1) %
1787             sctps->sctps_recvq_tq_list_cur_sz;
1788         for (try = 0; try < sctps->sctps_recvq_tq_list_cur_sz; try++) {
1789                 tq = sctps->sctps_recvq_tq_list[next_tq];
1790                 if (taskq_dispatch(tq, sctp_process_recvq, sctp,
1791                     TQ_NOSLEEP) != NULL) {
1792                         sctp->sctp_recvq_tq = tq;
1793                         return (B_TRUE);
1794                 }
1795                 next_tq = (next_tq + 1) % sctps->sctps_recvq_tq_list_cur_sz;
1796         }

1798         /*
1799          * Create one more taskq and try it.  Note that sctp_inc_taskq()
1800          * may not have created another taskq if the number of recvq
1801          * taskqs is at the maximum.  We are probably in a pretty bad
1802          * shape if this actually happens...
1803          */
1804         sctp_inc_taskq(sctps);
1805         tq = sctps->sctps_recvq_tq_list[sctps->sctps_recvq_tq_list_cur_sz - 1];
1806         if (taskq_dispatch(tq, sctp_process_recvq, sctp, TQ_NOSLEEP) != NULL) {
1807                 sctp->sctp_recvq_tq = tq;
1808                 return (B_TRUE);
1809         }
1810         SCTP_KSTAT(sctps, sctp_find_next_tq);
1811         return (B_FALSE);
1812 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   62085 Mon Jul 28 07:44:30 2014
new/usr/src/uts/common/inet/sctp/sctp_addr.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2004, 2010, Oracle and/or its affiliates. All rights reserved.
 */

#include <sys/types.h>
#include <sys/systm.h>
#include <sys/stream.h>
#include <sys/cmn_err.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/kmem.h>
#include <sys/socket.h>
#include <sys/sysmacros.h>
#include <sys/list.h>

#include <netinet/in.h>
#include <netinet/ip6.h>
#include <netinet/sctp.h>

#include <inet/common.h>
#include <inet/ip.h>
#include <inet/ip6.h>
#include <inet/ip_ire.h>
#include <inet/ip_if.h>
#include <inet/ipclassifier.h>
#include <inet/sctp_ip.h>
#include "sctp_impl.h"
#include "sctp_addr.h"

static void             sctp_ipif_inactive(sctp_ipif_t *);
static sctp_ipif_t      *sctp_lookup_ipif_addr(in6_addr_t *, boolean_t,
                            zoneid_t, boolean_t, uint_t, uint_t, boolean_t,
                            sctp_stack_t *);
static int              sctp_get_all_ipifs(sctp_t *, int);
static int              sctp_ipif_hash_insert(sctp_t *, sctp_ipif_t *, int,
                            boolean_t, boolean_t);
static void             sctp_ipif_hash_remove(sctp_t *, sctp_ipif_t *,
                            boolean_t);
static void             sctp_fix_saddr(sctp_t *, in6_addr_t *);
static int              sctp_compare_ipif_list(sctp_ipif_hash_t *,
                            sctp_ipif_hash_t *);
```



```
static int              sctp_copy_ipifs(sctp_ipif_hash_t *, sctp_t *, int);

#define SCTP_ADDR4_HASH(addr)   \
        (((addr) ^ ((addr) >> 8) ^ ((addr) >> 16) ^ ((addr) >> 24)) &   \
        (SCTP_IPIF_HASH - 1))

#define SCTP_ADDR6_HASH(addr)   \
        (((addr).s6_addr32[3] ^                                         \
        ((addr).s6_addr32[3] ^ (addr).s6_addr32[2]) >> 12)) &           \
        (SCTP_IPIF_HASH - 1))

#define SCTP_IPIF_ADDR_HASH(addr, isv6)                                 \
        ((isv6) ? SCTP_ADDR6_HASH((addr)) :                             \
        SCTP_ADDR4_HASH((addr)._S6_un._S6_u32[3]))

#define SCTP_IPIF_USABLE(sctp_ipif_state)       \
        ((sctp_ipif_state) == SCTP_IPIFS_UP ||  \
        (sctp_ipif_state) ==  SCTP_IPIFS_DOWN)

#define SCTP_IPIF_DISCARD(sctp_ipif_flags)      \
        ((sctp_ipif_flags) & (IPIF_PRIVATE | IPIF_DEPRECATED))

#define SCTP_IS_IPIF_LOOPBACK(ipif)             \
        ((ipif)->sctp_ipif_ill->sctp_ill_flags & PHYI_LOOPBACK)

#define SCTP_IS_IPIF_LINKLOCAL(ipif)            \
        ((ipif)->sctp_ipif_isv6 &&              \
        IN6_IS_ADDR_LINKLOCAL(&(ipif)->sctp_ipif_saddr))

#define SCTP_UNSUPP_AF(ipif, supp_af)   \
        ((!(ipif)->sctp_ipif_isv6 && !((supp_af) & PARM_SUPP_V4)) ||    \
        ((ipif)->sctp_ipif_isv6 && !((supp_af) & PARM_SUPP_V6)))

#define SCTP_IPIF_ZONE_MATCH(sctp, ipif)                                \
        IPCL_ZONE_MATCH((sctp)->sctp_connp, (ipif)->sctp_ipif_zoneid)

#define SCTP_ILL_HASH_FN(index)         ((index) % SCTP_ILL_HASH)
#define SCTP_ILL_TO_PHYINDEX(ill)       ((ill)->ill_phyint->phyint_ifindex)

/*
 * SCTP Interface list manipulation functions, locking used.
 */

/*
 * Delete an SCTP IPIF from the list if the refcount goes to 0 and it is
 * marked as condemned. Also, check if the ILL needs to go away.
 */
static void
sctp_ipif_inactive(sctp_ipif_t *sctp_ipif)
{
        sctp_ill_t      *sctp_ill;
        uint_t          hindex;
        uint_t          ill_index;
        sctp_stack_t    *sctps = sctp_ipif->sctp_ipif_ill->
            sctp_ill_netstack->netstack_sctp;

        rw_enter(&sctps->sctps_g_ills_lock, RW_READER);
        rw_enter(&sctps->sctps_g_ipifs_lock, RW_WRITER);

        hindex = SCTP_IPIF_ADDR_HASH(sctp_ipif->sctp_ipif_saddr,
            sctp_ipif->sctp_ipif_isv6);

        sctp_ill = sctp_ipif->sctp_ipif_ill;
        ASSERT(sctp_ill != NULL);
        ill_index = SCTP_ILL_HASH_FN(sctp_ill->sctp_ill_index);
        if (sctp_ipif->sctp_ipif_state != SCTP_IPIFS_CONDEMNED ||
```

```
128             sctp_ipif->sctp_ipif_refcnt != 0) {
129                 rw_exit(&sctps->sctps_g_ipifs_lock);
130                 rw_exit(&sctps->sctps_g_ills_lock);
131                 return;
132         }
133         list_remove(&sctps->sctps_g_ipifs[hindex].sctp_ipif_list,
134             sctp_ipif);
135         sctps->sctps_g_ipifs[hindex].ipif_count--;
136         sctps->sctps_g_ipifs_count--;
137         rw_destroy(&sctp_ipif->sctp_ipif_lock);
138         kmem_free(sctp_ipif, sizeof (sctp_ipif_t));

140         (void) atomic_dec_32_nv(&sctp_ill->sctp_ill_ipifcnt);
140         (void) atomic_add_32_nv(&sctp_ill->sctp_ill_ipifcnt, -1);
141         if (rw_tryupgrade(&sctps->sctps_g_ills_lock) != 0) {
142                 rw_downgrade(&sctps->sctps_g_ipifs_lock);
143                 if (sctp_ill->sctp_ill_ipifcnt == 0 &&
144                     sctp_ill->sctp_ill_state == SCTP_ILLS_CONDEMNED) {
145                         list_remove(&sctps->sctps_g_ills[ill_index].
146                             sctp_ill_list, (void *)sctp_ill);
147                         sctps->sctps_g_ills[ill_index].ill_count--;
148                         sctps->sctps_ills_count--;
149                         kmem_free(sctp_ill->sctp_ill_name,
150                             sctp_ill->sctp_ill_name_length);
151                         kmem_free(sctp_ill, sizeof (sctp_ill_t));
152                 }
153         }
154         rw_exit(&sctps->sctps_g_ipifs_lock);
155         rw_exit(&sctps->sctps_g_ills_lock);
156 }
```

*unchanged_portion_omitted_*

```
829 /* move ipif from f_ill to t_ill */
830 void
831 sctp_move_ipif(ipif_t *ipif, ill_t *f_ill, ill_t *t_ill)
832 {
833         sctp_ill_t      *fsctp_ill = NULL;
834         sctp_ill_t      *tsctp_ill = NULL;
835         sctp_ipif_t     *sctp_ipif;
836         uint_t          hindex;
837         int             i;
838         netstack_t      *ns = ipif->ipif_ill->ill_ipst->ips_netstack;
839         sctp_stack_t    *sctps = ns->netstack_sctp;

841         rw_enter(&sctps->sctps_g_ills_lock, RW_READER);
842         rw_enter(&sctps->sctps_g_ipifs_lock, RW_READER);

844         hindex = SCTP_ILL_HASH_FN(SCTP_ILL_TO_PHYINDEX(f_ill));
845         fsctp_ill = list_head(&sctps->sctps_g_ills[hindex].sctp_ill_list);
846         for (i = 0; i < sctps->sctps_g_ills[hindex].ill_count; i++) {
847                 if (fsctp_ill->sctp_ill_index == SCTP_ILL_TO_PHYINDEX(f_ill) &&
848                     fsctp_ill->sctp_ill_isv6 == f_ill->ill_isv6) {
849                         break;
850                 }
851                 fsctp_ill = list_next(
852                     &sctps->sctps_g_ills[hindex].sctp_ill_list, fsctp_ill);
853         }

855         hindex = SCTP_ILL_HASH_FN(SCTP_ILL_TO_PHYINDEX(t_ill));
856         tsctp_ill = list_head(&sctps->sctps_g_ills[hindex].sctp_ill_list);
857         for (i = 0; i < sctps->sctps_g_ills[hindex].ill_count; i++) {
858                 if (tsctp_ill->sctp_ill_index == SCTP_ILL_TO_PHYINDEX(t_ill) &&
859                     tsctp_ill->sctp_ill_isv6 == t_ill->ill_isv6) {
860                         break;
861                 }
862                 tsctp_ill = list_next(
```



```
863                     &sctps->sctps_g_ills[hindex].sctp_ill_list, tsctp_ill);
864         }

866         hindex = SCTP_IPIF_ADDR_HASH(ipif->ipif_v6lcl_addr,
867             ipif->ipif_ill->ill_isv6);
868         sctp_ipif = list_head(&sctps->sctps_g_ipifs[hindex].sctp_ipif_list);
869         for (i = 0; i < sctps->sctps_g_ipifs[hindex].ipif_count; i++) {
870                 if (sctp_ipif->sctp_ipif_id == ipif->ipif_seqid)
871                         break;
872                 sctp_ipif = list_next(
873                     &sctps->sctps_g_ipifs[hindex].sctp_ipif_list, sctp_ipif);
874         }
875         /* Should be an ASSERT? */
876         if (fsctp_ill == NULL || tsctp_ill == NULL || sctp_ipif == NULL) {
877                 ip1dbg(("sctp_move_ipif: error moving ipif %p from %p to %p\n",
878                     (void *)ipif, (void *)f_ill, (void *)t_ill));
879                 rw_exit(&sctps->sctps_g_ipifs_lock);
880                 rw_exit(&sctps->sctps_g_ills_lock);
881                 return;
882         }
883         rw_enter(&sctp_ipif->sctp_ipif_lock, RW_WRITER);
884         ASSERT(sctp_ipif->sctp_ipif_ill == fsctp_ill);
885         sctp_ipif->sctp_ipif_ill = tsctp_ill;
886         rw_exit(&sctp_ipif->sctp_ipif_lock);
887         (void) atomic_dec_32_nv(&fsctp_ill->sctp_ill_ipifcnt);
888         atomic_inc_32(&tsctp_ill->sctp_ill_ipifcnt);
887         (void) atomic_add_32_nv(&fsctp_ill->sctp_ill_ipifcnt, -1);
888         atomic_add_32(&tsctp_ill->sctp_ill_ipifcnt, 1);
889         rw_exit(&sctps->sctps_g_ipifs_lock);
890         rw_exit(&sctps->sctps_g_ills_lock);
891 }
```

*unchanged_portion_omitted_*

```
994 /*
995  * Insert a new SCTP ipif using 'ipif'. v6addr is the address that existed
996  * prior to the current address in 'ipif'. Only when an existing address
997  * is changed on an IPIF, will v6addr be specified. If the IPIF already
998  * exists in the global SCTP ipif table, then we either removed it, if
999  * it doesn't have any existing reference, or mark it condemned otherwise.
1000 * If an address is being brought up (IPIF_UP), then we need to scan
1001 * the SCTP list to check if there is any SCTP that points to the *same*
1002 * address on a different SCTP ipif and update in that case.
1003 */
1004 void
1005 sctp_update_ipif_addr(ipif_t *ipif, in6_addr_t v6addr)
1006 {
1007         ill_t           *ill = ipif->ipif_ill;
1008         int             i;
1009         sctp_ill_t      *sctp_ill;
1010         sctp_ill_t      *osctp_ill;
1011         sctp_ipif_t     *sctp_ipif = NULL;
1012         sctp_ipif_t     *osctp_ipif = NULL;
1013         uint_t          ill_index;
1014         int             hindex;
1015         sctp_stack_t    *sctps;

1017         sctps = ipif->ipif_ill->ill_ipst->ips_netstack->netstack_sctp;

1019         /* Index for new address */
1020         hindex = SCTP_IPIF_ADDR_HASH(ipif->ipif_v6lcl_addr, ill->ill_isv6);

1022         /*
1023          * The address on this IPIF is changing, we need to look for
1024          * this old address and mark it condemned, before creating
1025          * one for the new address.
1026          */
```

```
1027         osctp_ipif = sctp_lookup_ipif_addr(&v6addr, B_FALSE,
1028             ipif->ipif_zoneid, B_TRUE, SCTP_ILL_TO_PHYINDEX(ill),
1029             ipif->ipif_seqid, B_FALSE, sctps);

1031         rw_enter(&sctps->sctps_g_ills_lock, RW_READER);
1032         rw_enter(&sctps->sctps_g_ipifs_lock, RW_WRITER);

1034         ill_index = SCTP_ILL_HASH_FN(SCTP_ILL_TO_PHYINDEX(ill));
1035         sctp_ill = list_head(&sctps->sctps_g_ills[ill_index].sctp_ill_list);
1036         for (i = 0; i < sctps->sctps_g_ills[ill_index].ill_count; i++) {
1037                 if (sctp_ill->sctp_ill_index == SCTP_ILL_TO_PHYINDEX(ill) &&
1038                     sctp_ill->sctp_ill_isv6 == ill->ill_isv6) {
1039                         break;
1040                 }
1041                 sctp_ill = list_next(
1042                     &sctps->sctps_g_ills[ill_index].sctp_ill_list, sctp_ill);
1043         }

1045         if (sctp_ill == NULL) {
1046                 ip1dbg(("sctp_update_ipif_addr: ill not found ..\n"));
1047                 rw_exit(&sctps->sctps_g_ipifs_lock);
1048                 rw_exit(&sctps->sctps_g_ills_lock);
1049                 return;
1050         }

1052         if (osctp_ipif != NULL) {

1054                 /* The address is the same? */
1055                 if (IN6_ARE_ADDR_EQUAL(&ipif->ipif_v6lcl_addr, &v6addr)) {
1056                         boolean_t       chk_n_updt = B_FALSE;

1058                         rw_downgrade(&sctps->sctps_g_ipifs_lock);
1059                         rw_enter(&osctp_ipif->sctp_ipif_lock, RW_WRITER);
1060                         if (ipif->ipif_flags & IPIF_UP &&
1061                             osctp_ipif->sctp_ipif_state != SCTP_IPIFS_UP) {
1062                                 osctp_ipif->sctp_ipif_state = SCTP_IPIFS_UP;
1063                                 chk_n_updt = B_TRUE;
1064                         } else {
1065                                 osctp_ipif->sctp_ipif_state = SCTP_IPIFS_DOWN;
1066                         }
1067                         osctp_ipif->sctp_ipif_flags = ipif->ipif_flags;
1068                         rw_exit(&osctp_ipif->sctp_ipif_lock);
1069                         if (chk_n_updt) {
1070                                 sctp_chk_and_updt_saddr(hindex, osctp_ipif,
1071                                     sctps);
1072                         }
1073                         rw_exit(&sctps->sctps_g_ipifs_lock);
1074                         rw_exit(&sctps->sctps_g_ills_lock);
1075                         return;
1076                 }
1077                 /*
1078                  * We are effectively removing this address from the ILL.
1079                  */
1080                 if (osctp_ipif->sctp_ipif_refcnt != 0) {
1081                         osctp_ipif->sctp_ipif_state = SCTP_IPIFS_CONDEMNED;
1082                 } else {
1083                         list_t          *ipif_list;
1084                         int             ohindex;

1086                         osctp_ill = osctp_ipif->sctp_ipif_ill;
1087                         /* hash index for the old one */
1088                         ohindex = SCTP_IPIF_ADDR_HASH(
1089                             osctp_ipif->sctp_ipif_saddr,
1090                             osctp_ipif->sctp_ipif_isv6);

1092                         ipif_list =
```



```
1093                             &sctps->sctps_g_ipifs[ohindex].sctp_ipif_list;

1095                         list_remove(ipif_list, (void *)osctp_ipif);
1096                         sctps->sctps_g_ipifs[ohindex].ipif_count--;
1097                         sctps->sctps_g_ipifs_count--;
1098                         rw_destroy(&osctp_ipif->sctp_ipif_lock);
1099                         kmem_free(osctp_ipif, sizeof (sctp_ipif_t));
1100                         (void) atomic_dec_32_nv(&osctp_ill->sctp_ill_ipifcnt);
1100                         (void) atomic_add_32_nv(&osctp_ill->sctp_ill_ipifcnt,
1101                             -1);
1101                 }
1102         }

1104         sctp_ipif = kmem_zalloc(sizeof (sctp_ipif_t), KM_NOSLEEP);
1105         /* Try again? */
1106         if (sctp_ipif == NULL) {
1107                 cmn_err(CE_WARN, "sctp_update_ipif_addr: error adding "
1108                     "IPIF %p to SCTP's IPIF list", (void *)ipif);
1109                 rw_exit(&sctps->sctps_g_ipifs_lock);
1110                 rw_exit(&sctps->sctps_g_ills_lock);
1111                 return;
1112         }
1113         sctps->sctps_g_ipifs_count++;
1114         rw_init(&sctp_ipif->sctp_ipif_lock, NULL, RW_DEFAULT, NULL);
1115         sctp_ipif->sctp_ipif_saddr = ipif->ipif_v6lcl_addr;
1116         sctp_ipif->sctp_ipif_ill = sctp_ill;
1117         sctp_ipif->sctp_ipif_isv6 = ill->ill_isv6;
1118         sctp_ipif->sctp_ipif_zoneid = ipif->ipif_zoneid;
1119         sctp_ipif->sctp_ipif_id = ipif->ipif_seqid;
1120         if (ipif->ipif_flags & IPIF_UP)
1121                 sctp_ipif->sctp_ipif_state = SCTP_IPIFS_UP;
1122         else
1123                 sctp_ipif->sctp_ipif_state = SCTP_IPIFS_DOWN;
1124         sctp_ipif->sctp_ipif_flags = ipif->ipif_flags;
1125         /*
1126          * We add it to the head so that it is quicker to find good/recent
1127          * additions.
1128          */
1129         list_insert_head(&sctps->sctps_g_ipifs[hindex].sctp_ipif_list,
1130             (void *)sctp_ipif);
1131         sctps->sctps_g_ipifs[hindex].ipif_count++;
1132         atomic_inc_32(&sctp_ill->sctp_ill_ipifcnt);
1133         atomic_add_32(&sctp_ill->sctp_ill_ipifcnt, 1);
1133         if (sctp_ipif->sctp_ipif_state == SCTP_IPIFS_UP)
1134                 sctp_chk_and_updt_saddr(hindex, sctp_ipif, sctps);
1135         rw_exit(&sctps->sctps_g_ipifs_lock);
1136         rw_exit(&sctps->sctps_g_ills_lock);
1137 }

1139 /* Insert, Remove,  Mark up or Mark down the ipif */
1140 void
1141 sctp_update_ipif(ipif_t *ipif, int op)
1142 {
1143         ill_t           *ill = ipif->ipif_ill;
1144         int             i;
1145         sctp_ill_t      *sctp_ill;
1146         sctp_ipif_t     *sctp_ipif;
1147         uint_t          ill_index;
1148         uint_t          hindex;
1149         netstack_t      *ns = ipif->ipif_ill->ill_ipst->ips_netstack;
1150         sctp_stack_t    *sctps = ns->netstack_sctp;

1152         ip2dbg(("sctp_update_ipif: %s %d\n", ill->ill_name, ipif->ipif_seqid));

1154         rw_enter(&sctps->sctps_g_ills_lock, RW_READER);
1155         rw_enter(&sctps->sctps_g_ipifs_lock, RW_WRITER);
```

```
1157         ill_index = SCTP_ILL_HASH_FN(SCTP_ILL_TO_PHYINDEX(ill));
1158         sctp_ill = list_head(&sctps->sctps_g_ills[ill_index].sctp_ill_list);
1159         for (i = 0; i < sctps->sctps_g_ills[ill_index].ill_count; i++) {
1160                 if (sctp_ill->sctp_ill_index == SCTP_ILL_TO_PHYINDEX(ill) &&
1161                     sctp_ill->sctp_ill_isv6 == ill->ill_isv6) {
1162                         break;
1163                 }
1164                 sctp_ill = list_next(
1165                     &sctps->sctps_g_ills[ill_index].sctp_ill_list, sctp_ill);
1166         }
1167         if (sctp_ill == NULL) {
1168                 rw_exit(&sctps->sctps_g_ipifs_lock);
1169                 rw_exit(&sctps->sctps_g_ills_lock);
1170                 return;
1171         }

1173         hindex = SCTP_IPIF_ADDR_HASH(ipif->ipif_v6lcl_addr,
1174             ipif->ipif_ill->ill_isv6);
1175         sctp_ipif = list_head(&sctps->sctps_g_ipifs[hindex].sctp_ipif_list);
1176         for (i = 0; i < sctps->sctps_g_ipifs[hindex].ipif_count; i++) {
1177                 if (sctp_ipif->sctp_ipif_id == ipif->ipif_seqid) {
1178                         ASSERT(IN6_ARE_ADDR_EQUAL(&sctp_ipif->sctp_ipif_saddr,
1179                             &ipif->ipif_v6lcl_addr));
1180                         break;
1181                 }
1182                 sctp_ipif = list_next(
1183                     &sctps->sctps_g_ipifs[hindex].sctp_ipif_list,
1184                     sctp_ipif);
1185         }
1186         if (sctp_ipif == NULL) {
1187                 ip1dbg(("sctp_update_ipif: null sctp_ipif for %d\n", op));
1188                 rw_exit(&sctps->sctps_g_ipifs_lock);
1189                 rw_exit(&sctps->sctps_g_ills_lock);
1190                 return;
1191         }
1192         ASSERT(sctp_ill == sctp_ipif->sctp_ipif_ill);
1193         switch (op) {
1194         case SCTP_IPIF_REMOVE:
1195         {
1196                 list_t          *ipif_list;
1197                 list_t          *ill_list;

1199                 ill_list = &sctps->sctps_g_ills[ill_index].sctp_ill_list;
1200                 ipif_list = &sctps->sctps_g_ipifs[hindex].sctp_ipif_list;
1201                 if (sctp_ipif->sctp_ipif_refcnt != 0) {
1202                         sctp_ipif->sctp_ipif_state = SCTP_IPIFS_CONDEMNED;
1203                         rw_exit(&sctps->sctps_g_ipifs_lock);
1204                         rw_exit(&sctps->sctps_g_ills_lock);
1205                         return;
1206                 }
1207                 list_remove(ipif_list, (void *)sctp_ipif);
1208                 sctps->sctps_g_ipifs[hindex].ipif_count--;
1209                 sctps->sctps_g_ipifs_count--;
1210                 rw_destroy(&sctp_ipif->sctp_ipif_lock);
1211                 kmem_free(sctp_ipif, sizeof (sctp_ipif_t));
1212                 (void) atomic_dec_32_nv(&sctp_ill->sctp_ill_ipifcnt);
1213                 (void) atomic_add_32_nv(&sctp_ill->sctp_ill_ipifcnt, -1);
1213                 if (rw_tryupgrade(&sctps->sctps_g_ills_lock) != 0) {
1214                         rw_downgrade(&sctps->sctps_g_ipifs_lock);
1215                         if (sctp_ill->sctp_ill_ipifcnt == 0 &&
1216                             sctp_ill->sctp_ill_state == SCTP_ILLS_CONDEMNED) {
1217                                 list_remove(ill_list, (void *)sctp_ill);
1218                                 sctps->sctps_ills_count--;
1219                                 sctps->sctps_g_ills[ill_index].ill_count--;
1220                                 kmem_free(sctp_ill->sctp_ill_name,
```



```
1221                                     sctp_ill->sctp_ill_name_length);
1222                                 kmem_free(sctp_ill, sizeof (sctp_ill_t));
1223                         }
1224                 }
1225                 break;
1226         }

1228         case SCTP_IPIF_UP:

1230                 rw_downgrade(&sctps->sctps_g_ipifs_lock);
1231                 rw_enter(&sctp_ipif->sctp_ipif_lock, RW_WRITER);
1232                 sctp_ipif->sctp_ipif_state = SCTP_IPIFS_UP;
1233                 sctp_ipif->sctp_ipif_flags = ipif->ipif_flags;
1234                 rw_exit(&sctp_ipif->sctp_ipif_lock);
1235                 sctp_chk_and_updt_saddr(hindex, sctp_ipif,
1236                     ipif->ipif_ill->ill_ipst->ips_netstack->netstack_sctp);

1238                 break;

1240         case SCTP_IPIF_UPDATE:

1242                 rw_downgrade(&sctps->sctps_g_ipifs_lock);
1243                 rw_enter(&sctp_ipif->sctp_ipif_lock, RW_WRITER);
1244                 sctp_ipif->sctp_ipif_zoneid = ipif->ipif_zoneid;
1245                 sctp_ipif->sctp_ipif_flags = ipif->ipif_flags;
1246                 rw_exit(&sctp_ipif->sctp_ipif_lock);

1248                 break;

1250         case SCTP_IPIF_DOWN:

1252                 rw_downgrade(&sctps->sctps_g_ipifs_lock);
1253                 rw_enter(&sctp_ipif->sctp_ipif_lock, RW_WRITER);
1254                 sctp_ipif->sctp_ipif_state = SCTP_IPIFS_DOWN;
1255                 sctp_ipif->sctp_ipif_flags = ipif->ipif_flags;
1256                 rw_exit(&sctp_ipif->sctp_ipif_lock);

1258                 break;
1259         }
1260         rw_exit(&sctps->sctps_g_ipifs_lock);
1261         rw_exit(&sctps->sctps_g_ills_lock);
1262 }
______unchanged_portion_omitted_

2013 static void
2014 sctp_free_ipifs(sctp_stack_t *sctps)
2015 {
2016         int                     i;
2017         int                     l;
2018         sctp_ipif_t     *sctp_ipif;
2019         sctp_ill_t      *sctp_ill;

2021         if (sctps->sctps_g_ipifs_count == 0)
2022                 return;

2024         for (i = 0; i < SCTP_IPIF_HASH; i++) {
2025                 sctp_ipif = list_tail(&sctps->sctps_g_ipifs[i].sctp_ipif_list);
2026                 for (l = 0; l < sctps->sctps_g_ipifs[i].ipif_count; l++) {
2027                         sctp_ill = sctp_ipif->sctp_ipif_ill;

2029                         list_remove(&sctps->sctps_g_ipifs[i].sctp_ipif_list,
2030                             sctp_ipif);
2031                         sctps->sctps_g_ipifs_count--;
2032                         (void) atomic_dec_32_nv(&sctp_ill->sctp_ill_ipifcnt);
2033                         (void) atomic_add_32_nv(&sctp_ill->sctp_ill_ipifcnt,
2034                             -1);
```

```
2033                        kmem_free(sctp_ipif, sizeof (sctp_ipif_t));
2034                        sctp_ipif =
2035                            list_tail(&sctps->sctps_g_ipifs[i].sctp_ipif_list);
2036                }
2037                sctps->sctps_g_ipifs[i].ipif_count = 0;
2038        }
2039        ASSERT(sctps->sctps_g_ipifs_count == 0);
2040 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   18324 Mon Jul 28 07:44:30 2014
new/usr/src/uts/common/inet/sctp/sctp_conn.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

/* Process the COOKIE packet, mp, directed at the listener 'sctp' */
sctp_t *
sctp_conn_request(sctp_t *sctp, mblk_t *mp, uint_t ifindex, uint_t ip_hdr_len,
    sctp_init_chunk_t *iack, ip_recv_attr_t *ira)
{
        sctp_t  *eager;
        ip6_t   *ip6h;
        int     err;
        conn_t  *connp, *econnp;
        sctp_stack_t    *sctps;
        cred_t          *cr;
        pid_t           cpid;
        in6_addr_t      faddr, laddr;
        ip_xmit_attr_t  *ixa;
        sctp_listen_cnt_t *slc = sctp->sctp_listen_cnt;
        boolean_t       slc_set = B_FALSE;

        /*
         * No need to check for duplicate as this is the listener
         * and we are holding the lock.  This means that no new
         * connection can be created out of it.  And since the
         * fanout already done cannot find a match, it means that
         * there is no duplicate.
         */
        ASSERT(OK_32PTR(mp->b_rptr));

        connp = sctp->sctp_connp;
        sctps = sctp->sctp_sctps;

        /*
         * Enforce the limit set on the number of connections per listener.
         * Note that tlc_cnt starts with 1.  So need to add 1 to tlc_max
         * for comparison.
         */
        if (slc != NULL) {
                int64_t now;

                if (atomic_inc_32_nv(&slc->slc_cnt) > slc->slc_max + 1) {
                if (atomic_add_32_nv(&slc->slc_cnt, 1) > slc->slc_max + 1) {
                        now = ddi_get_lbolt64();
                        atomic_dec_32(&slc->slc_cnt);
                        atomic_add_32(&slc->slc_cnt, -1);
                        SCTP_KSTAT(sctps, sctp_listen_cnt_drop);
                        slc->slc_drop++;
                        if (now - slc->slc_report_time >
                            MSEC_TO_TICK(SCTP_SLC_REPORT_INTERVAL)) {
                                zcmn_err(connp->conn_zoneid, CE_WARN,
                                    "SCTP listener (port %d) association max "
                                    "(%u) reached: %u attempts dropped total\n",
                                    ntohs(connp->conn_lport),
                                    slc->slc_max, slc->slc_drop);
                                slc->slc_report_time = now;
                        }
                        return (NULL);
                }
                slc_set = B_TRUE;
        }

        if ((eager = sctp_create_eager(sctp)) == NULL) {
```



```
                if (slc_set)
                        atomic_dec_32(&slc->slc_cnt);
                        atomic_add_32(&slc->slc_cnt, -1);
                return (NULL);
        }
        econnp = eager->sctp_connp;

        if (connp->conn_policy != NULL) {
                /* Inherit the policy from the listener; use actions from ira */
                if (!ip_ipsec_policy_inherit(econnp, connp, ira)) {
                        sctp_close_eager(eager);
                        SCTPS_BUMP_MIB(sctps, sctpListenDrop);
                        return (NULL);
                }
        }

        ip6h = (ip6_t *)mp->b_rptr;
        if (ira->ira_flags & IXAF_IS_IPV4) {
                ipha_t  *ipha;

                ipha = (ipha_t *)ip6h;
                IN6_IPADDR_TO_V4MAPPED(ipha->ipha_dst, &laddr);
                IN6_IPADDR_TO_V4MAPPED(ipha->ipha_src, &faddr);
        } else {
                laddr = ip6h->ip6_dst;
                faddr = ip6h->ip6_src;
        }

        if (ira->ira_flags & IRAF_IPSEC_SECURE) {
                /*
                 * XXX need to fix the cached policy issue here.
                 * We temporarily set the conn_laddr/conn_faddr here so
                 * that IPsec can use it for the latched policy
                 * selector.  This is obvioursly wrong as SCTP can
                 * use different addresses...
                 */
                econnp->conn_laddr_v6 = laddr;
                econnp->conn_faddr_v6 = faddr;
                econnp->conn_saddr_v6 = laddr;
        }
        if (ipsec_conn_cache_policy(econnp,
            (ira->ira_flags & IRAF_IS_IPV4) != 0) != 0) {
                sctp_close_eager(eager);
                SCTPS_BUMP_MIB(sctps, sctpListenDrop);
                return (NULL);
        }

        /* Save for getpeerucred */
        cr = ira->ira_cred;
        cpid = ira->ira_cpid;

        if (is_system_labeled()) {
                ip_xmit_attr_t *ixa = econnp->conn_ixa;

                ASSERT(ira->ira_tsl != NULL);

                /* Discard any old label */
                if (ixa->ixa_free_flags & IXA_FREE_TSL) {
                        ASSERT(ixa->ixa_tsl != NULL);
                        label_rele(ixa->ixa_tsl);
                        ixa->ixa_free_flags &= ~IXA_FREE_TSL;
                        ixa->ixa_tsl = NULL;
                }

                if ((connp->conn_mlp_type != mlptSingle ||
                    connp->conn_mac_mode != CONN_MAC_DEFAULT) &&
```

```
256                     ira->ira_tsl != NULL) {
257                         /*
258                          * If this is an MLP connection or a MAC-Exempt
259                          * connection with an unlabeled node, packets are to be
260                          * exchanged using the security label of the received
261                          * Cookie packet instead of the server application's
262                          * label.
263                          * tsol_check_dest called from ip_set_destination
264                          * might later update TSF_UNLABELED by replacing
265                          * ixa_tsl with a new label.
266                          */
267                         label_hold(ira->ira_tsl);
268                         ip_xmit_attr_replace_tsl(ixa, ira->ira_tsl);
269                 } else {
270                         ixa->ixa_tsl = crgetlabel(econnp->conn_cred);
271                 }
272         }

274         err = sctp_accept_comm(sctp, eager, mp, ip_hdr_len, iack);
275         if (err != 0) {
276                 sctp_close_eager(eager);
277                 SCTPS_BUMP_MIB(sctps, sctpListenDrop);
278                 return (NULL);
279         }

281         ASSERT(eager->sctp_current->sf_ixa != NULL);

283         ixa = eager->sctp_current->sf_ixa;
284         if (!(ira->ira_flags & IXAF_IS_IPV4)) {
285                 ASSERT(!(ixa->ixa_flags & IXAF_IS_IPV4));

287                 if (IN6_IS_ADDR_LINKLOCAL(&ip6h->ip6_src) ||
288                     IN6_IS_ADDR_LINKLOCAL(&ip6h->ip6_dst)) {
289                         eager->sctp_linklocal = 1;

291                         ixa->ixa_flags |= IXAF_SCOPEID_SET;
292                         ixa->ixa_scopeid = ifindex;
293                         econnp->conn_incoming_ifindex = ifindex;
294                 }
295         }

297         /*
298          * On a clustered note send this notification to the clustering
299          * subsystem.
300          */
301         if (cl_sctp_connect != NULL) {
302                 uchar_t *slist;
303                 uchar_t *flist;
304                 size_t  fsize;
305                 size_t  ssize;

307                 fsize = sizeof (in6_addr_t) * eager->sctp_nfaddrs;
308                 ssize = sizeof (in6_addr_t) * eager->sctp_nsaddrs;
309                 slist = kmem_alloc(ssize, KM_NOSLEEP);
310                 flist = kmem_alloc(fsize, KM_NOSLEEP);
311                 if (slist == NULL || flist == NULL) {
312                         if (slist != NULL)
313                                 kmem_free(slist, ssize);
314                         if (flist != NULL)
315                                 kmem_free(flist, fsize);
316                         sctp_close_eager(eager);
317                         SCTPS_BUMP_MIB(sctps, sctpListenDrop);
318                         SCTP_KSTAT(sctps, sctp_cl_connect);
319                         return (NULL);
320                 }
321                 /* The clustering module frees these list */
```



```
322                 sctp_get_saddr_list(eager, slist, ssize);
323                 sctp_get_faddr_list(eager, flist, fsize);
324                 (*cl_sctp_connect)(econnp->conn_family, slist,
325                     eager->sctp_nsaddrs, econnp->conn_lport, flist,
326                     eager->sctp_nfaddrs, econnp->conn_fport, B_FALSE,
327                     (cl_sctp_handle_t)eager);
328         }

330         /* Connection established, so send up the conn_ind */
331         if ((eager->sctp_ulpd = sctp->sctp_ulp_newconn(sctp->sctp_ulpd,
332             (sock_lower_handle_t)eager, NULL, cr, cpid,
333             &eager->sctp_upcalls)) == NULL) {
334                 sctp_close_eager(eager);
335                 SCTPS_BUMP_MIB(sctps, sctpListenDrop);
336                 return (NULL);
337         }
338         ASSERT(SCTP_IS_DETACHED(eager));
339         eager->sctp_detached = B_FALSE;
340         return (eager);
341 }
```

_____***unchanged_portion_omitted_***

```
**********
   46234 Mon Jul 28 07:44:30 2014
new/usr/src/uts/common/inet/sctp/sctp_impl.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********
______unchanged_portion_omitted_

 402 #define SCTP_SLC_REPORT_INTERVAL        (30 * MINUTES)

 404 #define SCTP_DECR_LISTEN_CNT(sctp)                                      \
 405 {                                                                       \
 406         ASSERT((sctp)->sctp_listen_cnt->slc_cnt > 0);                   \
 407         if (atomic_dec_32_nv(&(sctp)->sctp_listen_cnt->slc_cnt) == 0) \
 407         if (atomic_add_32_nv(&(sctp)->sctp_listen_cnt->slc_cnt, -1) == 0) \
 408                 kmem_free((sctp)->sctp_listen_cnt, sizeof (sctp_listen_cnt_t));\
 409         (sctp)->sctp_listen_cnt = NULL;                                 \
 410 }
______unchanged_portion_omitted_
```

```
**********************************************************
  172567 Mon Jul 28 07:44:30 2014
new/usr/src/uts/common/inet/tcp/tcp_input.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1193 /* BEGIN CSTYLED */
1194 /*
1195  *
1196  * The sockfs ACCEPT path:
1197  * =======================
1198  *
1199  * The eager is now established in its own perimeter as soon as SYN is
1200  * received in tcp_input_listener(). When sockfs receives conn_ind, it
1201  * completes the accept processing on the acceptor STREAM. The sending
1202  * of conn_ind part is common for both sockfs listener and a TLI/XTI
1203  * listener but a TLI/XTI listener completes the accept processing
1204  * on the listener perimeter.
1205  *
1206  * Common control flow for 3 way handshake:
1207  * ----------------------------------------
1208  *
1209  * incoming SYN (listener perimeter)    -> tcp_input_listener()
1210  *
1211  * incoming SYN-ACK-ACK (eager perim)   -> tcp_input_data()
1212  * send T_CONN_IND (listener perim)     -> tcp_send_conn_ind()
1213  *
1214  * Sockfs ACCEPT Path:
1215  * -------------------
1216  *
1217  * open acceptor stream (tcp_open allocates tcp_tli_accept()
1218  * as STREAM entry point)
1219  *
1220  * soaccept() sends T_CONN_RES on the acceptor STREAM to tcp_tli_accept()
1221  *
1222  * tcp_tli_accept() extracts the eager and makes the q->q_ptr <-> eager
1223  * association (we are not behind eager's squeue but sockfs is protecting us
1224  * and no one knows about this stream yet. The STREAMS entry point q->q_info
1225  * is changed to point at tcp_wput().
1226  *
1227  * tcp_accept_common() sends any deferred eagers via tcp_send_pending() to
1228  * listener (done on listener's perimeter).
1229  *
1230  * tcp_tli_accept() calls tcp_accept_finish() on eagers perimeter to finish
1231  * accept.
1232  *
1233  * TLI/XTI client ACCEPT path:
1234  * ---------------------------
1235  *
1236  * soaccept() sends T_CONN_RES on the listener STREAM.
1237  *
1238  * tcp_tli_accept() -> tcp_accept_swap() complete the processing and send
1239  * a M_SETOPS mblk to eager perimeter to finish accept (tcp_accept_finish()).
1240  *
1241  * Locks:
1242  * ======
1243  *
1244  * listener->tcp_eager_lock protects the listeners->tcp_eager_next_q0 and
1245  * and listeners->tcp_eager_next_q.
1246  *
1247  * Referencing:
1248  * ============
1249  *
1250  * 1) We start out in tcp_input_listener by eager placing a ref on
1251  * listener and listener adding eager to listeners->tcp_eager_next_q0.
```



```
1252  *
1253  * 2) When a SYN-ACK-ACK arrives, we send the conn_ind to listener. Before
1254  * doing so we place a ref on the eager. This ref is finally dropped at the
1255  * end of tcp_accept_finish() while unwinding from the squeue, i.e. the
1256  * reference is dropped by the squeue framework.
1257  *
1258  * 3) The ref on listener placed in 1 above is dropped in tcp_accept_finish
1259  *
1260  * The reference must be released by the same entity that added the reference
1261  * In the above scheme, the eager is the entity that adds and releases the
1262  * references. Note that tcp_accept_finish executes in the squeue of the eager
1263  * (albeit after it is attached to the acceptor stream). Though 1. executes
1264  * in the listener's squeue, the eager is nascent at this point and the
1265  * reference can be considered to have been added on behalf of the eager.
1266  *
1267  * Eager getting a Reset or listener closing:
1268  * ==========================================
1269  *
1270  * Once the listener and eager are linked, the listener never does the unlink.
1271  * If the listener needs to close, tcp_eager_cleanup() is called which queues
1272  * a message on all eager perimeter. The eager then does the unlink, clears
1273  * any pointers to the listener's queue and drops the reference to the
1274  * listener. The listener waits in tcp_close outside the squeue until its
1275  * refcount has dropped to 1. This ensures that the listener has waited for
1276  * all eagers to clear their association with the listener.
1277  *
1278  * Similarly, if eager decides to go away, it can unlink itself and close.
1279  * When the T_CONN_RES comes down, we check if eager has closed. Note that
1280  * the reference to eager is still valid because of the extra ref we put
1281  * in tcp_send_conn_ind.
1282  *
1283  * Listener can always locate the eager under the protection
1284  * of the listener->tcp_eager_lock, and then do a refhold
1285  * on the eager during the accept processing.
1286  *
1287  * The acceptor stream accesses the eager in the accept processing
1288  * based on the ref placed on eager before sending T_conn_ind.
1289  * The only entity that can negate this refhold is a listener close
1290  * which is mutually exclusive with an active acceptor stream.
1291  *
1292  * Eager's reference on the listener
1293  * ===================================
1294  *
1295  * If the accept happens (even on a closed eager) the eager drops its
1296  * reference on the listener at the start of tcp_accept_finish. If the
1297  * eager is killed due to an incoming RST before the T_conn_ind is sent up,
1298  * the reference is dropped in tcp_closei_local. If the listener closes,
1299  * the reference is dropped in tcp_eager_kill. In all cases the reference
1300  * is dropped while executing in the eager's context (squeue).
1301  */
1302 /* END CSTYLED */

1304 /* Process the SYN packet, mp, directed at the listener 'tcp' */

1306 /*
1307  * THIS FUNCTION IS DIRECTLY CALLED BY IP VIA SQUEUE FOR SYN.
1308  * tcp_input_data will not see any packets for listeners since the listener
1309  * has conn_recv set to tcp_input_listener.
1310  */
1311 /* ARGSUSED */
1312 static void
1313 tcp_input_listener(void *arg, mblk_t *mp, void *arg2, ip_recv_attr_t *ira)
1314 {
1315         tcpha_t         *tcpha;
1316         uint32_t        seg_seq;
1317         tcp_t           *eager;
```

```
1318        int             err;
1319        conn_t          *econnp = NULL;
1320        squeue_t        *new_sqp;
1321        mblk_t          *mp1;
1322        uint_t          ip_hdr_len;
1323        conn_t          *lconnp = (conn_t *)arg;
1324        tcp_t           *listener = lconnp->conn_tcp;
1325        tcp_stack_t     *tcps = listener->tcp_tcps;
1326        ip_stack_t      *ipst = tcps->tcps_netstack->netstack_ip;
1327        uint_t          flags;
1328        mblk_t          *tpi_mp;
1329        uint_t          ifindex = ira->ira_ruifindex;
1330        boolean_t       tlc_set = B_FALSE;

1332        ip_hdr_len = ira->ira_ip_hdr_length;
1333        tcpha = (tcpha_t *)&mp->b_rptr[ip_hdr_len];
1334        flags = (unsigned int)tcpha->tha_flags & 0xFF;

1336        DTRACE_TCP5(receive, mblk_t *, NULL, ip_xmit_attr_t *, lconnp->conn_ixa,
1337            __dtrace_tcp_void_ip_t *, mp->b_rptr, tcp_t *, listener,
1338            __dtrace_tcp_tcph_t *, tcpha);

1340        if (!(flags & TH_SYN)) {
1341                if ((flags & TH_RST) || (flags & TH_URG)) {
1342                        freemsg(mp);
1343                        return;
1344                }
1345                if (flags & TH_ACK) {
1346                        /* Note this executes in listener's squeue */
1347                        tcp_xmit_listeners_reset(mp, ira, ipst, lconnp);
1348                        return;
1349                }

1351                freemsg(mp);
1352                return;
1353        }

1355        if (listener->tcp_state != TCPS_LISTEN)
1356                goto error2;

1358        ASSERT(IPCL_IS_BOUND(lconnp));

1360        mutex_enter(&listener->tcp_eager_lock);

1362        /*
1363         * The system is under memory pressure, so we need to do our part
1364         * to relieve the pressure.  So we only accept new request if there
1365         * is nothing waiting to be accepted or waiting to complete the 3-way
1366         * handshake.  This means that busy listener will not get too many
1367         * new requests which they cannot handle in time while non-busy
1368         * listener is still functioning properly.
1369         */
1370        if (tcps->tcps_reclaim && (listener->tcp_conn_req_cnt_q > 0 ||
1371            listener->tcp_conn_req_cnt_q0 > 0)) {
1372                mutex_exit(&listener->tcp_eager_lock);
1373                TCP_STAT(tcps, tcp_listen_mem_drop);
1374                goto error2;
1375        }

1377        if (listener->tcp_conn_req_cnt_q >= listener->tcp_conn_req_max) {
1378                mutex_exit(&listener->tcp_eager_lock);
1379                TCP_STAT(tcps, tcp_listendrop);
1380                TCPS_BUMP_MIB(tcps, tcpListenDrop);
1381                if (lconnp->conn_debug) {
1382                        (void) strlog(TCP_MOD_ID, 0, 1, SL_TRACE|SL_ERROR,
1383                            "tcp_input_listener: listen backlog (max=%d) "
```



```
1384                            "overflow (%d pending) on %s",
1385                            listener->tcp_conn_req_max,
1386                            listener->tcp_conn_req_cnt_q,
1387                            tcp_display(listener, NULL, DISP_PORT_ONLY));
1388                }
1389                goto error2;
1390        }

1392        if (listener->tcp_conn_req_cnt_q0 >=
1393            listener->tcp_conn_req_max + tcps->tcps_conn_req_max_q0) {
1394                /*
1395                 * Q0 is full. Drop a pending half-open req from the queue
1396                 * to make room for the new SYN req. Also mark the time we
1397                 * drop a SYN.
1398                 *
1399                 * A more aggressive defense against SYN attack will
1400                 * be to set the "tcp_syn_defense" flag now.
1401                 */
1402                TCP_STAT(tcps, tcp_listendropq0);
1403                listener->tcp_last_rcv_lbolt = ddi_get_lbolt64();
1404                if (!tcp_drop_q0(listener)) {
1405                        mutex_exit(&listener->tcp_eager_lock);
1406                        TCPS_BUMP_MIB(tcps, tcpListenDropQ0);
1407                        if (lconnp->conn_debug) {
1408                                (void) strlog(TCP_MOD_ID, 0, 3, SL_TRACE,
1409                                    "tcp_input_listener: listen half-open "
1410                                    "queue (max=%d) full (%d pending) on %s",
1411                                    tcps->tcps_conn_req_max_q0,
1412                                    listener->tcp_conn_req_cnt_q0,
1413                                    tcp_display(listener, NULL,
1414                                    DISP_PORT_ONLY));
1415                        }
1416                        goto error2;
1417                }
1418        }

1420        /*
1421         * Enforce the limit set on the number of connections per listener.
1422         * Note that tlc_cnt starts with 1.  So need to add 1 to tlc_max
1423         * for comparison.
1424         */
1425        if (listener->tcp_listen_cnt != NULL) {
1426                tcp_listen_cnt_t *tlc = listener->tcp_listen_cnt;
1427                int64_t now;

1429                if (atomic_inc_32_nv(&tlc->tlc_cnt) > tlc->tlc_max + 1) {
1429                *if (atomic_add_32_nv(&tlc->tlc_cnt, 1) > tlc->tlc_max + 1) {*
1430                        mutex_exit(&listener->tcp_eager_lock);
1431                        now = ddi_get_lbolt64();
1432                        atomic_dec_32(&tlc->tlc_cnt);
1432                        *atomic_add_32(&tlc->tlc_cnt, -1);*
1433                        TCP_STAT(tcps, tcp_listen_cnt_drop);
1434                        tlc->tlc_drop++;
1435                        if (now - tlc->tlc_report_time >
1436                            MSEC_TO_TICK(TCP_TLC_REPORT_INTERVAL)) {
1437                                zcmn_err(lconnp->conn_zoneid, CE_WARN,
1438                                    "Listener (port %d) connection max (%u) "
1439                                    "reached: %u attempts dropped total\n",
1440                                    ntohs(listener->tcp_connp->conn_lport),
1441                                    tlc->tlc_max, tlc->tlc_drop);
1442                                tlc->tlc_report_time = now;
1443                        }
1444                        goto error2;
1445                }
1446                tlc_set = B_TRUE;
1447        }
```

```
1449         mutex_exit(&listener->tcp_eager_lock);

1451         /*
1452          * IP sets ira_sqp to either the senders conn_sqp (for loopback)
1453          * or based on the ring (for packets from GLD). Otherwise it is
1454          * set based on lbolt i.e., a somewhat random number.
1455          */
1456         ASSERT(ira->ira_sqp != NULL);
1457         new_sqp = ira->ira_sqp;

1459         econnp = (conn_t *)tcp_get_conn(arg2, tcps);
1460         if (econnp == NULL)
1461                 goto error2;

1463         ASSERT(econnp->conn_netstack == lconnp->conn_netstack);
1464         econnp->conn_sqp = new_sqp;
1465         econnp->conn_initial_sqp = new_sqp;
1466         econnp->conn_ixa->ixa_sqp = new_sqp;

1468         econnp->conn_fport = tcpha->tha_lport;
1469         econnp->conn_lport = tcpha->tha_fport;

1471         err = conn_inherit_parent(lconnp, econnp);
1472         if (err != 0)
1473                 goto error3;

1475         /* We already know the laddr of the new connection is ours */
1476         econnp->conn_ixa->ixa_src_generation = ipst->ips_src_generation;

1478         ASSERT(OK_32PTR(mp->b_rptr));
1479         ASSERT(IPH_HDR_VERSION(mp->b_rptr) == IPV4_VERSION ||
1480             IPH_HDR_VERSION(mp->b_rptr) == IPV6_VERSION);

1482         if (lconnp->conn_family == AF_INET) {
1483                 ASSERT(IPH_HDR_VERSION(mp->b_rptr) == IPV4_VERSION);
1484                 tpi_mp = tcp_conn_create_v4(lconnp, econnp, mp, ira);
1485         } else {
1486                 tpi_mp = tcp_conn_create_v6(lconnp, econnp, mp, ira);
1487         }

1489         if (tpi_mp == NULL)
1490                 goto error3;

1492         eager = econnp->conn_tcp;
1493         eager->tcp_detached = B_TRUE;
1494         SOCK_CONNID_INIT(eager->tcp_connid);

1496         /*
1497          * Initialize the eager's tcp_t and inherit some parameters from
1498          * the listener.
1499          */
1500         tcp_init_values(eager, listener);

1502         ASSERT((econnp->conn_ixa->ixa_flags &
1503             (IXAF_SET_ULP_CKSUM | IXAF_VERIFY_SOURCE |
1504             IXAF_VERIFY_PMTU | IXAF_VERIFY_LSO)) ==
1505             (IXAF_SET_ULP_CKSUM | IXAF_VERIFY_SOURCE |
1506             IXAF_VERIFY_PMTU | IXAF_VERIFY_LSO));

1508         if (!tcps->tcps_dev_flow_ctl)
1509                 econnp->conn_ixa->ixa_flags |= IXAF_NO_DEV_FLOW_CTL;

1511         /* Prepare for diffing against previous packets */
1512         eager->tcp_recvifindex = 0;
1513         eager->tcp_recvhops = 0xffffffffU;
```



```
1515         if (!(ira->ira_flags & IRAF_IS_IPV4) && econnp->conn_bound_if == 0) {
1516                 if (IN6_IS_ADDR_LINKSCOPE(&econnp->conn_faddr_v6) ||
1517                     IN6_IS_ADDR_LINKSCOPE(&econnp->conn_laddr_v6)) {
1518                         econnp->conn_incoming_ifindex = ifindex;
1519                         econnp->conn_ixa->ixa_flags |= IXAF_SCOPEID_SET;
1520                         econnp->conn_ixa->ixa_scopeid = ifindex;
1521                 }
1522         }

1524         if ((ira->ira_flags & (IRAF_IS_IPV4|IRAF_IPV4_OPTIONS)) ==
1525             (IRAF_IS_IPV4|IRAF_IPV4_OPTIONS) &&
1526             tcps->tcps_rev_src_routes) {
1527                 ipha_t *ipha = (ipha_t *)mp->b_rptr;
1528                 ip_pkt_t *ipp = &econnp->conn_xmit_ipp;

1530                 /* Source routing option copyover (reverse it) */
1531                 err = ip_find_hdr_v4(ipha, ipp, B_TRUE);
1532                 if (err != 0) {
1533                         freemsg(tpi_mp);
1534                         goto error3;
1535                 }
1536                 ip_pkt_source_route_reverse_v4(ipp);
1537         }

1539         ASSERT(eager->tcp_conn.tcp_eager_conn_ind == NULL);
1540         ASSERT(!eager->tcp_tconnind_started);
1541         /*
1542          * If the SYN came with a credential, it's a loopback packet or a
1543          * labeled packet; attach the credential to the TPI message.
1544          */
1545         if (ira->ira_cred != NULL)
1546                 mblk_setcred(tpi_mp, ira->ira_cred, ira->ira_cpid);

1548         eager->tcp_conn.tcp_eager_conn_ind = tpi_mp;
1549         ASSERT(eager->tcp_ordrel_mp == NULL);

1551         /* Inherit the listener's non-STREAMS flag */
1552         if (IPCL_IS_NONSTR(lconnp)) {
1553                 econnp->conn_flags |= IPCL_NONSTR;
1554                 /* All non-STREAMS tcp_ts are sockets */
1555                 eager->tcp_issocket = B_TRUE;
1556         } else {
1557                 /*
1558                  * Pre-allocate the T_ordrel_ind mblk for TPI socket so that
1559                  * at close time, we will always have that to send up.
1560                  * Otherwise, we need to do special handling in case the
1561                  * allocation fails at that time.
1562                  */
1563                 if ((eager->tcp_ordrel_mp = mi_tpi_ordrel_ind()) == NULL)
1564                         goto error3;
1565         }
1566         /*
1567          * Now that the IP addresses and ports are setup in econnp we
1568          * can do the IPsec policy work.
1569          */
1570         if (ira->ira_flags & IRAF_IPSEC_SECURE) {
1571                 if (lconnp->conn_policy != NULL) {
1572                         /*
1573                          * Inherit the policy from the listener; use
1574                          * actions from ira
1575                          */
1576                         if (!ip_ipsec_policy_inherit(econnp, lconnp, ira)) {
1577                                 CONN_DEC_REF(econnp);
1578                                 freemsg(mp);
1579                                 goto error3;
```

```
1580                        }
1581                }
1582        }

1584        /*
1585         * tcp_set_destination() may set tcp_rwnd according to the route
1586         * metrics. If it does not, the eager's receive window will be set
1587         * to the listener's receive window later in this function.
1588         */
1589        eager->tcp_rwnd = 0;

1591        if (is_system_labeled()) {
1592                ip_xmit_attr_t *ixa = econnp->conn_ixa;

1594                ASSERT(ira->ira_tsl != NULL);
1595                /* Discard any old label */
1596                if (ixa->ixa_free_flags & IXA_FREE_TSL) {
1597                        ASSERT(ixa->ixa_tsl != NULL);
1598                        label_rele(ixa->ixa_tsl);
1599                        ixa->ixa_free_flags &= ~IXA_FREE_TSL;
1600                        ixa->ixa_tsl = NULL;
1601                }
1602                if ((lconnp->conn_mlp_type != mlptSingle ||
1603                    lconnp->conn_mac_mode != CONN_MAC_DEFAULT) &&
1604                    ira->ira_tsl != NULL) {
1605                        /*
1606                         * If this is an MLP connection or a MAC-Exempt
1607                         * connection with an unlabeled node, packets are to be
1608                         * exchanged using the security label of the received
1609                         * SYN packet instead of the server application's label.
1610                         * tsol_check_dest called from ip_set_destination
1611                         * might later update TSF_UNLABELED by replacing
1612                         * ixa_tsl with a new label.
1613                         */
1614                        label_hold(ira->ira_tsl);
1615                        ip_xmit_attr_replace_tsl(ixa, ira->ira_tsl);
1616                        DTRACE_PROBE2(mlp_syn_accept, conn_t *,
1617                            econnp, ts_label_t *, ixa->ixa_tsl)
1618                } else {
1619                        ixa->ixa_tsl = crgetlabel(econnp->conn_cred);
1620                        DTRACE_PROBE2(syn_accept, conn_t *,
1621                            econnp, ts_label_t *, ixa->ixa_tsl)
1622                }
1623                /*
1624                 * conn_connect() called from tcp_set_destination will verify
1625                 * the destination is allowed to receive packets at the
1626                 * security label of the SYN-ACK we are generating. As part of
1627                 * that, tsol_check_dest() may create a new effective label for
1628                 * this connection.
1629                 * Finally conn_connect() will call conn_update_label.
1630                 * All that remains for TCP to do is to call
1631                 * conn_build_hdr_template which is done as part of
1632                 * tcp_set_destination.
1633                 */
1634        }

1636        /*
1637         * Since we will clear tcp_listener before we clear tcp_detached
1638         * in the accept code we need tcp_hard_binding aka tcp_accept_inprogress
1639         * so we can tell a TCP_IS_DETACHED_NONEAGER apart.
1640         */
1641        eager->tcp_hard_binding = B_TRUE;

1643        tcp_bind_hash_insert(&tcps->tcps_bind_fanout[
1644            TCP_BIND_HASH(econnp->conn_lport)], eager, 0);
```



```
1646        CL_INET_CONNECT(econnp, B_FALSE, err);
1647        if (err != 0) {
1648                tcp_bind_hash_remove(eager);
1649                goto error3;
1650        }

1652        SOCK_CONNID_BUMP(eager->tcp_connid);

1654        /*
1655         * Adapt our mss, ttl, ... based on the remote address.
1656         */

1658        if (tcp_set_destination(eager) != 0) {
1659                TCPS_BUMP_MIB(tcps, tcpAttemptFails);
1660                /* Undo the bind_hash_insert */
1661                tcp_bind_hash_remove(eager);
1662                goto error3;
1663        }

1665        /* Process all TCP options. */
1666        tcp_process_options(eager, tcpha);

1668        /* Is the other end ECN capable? */
1669        if (tcps->tcps_ecn_permitted >= 1 &&
1670            (tcpha->tha_flags & (TH_ECE|TH_CWR)) == (TH_ECE|TH_CWR)) {
1671                eager->tcp_ecn_ok = B_TRUE;
1672        }

1674        /*
1675         * The listener's conn_rcvbuf should be the default window size or a
1676         * window size changed via SO_RCVBUF option. First round up the
1677         * eager's tcp_rwnd to the nearest MSS. Then find out the window
1678         * scale option value if needed. Call tcp_rwnd_set() to finish the
1679         * setting.
1680         *
1681         * Note if there is a rpipe metric associated with the remote host,
1682         * we should not inherit receive window size from listener.
1683         */
1684        eager->tcp_rwnd = MSS_ROUNDUP(
1685            (eager->tcp_rwnd == 0 ? econnp->conn_rcvbuf :
1686            eager->tcp_rwnd), eager->tcp_mss);
1687        if (eager->tcp_snd_ws_ok)
1688                tcp_set_ws_value(eager);
1689        /*
1690         * Note that this is the only place tcp_rwnd_set() is called for
1691         * accepting a connection.  We need to call it here instead of
1692         * after the 3-way handshake because we need to tell the other
1693         * side our rwnd in the SYN-ACK segment.
1694         */
1695        (void) tcp_rwnd_set(eager, eager->tcp_rwnd);

1697        ASSERT(eager->tcp_connp->conn_rcvbuf != 0 &&
1698            eager->tcp_connp->conn_rcvbuf == eager->tcp_rwnd);

1700        ASSERT(econnp->conn_rcvbuf != 0 &&
1701            econnp->conn_rcvbuf == eager->tcp_rwnd);

1703        /* Put a ref on the listener for the eager. */
1704        CONN_INC_REF(lconnp);
1705        mutex_enter(&listener->tcp_eager_lock);
1706        listener->tcp_eager_next_q0->tcp_eager_prev_q0 = eager;
1707        eager->tcp_eager_next_q0 = listener->tcp_eager_next_q0;
1708        listener->tcp_eager_next_q0 = eager;
1709        eager->tcp_eager_prev_q0 = listener;

1711        /* Set tcp_listener before adding it to tcp_conn_fanout */
```

```
1712         eager->tcp_listener = listener;
1713         eager->tcp_saved_listener = listener;

1715         /*
1716          * Set tcp_listen_cnt so that when the connection is done, the counter
1717          * is decremented.
1718          */
1719         eager->tcp_listen_cnt = listener->tcp_listen_cnt;

1721         /*
1722          * Tag this detached tcp vector for later retrieval
1723          * by our listener client in tcp_accept().
1724          */
1725         eager->tcp_conn_req_seqnum = listener->tcp_conn_req_seqnum;
1726         listener->tcp_conn_req_cnt_q0++;
1727         if (++listener->tcp_conn_req_seqnum == -1) {
1728                 /*
1729                  * -1 is "special" and defined in TPI as something
1730                  * that should never be used in T_CONN_IND
1731                  */
1732                 ++listener->tcp_conn_req_seqnum;
1733         }
1734         mutex_exit(&listener->tcp_eager_lock);

1736         if (listener->tcp_syn_defense) {
1737                 /* Don't drop the SYN that comes from a good IP source */
1738                 ipaddr_t *addr_cache;

1740                 addr_cache = (ipaddr_t *)(listener->tcp_ip_addr_cache);
1741                 if (addr_cache != NULL && econnp->conn_faddr_v4 ==
1742                     addr_cache[IP_ADDR_CACHE_HASH(econnp->conn_faddr_v4)]) {
1743                         eager->tcp_dontdrop = B_TRUE;
1744                 }
1745         }

1747         /*
1748          * We need to insert the eager in its own perimeter but as soon
1749          * as we do that, we expose the eager to the classifier and
1750          * should not touch any field outside the eager's perimeter.
1751          * So do all the work necessary before inserting the eager
1752          * in its own perimeter. Be optimistic that conn_connect()
1753          * will succeed but undo everything if it fails.
1754          */
1755         seg_seq = ntohl(tcpha->tha_seq);
1756         eager->tcp_irs = seg_seq;
1757         eager->tcp_rack = seg_seq;
1758         eager->tcp_rnxt = seg_seq + 1;
1759         eager->tcp_tcpha->tha_ack = htonl(eager->tcp_rnxt);
1760         TCPS_BUMP_MIB(tcps, tcpPassiveOpens);
1761         eager->tcp_state = TCPS_SYN_RCVD;
1762         DTRACE_TCP6(state__change, void, NULL, ip_xmit_attr_t *,
1763             econnp->conn_ixa, void, NULL, tcp_t *, eager, void, NULL,
1764             int32_t, TCPS_LISTEN);

1766         mp1 = tcp_xmit_mp(eager, eager->tcp_xmit_head, eager->tcp_mss,
1767             NULL, NULL, eager->tcp_iss, B_FALSE, NULL, B_FALSE);
1768         if (mp1 == NULL) {
1769                 /*
1770                  * Increment the ref count as we are going to
1771                  * enqueueing an mp in squeue
1772                  */
1773                 CONN_INC_REF(econnp);
1774                 goto error;
1775         }

1777         /*
```



```
1778          * We need to start the rto timer. In normal case, we start
1779          * the timer after sending the packet on the wire (or at
1780          * least believing that packet was sent by waiting for
1781          * conn_ip_output() to return). Since this is the first packet
1782          * being sent on the wire for the eager, our initial tcp_rto
1783          * is at least tcp_rexmit_interval_min which is a fairly
1784          * large value to allow the algorithm to adjust slowly to large
1785          * fluctuations of RTT during first few transmissions.
1786          *
1787          * Starting the timer first and then sending the packet in this
1788          * case shouldn't make much difference since tcp_rexmit_interval_min
1789          * is of the order of several 100ms and starting the timer
1790          * first and then sending the packet will result in difference
1791          * of few micro seconds.
1792          *
1793          * Without this optimization, we are forced to hold the fanout
1794          * lock across the ipcl_bind_insert() and sending the packet
1795          * so that we don't race against an incoming packet (maybe RST)
1796          * for this eager.
1797          *
1798          * It is necessary to acquire an extra reference on the eager
1799          * at this point and hold it until after tcp_send_data() to
1800          * ensure against an eager close race.
1801          */

1803         CONN_INC_REF(econnp);

1805         TCP_TIMER_RESTART(eager, eager->tcp_rto);

1807         /*
1808          * Insert the eager in its own perimeter now. We are ready to deal
1809          * with any packets on eager.
1810          */
1811         if (ipcl_conn_insert(econnp) != 0)
1812                 goto error;

1814         ASSERT(econnp->conn_ixa->ixa_notify_cookie == econnp->conn_tcp);
1815         freemsg(mp);
1816         /*
1817          * Send the SYN-ACK. Use the right squeue so that conn_ixa is
1818          * only used by one thread at a time.
1819          */
1820         if (econnp->conn_sqp == lconnp->conn_sqp) {
1821                 DTRACE_TCP5(send, mblk_t *, NULL, ip_xmit_attr_t *,
1822                     econnp->conn_ixa, __dtrace_tcp_void_ip_t *, mp1->b_rptr,
1823                     tcp_t *, eager, __dtrace_tcp_tcph_t *,
1824                     &mp1->b_rptr[econnp->conn_ixa->ixa_ip_hdr_length]);
1825                 (void) conn_ip_output(mp1, econnp->conn_ixa);
1826                 CONN_DEC_REF(econnp);
1827         } else {
1828                 SQUEUE_ENTER_ONE(econnp->conn_sqp, mp1, tcp_send_synack,
1829                     econnp, NULL, SQ_PROCESS, SQTAG_TCP_SEND_SYNACK);
1830         }
1831         return;
1832 error:
1833         freemsg(mp1);
1834         eager->tcp_closemp_used = B_TRUE;
1835         TCP_DEBUG_GETPCSTACK(eager->tcmp_stk, 15);
1836         mp1 = &eager->tcp_closemp;
1837         SQUEUE_ENTER_ONE(econnp->conn_sqp, mp1, tcp_eager_kill,
1838             econnp, NULL, SQ_FILL, SQTAG_TCP_CONN_REQ_2);

1840         /*
1841          * If a connection already exists, send the mp to that connections so
1842          * that it can be appropriately dealt with.
1843          */
```

```
1844         ipst = tcps->tcps_netstack->netstack_ip;

1846         if ((connp = ipcl_classify(mp, ira, ipst)) != NULL) {
1847                 if (!IPCL_IS_CONNECTED(connp)) {
1848                         /*
1849                          * Something bad happened. ipcl_conn_insert()
1850                          * failed because a connection already existed
1851                          * in connected hash but we can't find it
1852                          * anymore (someone blew it away). Just
1853                          * free this message and hopefully remote
1854                          * will retransmit at which time the SYN can be
1855                          * treated as a new connection or dealth with
1856                          * a TH_RST if a connection already exists.
1857                          */
1858                         CONN_DEC_REF(connp);
1859                         freemsg(mp);
1860                 } else {
1861                         SQUEUE_ENTER_ONE(connp->conn_sqp, mp, tcp_input_data,
1862                             connp, ira, SQ_FILL, SQTAG_TCP_CONN_REQ_1);
1863                 }
1864         } else {
1865                 /* Nobody wants this packet */
1866                 freemsg(mp);
1867         }
1868         return;
1869 error3:
1870         CONN_DEC_REF(connp);
1871 error2:
1872         freemsg(mp);
1873         if (tlc_set)
1874                 atomic_dec_32(&listener->tcp_listen_cnt->tlc_cnt);
1874                 atomic_add_32(&listener->tcp_listen_cnt->tlc_cnt, -1);
1875 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   30068 Mon Jul 28 07:44:31 2014
new/usr/src/uts/common/inet/tcp_impl.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 369 #define TCP_TLC_REPORT_INTERVAL (30 * MINUTES)

 371 #define TCP_DECR_LISTEN_CNT(tcp)                                        \
 372 {                                                                       \
 373         ASSERT((tcp)->tcp_listen_cnt->tlc_cnt > 0);                     \
 374         if (atomic_dec_32_nv(&(tcp)->tcp_listen_cnt->tlc_cnt) == 0) \
 374         if (atomic_add_32_nv(&(tcp)->tcp_listen_cnt->tlc_cnt, -1) == 0) \
 375                 kmem_free((tcp)->tcp_listen_cnt, sizeof (tcp_listen_cnt_t)); \
 376         (tcp)->tcp_listen_cnt = NULL;                                   \
 377 }
______unchanged_portion_omitted_
```

****
67221 Mon Jul 28 07:44:31 2014
new/usr/src/uts/common/io/bscbus.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
2609 #ifdef BSCBUS_LOGSTATUS
2610 void bscbus_cmd_log(struct bscbus_channel_state *csp, bsc_cmd_stamp_t cat,
2611     uint8_t status, uint8_t data)
2612 {
2613         int idx;
2614         bsc_cmd_log_t *logp;
2615         struct bscbus_state *ssp;

2617         if ((csp) == NULL)
2618                 return;
2619         if ((ssp = (csp)->ssp) == NULL)
2620                 return;
2621         if (ssp->cmd_log_size == 0)
2622                 return;
2623         if ((bscbus_cmd_log_flags & (1 << cat)) == 0)
2624                 return;
2625         idx = atomic_inc_32_nv(&ssp->cmd_log_idx);
2625         idx = atomic_add_32_nv(&ssp->cmd_log_idx, 1);
2626         logp = &ssp->cmd_log[idx % ssp->cmd_log_size];
2627         logp->bcl_seq = idx;
2628         logp->bcl_cat = cat;
2629         logp->bcl_now = gethrtime();
2630         logp->bcl_chno = csp->chno;
2631         logp->bcl_cmdstate = csp->cmdstate;
2632         logp->bcl_status = status;
2633         logp->bcl_data = data;
2634 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   39720 Mon Jul 28 07:44:31 2014
new/usr/src/uts/common/io/chxge/pe.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2005 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * This file is part of the Chelsio T1 Ethernet driver.
 *
 * Copyright (C) 2003-2005 Chelsio Communications.  All rights reserved.
 */

/*
 * Solaris Multithreaded STREAMS Chelsio PCI Ethernet Driver.
 * Interface code
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/types.h>
#include <sys/systm.h>
#include <sys/cmn_err.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/byteorder.h>
#include <sys/atomic.h>
#include <sys/ethernet.h>
#if PE_PROFILING_ENABLED
#include <sys/time.h>
#endif
#include <sys/gld.h>
#include "ostypes.h"
#include "common.h"
#include "oschtoe.h"
#ifdef CONFIG_CHELSIO_T1_1G
#include "fpga_defs.h"
#endif
#include "regs.h"
#ifdef CONFIG_CHELSIO_T1_OFFLOAD
#include "mc3.h"
#include "mc4.h"
```



```
#endif
#include "sge.h"
#include "tp.h"
#ifdef CONFIG_CHELSIO_T1_OFFLOAD
#include "ulp.h"
#endif
#include "espi.h"
#include "elmer0.h"
#include "gmac.h"
#include "cphy.h"
#include "suni1x10gexp_regs.h"
#include "ch.h"

#define MLEN(mp) ((mp)->b_wptr - (mp)->b_rptr)

extern uint32_t buffers_in_use[];
extern kmutex_t in_use_l;
extern uint32_t in_use_index;

static void link_start(ch_t *sa, struct pe_port_t *pp);
static ch_esb_t *ch_alloc_small_esbbuf(ch_t *sa, uint32_t i);
static ch_esb_t *ch_alloc_big_esbbuf(ch_t *sa, uint32_t i);
void ch_big_rbuf_recycle(ch_esb_t *rbp);
void ch_small_rbuf_recycle(ch_esb_t *rbp);
static const struct board_info *pe_sa_init(ch_t *sa);
static int ch_set_config_data(ch_t *chp);
void pe_rbuf_pool_free(ch_t *chp);
static void pe_free_driver_resources(ch_t *sa);
static void update_mtu_tab(ch_t *adapter);
static int pe_change_mtu(ch_t *chp);

/*
 * CPL5 Defines (from netinet/cpl5_commands.h)
 */
#define FLITSTOBYTES    8

#define CPL_FORMAT_0_SIZE 8
#define CPL_FORMAT_1_SIZE 16
#define CPL_FORMAT_2_SIZE 24
#define CPL_FORMAT_3_SIZE 32
#define CPL_FORMAT_4_SIZE 40
#define CPL_FORMAT_5_SIZE 48

#define TID_MASK 0xffffff

#define PE_LINK_SPEED_AUTONEG   5

static int pe_small_rbuf_pool_init(ch_t *sa);
static int pe_big_rbuf_pool_init(ch_t *sa);
static int pe_make_fake_arp(ch_t *chp, unsigned char *arpp);
static uint32_t pe_get_ip(unsigned char *arpp);

/*
 * May be set in /etc/system to 0 to use default latency timer for 10G.
 * See PCI register 0xc definition.
 */
int enable_latency_timer = 1;

/*
 * May be set in /etc/system to 0 to disable hardware checksum for
 * TCP and UDP.
 */
int enable_checksum_offload = 1;

/*
 * Multiplier for freelist pool.
```

```
 126  */
 127 int fl_sz_multiplier = 6;

 129 uint_t
 130 pe_intr(ch_t *sa)
 131 {
 132         mutex_enter(&sa->ch_intr);

 134         if (sge_data_in(sa->sge)) {
 135                 sa->isr_intr++;
 136                 mutex_exit(&sa->ch_intr);
 137                 return (DDI_INTR_CLAIMED);
 138         }

 140         mutex_exit(&sa->ch_intr);

 142         return (DDI_INTR_UNCLAIMED);
 143 }
```

*unchanged_portion_omitted_*

```
1367 void
1368 ch_small_rbuf_recycle(ch_esb_t *rbp)
1369 {
1370         ch_t *sa = rbp->cs_sa;

1372         if (rbp->cs_flag) {
1373                 uint32_t i;
1374                 /*
1375                  * free private buffer allocated in ch_alloc_esbbuf()
1376                  */
1377                 ch_free_dma_mem(rbp->cs_dh, rbp->cs_ah);

1379                 i = rbp->cs_index;

1381                 /*
1382                  * free descripter buffer
1383                  */
1384                 kmem_free(rbp, sizeof (ch_esb_t));

1386                 /*
1387                  * decrement count of receive buffers freed by callback
1388                  * We decrement here so anyone trying to do fini will
1389                  * only remove the driver once the counts go to 0.
1390                  */
1391                 atomic_dec_32(&buffers_in_use[i]);
1393                 atomic_add_32(&buffers_in_use[i], -1);

1393                 return;
1394         }

1396         mutex_enter(&sa->ch_small_esbl);
1397         rbp->cs_next = sa->ch_small_esb_free;
1398         sa->ch_small_esb_free = rbp;
1399         mutex_exit(&sa->ch_small_esbl);

1401         /*
1402          * decrement count of receive buffers freed by callback
1403          */
1404         atomic_dec_32(&buffers_in_use[rbp->cs_index]);
1406         atomic_add_32(&buffers_in_use[rbp->cs_index], -1);
1405 }

1407 /*
1408  * callback function from freeb() when esballoced mblk freed.
1409  */
1410 void
```



```
1411 ch_big_rbuf_recycle(ch_esb_t *rbp)
1412 {
1413         ch_t *sa = rbp->cs_sa;

1415         if (rbp->cs_flag) {
1416                 uint32_t i;
1417                 /*
1418                  * free private buffer allocated in ch_alloc_esbbuf()
1419                  */
1420                 ch_free_dma_mem(rbp->cs_dh, rbp->cs_ah);

1422                 i = rbp->cs_index;

1424                 /*
1425                  * free descripter buffer
1426                  */
1427                 kmem_free(rbp, sizeof (ch_esb_t));

1429                 /*
1430                  * decrement count of receive buffers freed by callback
1431                  * We decrement here so anyone trying to do fini will
1432                  * only remove the driver once the counts go to 0.
1433                  */
1434                 atomic_dec_32(&buffers_in_use[i]);
1436                 atomic_add_32(&buffers_in_use[i], -1);

1436                 return;
1437         }

1439         mutex_enter(&sa->ch_big_esbl);
1440         rbp->cs_next = sa->ch_big_esb_free;
1441         sa->ch_big_esb_free = rbp;
1442         mutex_exit(&sa->ch_big_esbl);

1444         /*
1445          * decrement count of receive buffers freed by callback
1446          */
1447         atomic_dec_32(&buffers_in_use[rbp->cs_index]);
1449         atomic_add_32(&buffers_in_use[rbp->cs_index], -1);
1448 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   97066 Mon Jul 28 07:44:31 2014
new/usr/src/uts/common/io/comstar/lu/stmf_sbd/sbd.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1716 int
1717 sbd_create_register_lu(sbd_create_and_reg_lu_t *slu, int struct_sz,
1718     uint32_t *err_ret)
1719 {
1720         char *namebuf;
1721         sbd_lu_t *sl;
1722         stmf_lu_t *lu;
1723         char *p;
1724         int sz;
1725         int alloc_sz;
1726         int ret = EIO;
1727         int flag;
1728         int wcd = 0;
1729         uint32_t hid = 0;
1730         enum vtype vt;

1732         sz = struct_sz - sizeof (sbd_create_and_reg_lu_t) + 8 + 1;

1734         *err_ret = 0;

1736         /* Lets validate various offsets */
1737         if (((slu->slu_meta_fname_valid) &&
1738             (slu->slu_meta_fname_off >= sz)) ||
1739             (slu->slu_data_fname_off >= sz) ||
1740             ((slu->slu_alias_valid) &&
1741             (slu->slu_alias_off >= sz)) ||
1742             ((slu->slu_mgmt_url_valid) &&
1743             (slu->slu_mgmt_url_off >= sz)) ||
1744             ((slu->slu_serial_valid) &&
1745             ((slu->slu_serial_off + slu->slu_serial_size) >= sz))) {
1746                 return (EINVAL);
1747         }

1749         namebuf = kmem_zalloc(sz, KM_SLEEP);
1750         bcopy(slu->slu_buf, namebuf, sz - 1);
1751         namebuf[sz - 1] = 0;

1753         alloc_sz = sizeof (sbd_lu_t) + sizeof (sbd_pgr_t);
1754         if (slu->slu_meta_fname_valid) {
1755                 alloc_sz += strlen(namebuf + slu->slu_meta_fname_off) + 1;
1756         }
1757         alloc_sz += strlen(namebuf + slu->slu_data_fname_off) + 1;
1758         if (slu->slu_alias_valid) {
1759                 alloc_sz += strlen(namebuf + slu->slu_alias_off) + 1;
1760         }
1761         if (slu->slu_mgmt_url_valid) {
1762                 alloc_sz += strlen(namebuf + slu->slu_mgmt_url_off) + 1;
1763         }
1764         if (slu->slu_serial_valid) {
1765                 alloc_sz += slu->slu_serial_size;
1766         }

1768         lu = (stmf_lu_t *)stmf_alloc(STMF_STRUCT_STMF_LU, alloc_sz, 0);
1769         if (lu == NULL) {
1770                 kmem_free(namebuf, sz);
1771                 return (ENOMEM);
1772         }
1773         sl = (sbd_lu_t *)lu->lu_provider_private;
1774         bzero(sl, alloc_sz);
```



```
1775         sl->sl_lu = lu;
1776         sl->sl_alloc_size = alloc_sz;
1777         sl->sl_pgr = (sbd_pgr_t *)(sl + 1);
1778         rw_init(&sl->sl_pgr->pgr_lock, NULL, RW_DRIVER, NULL);
1779         mutex_init(&sl->sl_lock, NULL, MUTEX_DRIVER, NULL);
1780         mutex_init(&sl->sl_metadata_lock, NULL, MUTEX_DRIVER, NULL);
1781         rw_init(&sl->sl_access_state_lock, NULL, RW_DRIVER, NULL);
1782         p = ((char *)sl) + sizeof (sbd_lu_t) + sizeof (sbd_pgr_t);
1783         sl->sl_data_filename = p;
1784         (void) strcpy(sl->sl_data_filename, namebuf + slu->slu_data_fname_off);
1785         p += strlen(sl->sl_data_filename) + 1;
1786         sl->sl_meta_offset = SBD_META_OFFSET;
1787         sl->sl_access_state = SBD_LU_ACTIVE;
1788         if (slu->slu_meta_fname_valid) {
1789                 sl->sl_alias = sl->sl_name = sl->sl_meta_filename = p;
1790                 (void) strcpy(sl->sl_meta_filename, namebuf +
1791                     slu->slu_meta_fname_off);
1792                 p += strlen(sl->sl_meta_filename) + 1;
1793         } else {
1794                 sl->sl_alias = sl->sl_name = sl->sl_data_filename;
1795                 if (sbd_is_zvol(sl->sl_data_filename)) {
1796                         sl->sl_flags |= SL_ZFS_META;
1797                         sl->sl_meta_offset = 0;
1798                 } else {
1799                         sl->sl_flags |= SL_SHARED_META;
1800                         sl->sl_data_offset = SHARED_META_DATA_SIZE;
1801                         sl->sl_total_meta_size = SHARED_META_DATA_SIZE;
1802                         sl->sl_meta_size_used = 0;
1803                 }
1804         }
1805         if (slu->slu_alias_valid) {
1806                 sl->sl_alias = p;
1807                 (void) strcpy(p, namebuf + slu->slu_alias_off);
1808                 p += strlen(sl->sl_alias) + 1;
1809         }
1810         if (slu->slu_mgmt_url_valid) {
1811                 sl->sl_mgmt_url = p;
1812                 (void) strcpy(p, namebuf + slu->slu_mgmt_url_off);
1813                 p += strlen(sl->sl_mgmt_url) + 1;
1814         }
1815         if (slu->slu_serial_valid) {
1816                 sl->sl_serial_no = (uint8_t *)p;
1817                 bcopy(namebuf + slu->slu_serial_off, sl->sl_serial_no,
1818                     slu->slu_serial_size);
1819                 sl->sl_serial_no_size = slu->slu_serial_size;
1820                 p += slu->slu_serial_size;
1821         }
1822         kmem_free(namebuf, sz);
1823         if (slu->slu_vid_valid) {
1824                 bcopy(slu->slu_vid, sl->sl_vendor_id, 8);
1825                 sl->sl_flags |= SL_VID_VALID;
1826         }
1827         if (slu->slu_pid_valid) {
1828                 bcopy(slu->slu_pid, sl->sl_product_id, 16);
1829                 sl->sl_flags |= SL_PID_VALID;
1830         }
1831         if (slu->slu_rev_valid) {
1832                 bcopy(slu->slu_rev, sl->sl_revision, 4);
1833                 sl->sl_flags |= SL_REV_VALID;
1834         }
1835         if (slu->slu_write_protected) {
1836                 sl->sl_flags |= SL_WRITE_PROTECTED;
1837         }
1838         if (slu->slu_blksize_valid) {
1839                 if ((slu->slu_blksize & (slu->slu_blksize - 1)) ||
1840                     (slu->slu_blksize > (32 * 1024)) ||
```

```
1841                    (slu->slu_blksize == 0)) {
1842                        *err_ret = SBD_RET_INVALID_BLKSIZE;
1843                        ret = EINVAL;
1844                        goto scm_err_out;
1845                }
1846                while ((1 << sl->sl_data_blocksize_shift) != slu->slu_blksize) {
1847                        sl->sl_data_blocksize_shift++;
1848                }
1849        } else {
1850                sl->sl_data_blocksize_shift = 9;        /* 512 by default */
1851                slu->slu_blksize = 512;
1852        }

1854        /* Now lets start creating meta */
1855        sl->sl_trans_op = SL_OP_CREATE_REGISTER_LU;
1856        if (sbd_link_lu(sl) != SBD_SUCCESS) {
1857                *err_ret = SBD_RET_FILE_ALREADY_REGISTERED;
1858                ret = EALREADY;
1859                goto scm_err_out;
1860        }

1862        /* 1st focus on the data store */
1863        if (slu->slu_lu_size_valid) {
1864                sl->sl_lu_size = slu->slu_lu_size;
1865        }
1866        ret = sbd_open_data_file(sl, err_ret, slu->slu_lu_size_valid, 0, 0);
1867        slu->slu_ret_filesize_nbits = sl->sl_data_fs_nbits;
1868        slu->slu_lu_size = sl->sl_lu_size;
1869        if (ret) {
1870                goto scm_err_out;
1871        }

1873        /*
1874         * Check if we were explicitly asked to disable/enable write
1875         * cache on the device, otherwise get current device setting.
1876         */
1877        if (slu->slu_writeback_cache_disable_valid) {
1878                if (slu->slu_writeback_cache_disable) {
1879                        /*
1880                         * Set write cache disable on the device. If it fails,
1881                         * we'll support it using sync/flush.
1882                         */
1883                        (void) sbd_wcd_set(1, sl);
1884                        wcd = 1;
1885                } else {
1886                        /*
1887                         * Set write cache enable on the device. If it fails,
1888                         * return an error.
1889                         */
1890                        if (sbd_wcd_set(0, sl) != SBD_SUCCESS) {
1891                                *err_ret = SBD_RET_WRITE_CACHE_SET_FAILED;
1892                                ret = EFAULT;
1893                                goto scm_err_out;
1894                        }
1895                }
1896        } else {
1897                sbd_wcd_get(&wcd, sl);
1898        }

1900        if (wcd) {
1901                sl->sl_flags |= SL_WRITEBACK_CACHE_DISABLE |
1902                    SL_SAVED_WRITE_CACHE_DISABLE;
1903        }

1905        if (sl->sl_flags & SL_SHARED_META) {
1906                goto over_meta_open;
```



```
1907        }
1908        if (sl->sl_flags & SL_ZFS_META) {
1909                if (sbd_create_zfs_meta_object(sl) != SBD_SUCCESS) {
1910                        *err_ret = SBD_RET_ZFS_META_CREATE_FAILED;
1911                        ret = ENOMEM;
1912                        goto scm_err_out;
1913                }
1914                sl->sl_meta_blocksize_shift = 0;
1915                goto over_meta_create;
1916        }
1917        if ((ret = lookupname(sl->sl_meta_filename, UIO_SYSSPACE, FOLLOW,
1918            NULLVPP, &sl->sl_meta_vp)) != 0) {
1919                *err_ret = SBD_RET_META_FILE_LOOKUP_FAILED;
1920                goto scm_err_out;
1921        }
1922        sl->sl_meta_vtype = vt = sl->sl_meta_vp->v_type;
1923        VN_RELE(sl->sl_meta_vp);
1924        if ((vt != VREG) && (vt != VCHR) && (vt != VBLK)) {
1925                *err_ret = SBD_RET_WRONG_META_FILE_TYPE;
1926                ret = EINVAL;
1927                goto scm_err_out;
1928        }
1929        if (vt == VREG) {
1930                sl->sl_meta_blocksize_shift = 0;
1931        } else {
1932                sl->sl_meta_blocksize_shift = 9;
1933        }
1934        flag = FREAD | FWRITE | FOFFMAX | FEXCL;
1935        if ((ret = vn_open(sl->sl_meta_filename, UIO_SYSSPACE, flag, 0,
1936            &sl->sl_meta_vp, 0, 0)) != 0) {
1937                *err_ret = SBD_RET_META_FILE_OPEN_FAILED;
1938                goto scm_err_out;
1939        }
1940 over_meta_create:
1941        sl->sl_total_meta_size = sl->sl_meta_offset + sizeof (sbd_meta_start_t);
1942        sl->sl_total_meta_size +=
1943            (((uint64_t)1) << sl->sl_meta_blocksize_shift) - 1;
1944        sl->sl_total_meta_size &=
1945            ~((((uint64_t)1) << sl->sl_meta_blocksize_shift) - 1);
1946        sl->sl_meta_size_used = 0;
1947 over_meta_open:
1948        sl->sl_flags |= SL_META_OPENED;

1950        sl->sl_device_id[3] = 16;
1951        if (slu->slu_guid_valid) {
1952                sl->sl_device_id[0] = 0xf1;
1953                sl->sl_device_id[1] = 3;
1954                sl->sl_device_id[2] = 0;
1955                bcopy(slu->slu_guid, sl->sl_device_id + 4, 16);
1956        } else {
1957                if (slu->slu_host_id_valid)
1958                        hid = slu->slu_host_id;
1959                if (!slu->slu_company_id_valid)
1960                        slu->slu_company_id = COMPANY_ID_SUN;
1961                if (stmf_scsilib_uniq_lu_id2(slu->slu_company_id, hid,
1962                    (scsi_devid_desc_t *)&sl->sl_device_id[0]) !=
1963                    STMF_SUCCESS) {
1964                        *err_ret = SBD_RET_META_CREATION_FAILED;
1965                        ret = EIO;
1966                        goto scm_err_out;
1967                }
1968                bcopy(sl->sl_device_id + 4, slu->slu_guid, 16);
1969        }

1971        /* Lets create the meta now */
1972        mutex_enter(&sl->sl_metadata_lock);
```

```
1973         if (sbd_write_meta_start(sl, sl->sl_total_meta_size,
1974             sizeof (sbd_meta_start_t)) != SBD_SUCCESS) {
1975                 mutex_exit(&sl->sl_metadata_lock);
1976                 *err_ret = SBD_RET_META_CREATION_FAILED;
1977                 ret = EIO;
1978                 goto scm_err_out;
1979         }
1980         mutex_exit(&sl->sl_metadata_lock);
1981         sl->sl_meta_size_used = sl->sl_meta_offset + sizeof (sbd_meta_start_t);

1983         if (sbd_write_lu_info(sl) != SBD_SUCCESS) {
1984                 *err_ret = SBD_RET_META_CREATION_FAILED;
1985                 ret = EIO;
1986                 goto scm_err_out;
1987         }

1989         if (sbd_pgr_meta_init(sl) != SBD_SUCCESS) {
1990                 *err_ret = SBD_RET_META_CREATION_FAILED;
1991                 ret = EIO;
1992                 goto scm_err_out;
1993         }

1995         /*
1996          * Update the zvol separately as this need only be called upon
1997          * completion of the metadata initialization.
1998          */
1999         if (sl->sl_flags & SL_ZFS_META) {
2000                 if (sbd_update_zfs_prop(sl) != SBD_SUCCESS) {
2001                         *err_ret = SBD_RET_META_CREATION_FAILED;
2002                         ret = EIO;
2003                         goto scm_err_out;
2004                 }
2005         }

2007         ret = sbd_populate_and_register_lu(sl, err_ret);
2008         if (ret) {
2009                 goto scm_err_out;
2010         }

2012         sl->sl_trans_op = SL_OP_NONE;
2013         atomic_inc_32(&sbd_lu_count);
2013         atomic_add_32(&sbd_lu_count, 1);
2014         return (0);

2016 scm_err_out:
2017         return (sbd_close_delete_lu(sl, ret));
2018 }
```

*unchanged_portion_omitted_*

```
2132 int
2133 sbd_create_standby_lu(sbd_create_standby_lu_t *slu, uint32_t *err_ret)
2134 {
2135         sbd_lu_t *sl;
2136         stmf_lu_t *lu;
2137         int ret = EIO;
2138         int alloc_sz;

2140         alloc_sz = sizeof (sbd_lu_t) + sizeof (sbd_pgr_t) +
2141             slu->stlu_meta_fname_size;
2142         lu = (stmf_lu_t *)stmf_alloc(STMF_STRUCT_STMF_LU, alloc_sz, 0);
2143         if (lu == NULL) {
2144                 return (ENOMEM);
2145         }
2146         sl = (sbd_lu_t *)lu->lu_provider_private;
2147         bzero(sl, alloc_sz);
2148         sl->sl_lu = lu;
```



```
2149         sl->sl_alloc_size = alloc_sz;

2151         sl->sl_pgr = (sbd_pgr_t *)(sl + 1);
2152         sl->sl_meta_filename = ((char *)sl) + sizeof (sbd_lu_t) +
2153             sizeof (sbd_pgr_t);

2155         if (slu->stlu_meta_fname_size > 0) {
2156                 (void) strcpy(sl->sl_meta_filename, slu->stlu_meta_fname);
2157         }
2158         sl->sl_name = sl->sl_meta_filename;

2160         sl->sl_device_id[3] = 16;
2161         sl->sl_device_id[0] = 0xf1;
2162         sl->sl_device_id[1] = 3;
2163         sl->sl_device_id[2] = 0;
2164         bcopy(slu->stlu_guid, sl->sl_device_id + 4, 16);
2165         lu->lu_id = (scsi_devid_desc_t *)sl->sl_device_id;
2166         sl->sl_access_state = SBD_LU_STANDBY;

2168         rw_init(&sl->sl_pgr->pgr_lock, NULL, RW_DRIVER, NULL);
2169         mutex_init(&sl->sl_lock, NULL, MUTEX_DRIVER, NULL);
2170         mutex_init(&sl->sl_metadata_lock, NULL, MUTEX_DRIVER, NULL);
2171         rw_init(&sl->sl_access_state_lock, NULL, RW_DRIVER, NULL);

2173         sl->sl_trans_op = SL_OP_CREATE_REGISTER_LU;

2175         if (sbd_link_lu(sl) != SBD_SUCCESS) {
2176                 *err_ret = SBD_RET_FILE_ALREADY_REGISTERED;
2177                 ret = EALREADY;
2178                 goto scs_err_out;
2179         }

2181         ret = sbd_populate_and_register_lu(sl, err_ret);
2182         if (ret) {
2183                 goto scs_err_out;
2184         }

2186         sl->sl_trans_op = SL_OP_NONE;
2187         atomic_inc_32(&sbd_lu_count);
2187         atomic_add_32(&sbd_lu_count, 1);
2188         return (0);

2190 scs_err_out:
2191         return (sbd_close_delete_lu(sl, ret));
2192 }
```

*unchanged_portion_omitted_*

```
2227 int
2228 sbd_import_lu(sbd_import_lu_t *ilu, int struct_sz, uint32_t *err_ret,
2229     int no_register, sbd_lu_t **slr)
2230 {
2231         stmf_lu_t *lu;
2232         sbd_lu_t *sl;
2233         sbd_lu_info_1_1_t *sli = NULL;
2234         int asz;
2235         int ret = 0;
2236         stmf_status_t stret;
2237         int flag;
2238         int wcd = 0;
2239         int data_opened;
2240         uint16_t sli_buf_sz;
2241         uint8_t *sli_buf_copy = NULL;
2242         enum vtype vt;
2243         int standby = 0;
2244         sbd_status_t sret;
```

```
2246         if (no_register && slr == NULL) {
2247                 return (EINVAL);
2248         }
2249         ilu->ilu_meta_fname[struct_sz - sizeof (*ilu) + 8 - 1] = 0;
2250         /*
2251          * check whether logical unit is already registered ALUA
2252          * For a standby logical unit, the meta filename is set. Use
2253          * that to search for an existing logical unit.
2254          */
2255         sret = sbd_find_and_lock_lu(NULL, (uint8_t *)&(ilu->ilu_meta_fname),
2256             SL_OP_IMPORT_LU, &sl);

2258         if (sret == SBD_SUCCESS) {
2259                 if (sl->sl_access_state != SBD_LU_ACTIVE) {
2260                         no_register = 1;
2261                         standby = 1;
2262                         lu = sl->sl_lu;
2263                         if (sl->sl_alias_alloc_size) {
2264                                 kmem_free(sl->sl_alias,
2265                                     sl->sl_alias_alloc_size);
2266                                 sl->sl_alias_alloc_size = 0;
2267                                 sl->sl_alias = NULL;
2268                                 lu->lu_alias = NULL;
2269                         }
2270                         if (sl->sl_meta_filename == NULL) {
2271                                 sl->sl_meta_filename = sl->sl_data_filename;
2272                         } else if (sl->sl_data_fname_alloc_size) {
2273                                 kmem_free(sl->sl_data_filename,
2274                                     sl->sl_data_fname_alloc_size);
2275                                 sl->sl_data_fname_alloc_size = 0;
2276                         }
2277                         if (sl->sl_serial_no_alloc_size) {
2278                                 kmem_free(sl->sl_serial_no,
2279                                     sl->sl_serial_no_alloc_size);
2280                                 sl->sl_serial_no_alloc_size = 0;
2281                         }
2282                         if (sl->sl_mgmt_url_alloc_size) {
2283                                 kmem_free(sl->sl_mgmt_url,
2284                                     sl->sl_mgmt_url_alloc_size);
2285                                 sl->sl_mgmt_url_alloc_size = 0;
2286                         }
2287                 } else {
2288                         *err_ret = SBD_RET_FILE_ALREADY_REGISTERED;
2289                         bcopy(sl->sl_device_id + 4, ilu->ilu_ret_guid, 16);
2290                         sl->sl_trans_op = SL_OP_NONE;
2291                         return (EALREADY);
2292                 }
2293         } else if (sret == SBD_NOT_FOUND) {
2294                 asz = strlen(ilu->ilu_meta_fname) + 1;

2296                 lu = (stmf_lu_t *)stmf_alloc(STMF_STRUCT_STMF_LU,
2297                     sizeof (sbd_lu_t) + sizeof (sbd_pgr_t) + asz, 0);
2298                 if (lu == NULL) {
2299                         return (ENOMEM);
2300                 }
2301                 sl = (sbd_lu_t *)lu->lu_provider_private;
2302                 bzero(sl, sizeof (*sl));
2303                 sl->sl_lu = lu;
2304                 sl->sl_pgr = (sbd_pgr_t *)(sl + 1);
2305                 sl->sl_meta_filename = ((char *)sl) + sizeof (*sl) +
2306                     sizeof (sbd_pgr_t);
2307                 (void) strcpy(sl->sl_meta_filename, ilu->ilu_meta_fname);
2308                 sl->sl_name = sl->sl_meta_filename;
2309                 rw_init(&sl->sl_pgr->pgr_lock, NULL, RW_DRIVER, NULL);
2310                 rw_init(&sl->sl_access_state_lock, NULL, RW_DRIVER, NULL);
2311                 mutex_init(&sl->sl_lock, NULL, MUTEX_DRIVER, NULL);
```



```
2312                 mutex_init(&sl->sl_metadata_lock, NULL, MUTEX_DRIVER, NULL);
2313                 sl->sl_trans_op = SL_OP_IMPORT_LU;
2314         } else {
2315                 *err_ret = SBD_RET_META_FILE_LOOKUP_FAILED;
2316                 return (EIO);
2317         }

2319         /* we're only loading the metadata */
2320         if (!no_register) {
2321                 if (sbd_link_lu(sl) != SBD_SUCCESS) {
2322                         *err_ret = SBD_RET_FILE_ALREADY_REGISTERED;
2323                         bcopy(sl->sl_device_id + 4, ilu->ilu_ret_guid, 16);
2324                         ret = EALREADY;
2325                         goto sim_err_out;
2326                 }
2327         }
2328         if ((ret = lookupname(sl->sl_meta_filename, UIO_SYSSPACE, FOLLOW,
2329             NULLVPP, &sl->sl_meta_vp)) != 0) {
2330                 *err_ret = SBD_RET_META_FILE_LOOKUP_FAILED;
2331                 goto sim_err_out;
2332         }
2333         if (sbd_is_zvol(sl->sl_meta_filename)) {
2334                 sl->sl_flags |= SL_ZFS_META;
2335                 sl->sl_data_filename = sl->sl_meta_filename;
2336         }
2337         sl->sl_meta_vtype = vt = sl->sl_meta_vp->v_type;
2338         VN_RELE(sl->sl_meta_vp);
2339         if ((vt != VREG) && (vt != VCHR) && (vt != VBLK)) {
2340                 *err_ret = SBD_RET_WRONG_META_FILE_TYPE;
2341                 ret = EINVAL;
2342                 goto sim_err_out;
2343         }
2344         if (sl->sl_flags & SL_ZFS_META) {
2345                 if (sbd_open_zfs_meta(sl) != SBD_SUCCESS) {
2346                         /* let see if metadata is in the 64k block */
2347                         sl->sl_flags &= ~SL_ZFS_META;
2348                 }
2349         }
2350         if (!(sl->sl_flags & SL_ZFS_META)) {
2351                 /* metadata is always writable */
2352                 flag = FREAD | FWRITE | FOFFMAX | FEXCL;
2353                 if ((ret = vn_open(sl->sl_meta_filename, UIO_SYSSPACE, flag, 0,
2354                     &sl->sl_meta_vp, 0, 0)) != 0) {
2355                         *err_ret = SBD_RET_META_FILE_OPEN_FAILED;
2356                         goto sim_err_out;
2357                 }
2358         }
2359         if ((sl->sl_flags & SL_ZFS_META) || (vt == VREG)) {
2360                 sl->sl_meta_blocksize_shift = 0;
2361         } else {
2362                 sl->sl_meta_blocksize_shift = 9;
2363         }
2364         sl->sl_meta_offset = (sl->sl_flags & SL_ZFS_META) ? 0 : SBD_META_OFFSET;
2365         sl->sl_flags |= SL_META_OPENED;

2367         mutex_enter(&sl->sl_metadata_lock);
2368         sret = sbd_load_meta_start(sl);
2369         mutex_exit(&sl->sl_metadata_lock);
2370         if (sret != SBD_SUCCESS) {
2371                 if (sret == SBD_META_CORRUPTED) {
2372                         *err_ret = SBD_RET_NO_META;
2373                 } else if (sret == SBD_NOT_SUPPORTED) {
2374                         *err_ret = SBD_RET_VERSION_NOT_SUPPORTED;
2375                 } else {
2376                         *err_ret = SBD_RET_NO_META;
2377                 }
```

```
2378                 ret = EINVAL;
2379                 goto sim_err_out;
2380         }

2382         /* Now lets see if we can read the most recent LU info */
2383         sret = sbd_read_meta_section(sl, (sm_section_hdr_t **)&sli,
2384             SMS_ID_LU_INFO_1_1);
2385         if ((sret == SBD_NOT_FOUND) && ((sl->sl_flags & SL_ZFS_META) == 0)) {
2386                 ret = sbd_load_sli_1_0(sl, err_ret);
2387                 if (ret) {
2388                         goto sim_err_out;
2389                 }
2390                 goto sim_sli_loaded;
2391         }
2392         if (sret != SBD_SUCCESS) {
2393                 *err_ret = SBD_RET_NO_META;
2394                 ret = EIO;
2395                 goto sim_err_out;
2396         }
2397         /* load sli 1.1 */
2398         if (sli->sli_data_order != SMS_DATA_ORDER) {
2399                 sbd_swap_lu_info_1_1(sli);
2400                 if (sli->sli_data_order != SMS_DATA_ORDER) {
2401                         *err_ret = SBD_RET_NO_META;
2402                         ret = EIO;
2403                         goto sim_err_out;
2404                 }
2405         }

2407         sli_buf_sz = sli->sli_sms_header.sms_size -
2408             sizeof (sbd_lu_info_1_1_t) + 8;
2409         sli_buf_copy = kmem_alloc(sli_buf_sz + 1, KM_SLEEP);
2410         bcopy(sli->sli_buf, sli_buf_copy, sli_buf_sz);
2411         sli_buf_copy[sli_buf_sz] = 0;

2413         /* Make sure all the offsets are within limits */
2414         if (((sli->sli_flags & SLI_META_FNAME_VALID) &&
2415             (sli->sli_meta_fname_offset > sli_buf_sz)) ||
2416             ((sli->sli_flags & SLI_DATA_FNAME_VALID) &&
2417             (sli->sli_data_fname_offset > sli_buf_sz)) ||
2418             ((sli->sli_flags & SLI_MGMT_URL_VALID) &&
2419             (sli->sli_mgmt_url_offset > sli_buf_sz)) ||
2420             ((sli->sli_flags & SLI_SERIAL_VALID) &&
2421             ((sli->sli_serial_offset + sli->sli_serial_size) > sli_buf_sz)) ||
2422             ((sli->sli_flags & SLI_ALIAS_VALID) &&
2423             (sli->sli_alias_offset > sli_buf_sz))) {
2424                 *err_ret = SBD_RET_NO_META;
2425                 ret = EIO;
2426                 goto sim_err_out;
2427         }

2429         sl->sl_lu_size = sli->sli_lu_size;
2430         sl->sl_data_blocksize_shift = sli->sli_data_blocksize_shift;
2431         bcopy(sli->sli_device_id, sl->sl_device_id, 20);
2432         if (sli->sli_flags & SLI_SERIAL_VALID) {
2433                 sl->sl_serial_no_size = sl->sl_serial_no_alloc_size =
2434                     sli->sli_serial_size;
2435                 sl->sl_serial_no = kmem_zalloc(sli->sli_serial_size, KM_SLEEP);
2436                 bcopy(sli_buf_copy + sli->sli_serial_offset, sl->sl_serial_no,
2437                     sl->sl_serial_no_size);
2438         }
2439         if (sli->sli_flags & SLI_SEPARATE_META) {
2440                 sl->sl_total_data_size = sl->sl_lu_size;
2441                 if (sli->sli_flags & SLI_DATA_FNAME_VALID) {
2442                         sl->sl_data_fname_alloc_size = strlen((char *)
2443                             sli_buf_copy + sli->sli_data_fname_offset) + 1;
```



```
2444                         sl->sl_data_filename = kmem_zalloc(
2445                             sl->sl_data_fname_alloc_size, KM_SLEEP);
2446                         (void) strcpy(sl->sl_data_filename,
2447                             (char *)sli_buf_copy + sli->sli_data_fname_offset);
2448                 }
2449         } else {
2450                 if (sl->sl_flags & SL_ZFS_META) {
2451                         sl->sl_total_data_size = sl->sl_lu_size;
2452                         sl->sl_data_offset = 0;
2453                 } else {
2454                         sl->sl_total_data_size =
2455                             sl->sl_lu_size + SHARED_META_DATA_SIZE;
2456                         sl->sl_data_offset = SHARED_META_DATA_SIZE;
2457                         sl->sl_flags |= SL_SHARED_META;
2458                 }
2459         }
2460         if (sli->sli_flags & SLI_ALIAS_VALID) {
2461                 sl->sl_alias_alloc_size = strlen((char *)sli_buf_copy +
2462                     sli->sli_alias_offset) + 1;
2463                 sl->sl_alias = kmem_alloc(sl->sl_alias_alloc_size, KM_SLEEP);
2464                 (void) strcpy(sl->sl_alias, (char *)sli_buf_copy +
2465                     sli->sli_alias_offset);
2466         }
2467         if (sli->sli_flags & SLI_MGMT_URL_VALID) {
2468                 sl->sl_mgmt_url_alloc_size = strlen((char *)sli_buf_copy +
2469                     sli->sli_mgmt_url_offset) + 1;
2470                 sl->sl_mgmt_url = kmem_alloc(sl->sl_mgmt_url_alloc_size,
2471                     KM_SLEEP);
2472                 (void) strcpy(sl->sl_mgmt_url, (char *)sli_buf_copy +
2473                     sli->sli_mgmt_url_offset);
2474         }
2475         if (sli->sli_flags & SLI_WRITE_PROTECTED) {
2476                 sl->sl_flags |= SL_WRITE_PROTECTED;
2477         }
2478         if (sli->sli_flags & SLI_VID_VALID) {
2479                 sl->sl_flags |= SL_VID_VALID;
2480                 bcopy(sli->sli_vid, sl->sl_vendor_id, 8);
2481         }
2482         if (sli->sli_flags & SLI_PID_VALID) {
2483                 sl->sl_flags |= SL_PID_VALID;
2484                 bcopy(sli->sli_pid, sl->sl_product_id, 16);
2485         }
2486         if (sli->sli_flags & SLI_REV_VALID) {
2487                 sl->sl_flags |= SL_REV_VALID;
2488                 bcopy(sli->sli_rev, sl->sl_revision, 4);
2489         }
2490         if (sli->sli_flags & SLI_WRITEBACK_CACHE_DISABLE) {
2491                 sl->sl_flags |= SL_WRITEBACK_CACHE_DISABLE;
2492         }
2493 sim_sli_loaded:
2494         if ((sl->sl_flags & SL_SHARED_META) == 0) {
2495                 data_opened = 0;
2496         } else {
2497                 data_opened = 1;
2498                 sl->sl_data_filename = sl->sl_meta_filename;
2499                 sl->sl_data_vp = sl->sl_meta_vp;
2500                 sl->sl_data_vtype = sl->sl_meta_vtype;
2501         }

2503         sret = sbd_pgr_meta_load(sl);
2504         if (sret != SBD_SUCCESS) {
2505                 *err_ret = SBD_RET_NO_META;
2506                 ret = EIO;
2507                 goto sim_err_out;
2508         }
```

```
2510         ret = sbd_open_data_file(sl, err_ret, 1, data_opened, 0);
2511         if (ret) {
2512                 goto sim_err_out;
2513         }

2515         /*
2516          * set write cache disable on the device
2517          * Note: this shouldn't fail on import unless the cache capabilities
2518          * of the device changed. If that happened, modify will need to
2519          * be used to set the cache flag appropriately after import is done.
2520          */
2521         if (sl->sl_flags & SL_WRITEBACK_CACHE_DISABLE) {
2522                 (void) sbd_wcd_set(1, sl);
2523                 wcd = 1;
2524         /*
2525          * if not explicitly set, attempt to set it to enable, if that fails
2526          * get the current setting and use that
2527          */
2528         } else {
2529                 sret = sbd_wcd_set(0, sl);
2530                 if (sret != SBD_SUCCESS) {
2531                         sbd_wcd_get(&wcd, sl);
2532                 }
2533         }

2535         if (wcd) {
2536                 sl->sl_flags |= SL_WRITEBACK_CACHE_DISABLE |
2537                     SL_SAVED_WRITE_CACHE_DISABLE;
2538         }

2540         /* we're only loading the metadata */
2541         if (!no_register) {
2542                 ret = sbd_populate_and_register_lu(sl, err_ret);
2543                 if (ret) {
2544                         goto sim_err_out;
2545                 }
2546                 atomic_inc_32(&sbd_lu_count);
2546                 atomic_add_32(&sbd_lu_count, 1);
2547         }

2549         bcopy(sl->sl_device_id + 4, ilu->ilu_ret_guid, 16);
2550         sl->sl_trans_op = SL_OP_NONE;

2552         if (sli) {
2553                 kmem_free(sli, sli->sli_sms_header.sms_size);
2554                 sli = NULL;
2555         }
2556         if (sli_buf_copy) {
2557                 kmem_free(sli_buf_copy, sli_buf_sz + 1);
2558                 sli_buf_copy = NULL;
2559         }
2560         if (no_register && !standby) {
2561                 *slr = sl;
2562         }

2564         /*
2565          * if this was imported from standby, set the access state
2566          * to active.
2567          */
2568         if (standby) {
2569                 sbd_it_data_t *it;
2570                 mutex_enter(&sl->sl_lock);
2571                 sl->sl_access_state = SBD_LU_ACTIVE;
2572                 for (it = sl->sl_it_list; it != NULL;
2573                     it = it->sbd_it_next) {
2574                         it->sbd_it_ua_conditions |=
```



```
2575                             SBD_UA_ASYMMETRIC_ACCESS_CHANGED;
2576                         it->sbd_it_ua_conditions |= SBD_UA_POR;
2577                         it->sbd_it_flags |=  SBD_IT_PGR_CHECK_FLAG;
2578                 }
2579                 mutex_exit(&sl->sl_lock);
2580                 /* call set access state */
2581                 stret = stmf_set_lu_access(lu, STMF_LU_ACTIVE);
2582                 if (stret != STMF_SUCCESS) {
2583                         *err_ret = SBD_RET_ACCESS_STATE_FAILED;
2584                         sl->sl_access_state = SBD_LU_STANDBY;
2585                         goto sim_err_out;
2586                 }
2587                 if (sl->sl_alias) {
2588                         lu->lu_alias = sl->sl_alias;
2589                 } else {
2590                         lu->lu_alias = sl->sl_name;
2591                 }
2592         }
2593         sl->sl_access_state = SBD_LU_ACTIVE;
2594         return (0);

2596 sim_err_out:
2597         if (sli) {
2598                 kmem_free(sli, sli->sli_sms_header.sms_size);
2599                 sli = NULL;
2600         }
2601         if (sli_buf_copy) {
2602                 kmem_free(sli_buf_copy, sli_buf_sz + 1);
2603                 sli_buf_copy = NULL;
2604         }

2606         if (standby) {
2607                 *err_ret = SBD_RET_ACCESS_STATE_FAILED;
2608                 sl->sl_trans_op = SL_OP_NONE;
2609                 return (EIO);
2610         } else {
2611                 return (sbd_close_delete_lu(sl, ret));
2612         }
2613 }
_____unchanged_portion_omitted_

2917 /* ARGSUSED */
2918 int
2919 sbd_delete_locked_lu(sbd_lu_t *sl, uint32_t *err_ret,
2920     stmf_state_change_info_t *ssi)
2921 {
2922         int i;
2923         stmf_status_t ret;

2925         if ((sl->sl_state == STMF_STATE_OFFLINE) &&
2926             !sl->sl_state_not_acked) {
2927                 goto sdl_do_dereg;
2928         }

2930         if ((sl->sl_state != STMF_STATE_ONLINE) ||
2931             sl->sl_state_not_acked) {
2932                 return (EBUSY);
2933         }

2935         ret = stmf_ctl(STMF_CMD_LU_OFFLINE, sl->sl_lu, ssi);
2936         if ((ret != STMF_SUCCESS) && (ret != STMF_ALREADY)) {
2937                 return (EBUSY);
2938         }

2940         for (i = 0; i < 500; i++) {
2941                 if ((sl->sl_state == STMF_STATE_OFFLINE) &&
```

```
2942                     !sl->sl_state_not_acked) {
2943                         goto sdl_do_dereg;
2944                 }
2945                 delay(drv_usectohz(10000));
2946         }
2947         return (EBUSY);

2949 sdl_do_dereg:;
2950         if (stmf_deregister_lu(sl->sl_lu) != STMF_SUCCESS)
2951                 return (EBUSY);
2952         atomic_dec_32(&sbd_lu_count);
2952         atomic_add_32(&sbd_lu_count, -1);

2954         return (sbd_close_delete_lu(sl, 0));
2955 }
```

_____unchanged_portion_omitted_

****
8187 Mon Jul 28 07:44:32 2014
new/usr/src/uts/common/io/comstar/port/fcoet/fcoet.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____unchanged_portion_omitted_

```
 215 /*
 216  * Add the reference to avoid such situation:
 217  * 1, Frame received, then abort happen (maybe because local port offline, or
 218  * remote port abort the cmd), cmd is aborted and then freed right after we
 219  * get the exchange from hash table in fcoet_rx_frame.
 220  * 2, Frame sent out, then queued in fcoe for release. then abort happen, cmd
 221  * is aborted and then freed before fcoe_watchdog() call up to release the
 222  * frame.
 223  * These two situation should seldom happen. But just invoke this seems won't
 224  * downgrade the performance too much, so we keep it.
 225  */
 226 #define FCOET_BUSY_XCHG(xch)    atomic_inc_8(&(xch)->xch_ref)
 227 #define FCOET_RELE_XCHG(xch)    atomic_dec_8(&(xch)->xch_ref)
 226 #define FCOET_BUSY_XCHG(xch)    atomic_add_8(&(xch)->xch_ref, 1)
 227 #define FCOET_RELE_XCHG(xch)    atomic_add_8(&(xch)->xch_ref, -1)

 229 #define XCH_FLAG_NONFCP_REQ_SENT        0x0001
 230 #define XCH_FLAG_NONFCP_RESP_SENT       0x0002
 231 #define XCH_FLAG_FCP_CMD_RCVD           0x0004
 232 #define XCH_FLAG_INI_ASKED_ABORT        0x0008
 233 #define XCH_FLAG_FCT_CALLED_ABORT       0x0010
 234 #define XCH_FLAG_IN_HASH_TABLE          0x0020

 236 /*
 237  * IOCTL supporting stuff
 238  */
 239 #define FCOET_IOCTL_FLAG_MASK           0xFF
 240 #define FCOET_IOCTL_FLAG_IDLE           0x00
 241 #define FCOET_IOCTL_FLAG_OPEN           0x01
 242 #define FCOET_IOCTL_FLAG_EXCL           0x02

 244 /*
 245  * define common-used conversion and calculation macros
 246  */
 247 #define FRM2SS(x_frm)                                                   \
 248         ((fcoet_soft_state_t *)(x_frm)->frm_eport->eport_client_private)
 249 #define FRM2TFM(x_frm)  ((fcoet_frame_t *)(x_frm)->frm_client_private)

 251 #define PORT2SS(x_port) ((fcoet_soft_state_t *)(x_port)->port_fca_private)
 252 #define EPORT2SS(x_port) ((fcoet_soft_state_t *)(x_port)->eport_client_private)

 254 #define XCH2ELS(x_xch)  ((fct_els_t *)x_xch->xch_cmd->cmd_specific)
 255 #define XCH2CT(x_xch)   ((fct_ct_t *)x_xch->xch_cmd->cmd_specific)
 256 #define XCH2TASK(x_xch) ((scsi_task_t *)x_xch->xch_cmd->cmd_specific)

 258 #define CMD2ELS(x_cmd)  ((fct_els_t *)x_cmd->cmd_specific)
 259 #define CMD2CT(x_cmd)   ((fct_sol_ct_t *)x_cmd->cmd_specific)
 260 #define CMD2TASK(x_cmd) ((scsi_task_t *)x_cmd->cmd_specific)
 261 #define CMD2XCH(x_cmd)  ((fcoet_exchange_t *)x_cmd->cmd_fca_private)
 262 #define CMD2SS(x_cmd)                                                   \
 263         ((fcoet_soft_state_t *)(x_cmd)->cmd_port->port_fca_private)

 265 void fcoet_init_tfm(fcoe_frame_t *frm, fcoet_exchange_t *xch);
 266 fct_status_t fcoet_send_status(fct_cmd_t *cmd);
 267 void fcoet_modhash_find_cb(mod_hash_key_t, mod_hash_val_t);

 269 /*
 270  * DBUF stuff
 271  */
 272 #define FCOET_DB_SEG_NUM(x_db) (x_db->db_port_private)
```



```
 273 #define FCOET_DB_NETB(x_db)                                             \
 274         (((uintptr_t)FCOET_DB_SEG_NUM(x_db)) *                  \
 275         sizeof (struct stmf_sglist_ent) + (uintptr_t)(x_db)->db_sglist)

 277 #define FCOET_SET_SEG_NUM(x_db, x_num)                  \
 278         {                                               \
 279                 FCOET_DB_SEG_NUM(x_db) = (void *)(unsigned long)x_num;  \
 280         }

 282 #define FCOET_GET_SEG_NUM(x_db) ((int)(unsigned long)FCOET_DB_SEG_NUM(x_db))


 285 #define FCOET_SET_NETB(x_db, x_idx, x_netb)                             \
 286         {                                                               \
 287                 ((void **)FCOET_DB_NETB(x_db))[x_idx] = x_netb; \
 288         }

 290 #define FCOET_GET_NETB(x_db, x_idx)             \
 291         (((void **)FCOET_DB_NETB(x_db))[x_idx])

 293 #define PRT_FRM_HDR(x_p, x_f)                                           \
 294         {                                                               \
 295                 FCOET_LOG(x_p, "rctl/%x, type/%x, fctl/%x, oxid/%x",    \
 296                     FCOE_B2V_1((x_f)->frm_hdr->hdr_r_ctl),              \
 297                     FCOE_B2V_1((x_f)->frm_hdr->hdr_type),               \
 298                     FCOE_B2V_3((x_f)->frm_hdr->hdr_f_ctl),              \
 299                     FCOE_B2V_4((x_f)->frm_hdr->hdr_oxid));              \
 300         }

 302 #endif  /* _KERNEL */

 304 #ifdef  __cplusplus
 305 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   27202 Mon Jul 28 07:44:32 2014
new/usr/src/uts/common/io/comstar/port/fcoet/fcoet_fc.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 208 /*
 209  * It's for read/write (xfer_rdy)
 210  */
 211 /* ARGSUSED */
 212 fct_status_t
 213 fcoet_xfer_scsi_data(fct_cmd_t *cmd, stmf_data_buf_t *dbuf, uint32_t ioflags)
 214 {
 215         fcoe_frame_t    *frm;
 216         int              idx;
 217         int              frm_num;
 218         int              data_size;
 219         int              left_size;
 220         int              offset;
 221         fcoet_exchange_t *xch = CMD2XCH(cmd);

 223         ASSERT(!xch->xch_dbufs[dbuf->db_relative_offset/FCOET_MAX_DBUF_LEN]);
 224         xch->xch_dbufs[dbuf->db_relative_offset/FCOET_MAX_DBUF_LEN] = dbuf;

 226         left_size = (int)dbuf->db_data_size;
 227         if (dbuf->db_relative_offset == 0)
 228                 xch->xch_left_data_size =
 229                     XCH2TASK(xch)->task_expected_xfer_length;

 231         if (dbuf->db_flags & DB_DIRECTION_FROM_RPORT) {
 232                 /*
 233                  * If it's write type command, we need send xfer_rdy now
 234                  * We may need to consider bidirectional command later
 235                  */
 236                 dbuf->db_sglist_length = 0;
 237                 frm = CMD2SS(cmd)->ss_eport->eport_alloc_frame(
 238                     CMD2SS(cmd)->ss_eport, sizeof (fcoe_fcp_xfer_rdy_t) +
 239                     FCFH_SIZE, NULL);
 240                 if (frm == NULL) {
 241                         ASSERT(0);
 242                         return (FCT_FAILURE);
 243                 } else {
 244                         fcoet_init_tfm(frm, CMD2XCH(cmd));
 245                         bzero(frm->frm_payload, frm->frm_payload_size);
 246                 }

 248                 FFM_R_CTL(0x05, frm);
 249                 FRM2TFM(frm)->tfm_rctl = 0x05;
 250                 FFM_TYPE(0x08, frm);
 251                 FFM_F_CTL(0x890000, frm);
 252                 FFM_OXID(cmd->cmd_oxid, frm);
 253                 FFM_RXID(cmd->cmd_rxid, frm);
 254                 FFM_S_ID(cmd->cmd_lportid, frm);
 255                 FFM_D_ID(cmd->cmd_rportid, frm);
 256                 FCOE_V2B_4(dbuf->db_relative_offset, frm->frm_payload);
 257                 FCOE_V2B_4(dbuf->db_data_size, frm->frm_payload + 4);
 258                 CMD2SS(cmd)->ss_eport->eport_tx_frame(frm);

 260                 return (FCT_SUCCESS);
 261         }

 263         /*
 264          * It's time to transfer READ data to remote side
 265          */
 266         frm_num = (dbuf->db_data_size + CMD2SS(cmd)->ss_fcp_data_payload_size -
```



```
 267             1) / CMD2SS(cmd)->ss_fcp_data_payload_size;
 268         offset = dbuf->db_relative_offset;
 269         for (idx = 0; idx < frm_num; idx++) {
 270                 if (idx == (frm_num -1)) {
 271                         data_size = P2ROUNDUP(left_size, 4);
 272                 } else {
 273                         data_size = CMD2SS(cmd)->ss_fcp_data_payload_size;
 274                 }

 276                 frm = CMD2SS(cmd)->ss_eport->eport_alloc_frame(
 277                     CMD2SS(cmd)->ss_eport, data_size + FCFH_SIZE,
 278                     FCOET_GET_NETB(dbuf, idx));
 279                 if (frm == NULL) {
 280                         ASSERT(0);
 281                         return (FCT_FAILURE);
 282                 } else {
 283                         fcoet_init_tfm(frm, CMD2XCH(cmd));
 284                         /*
 285                          * lock the xchg to avoid being released (by abort)
 286                          * after sent out and before release
 287                          */
 288                         FCOET_BUSY_XCHG(CMD2XCH(cmd));
 289                 }

 291                 FFM_R_CTL(0x01, frm);
 292                 FRM2TFM(frm)->tfm_rctl = 0x01;
 293                 FRM2TFM(frm)->tfm_buf_idx =
 294                     dbuf->db_relative_offset/FCOET_MAX_DBUF_LEN;
 295                 FFM_TYPE(0x08, frm);
 296                 if (idx != frm_num - 1) {
 297                         FFM_F_CTL(0x800008, frm);
 298                 } else {
 299                         FFM_F_CTL(0x880008 | (data_size - left_size), frm);
 300                 }

 302                 FFM_OXID(cmd->cmd_oxid, frm);
 303                 FFM_RXID(cmd->cmd_rxid, frm);
 304                 FFM_S_ID(cmd->cmd_lportid, frm);
 305                 FFM_D_ID(cmd->cmd_rportid, frm);
 306                 FFM_SEQ_CNT(xch->xch_sequence_no, frm);
 307                 atomic_inc_8(&xch->xch_sequence_no);
 307                 atomic_add_8(&xch->xch_sequence_no, 1);
 308                 FFM_PARAM(offset, frm);
 309                 offset += data_size;
 310                 left_size -= data_size;

 312                 /*
 313                  * Disassociate netbs which will be freed by NIC driver
 314                  */
 315                 FCOET_SET_NETB(dbuf, idx, NULL);

 317                 CMD2SS(cmd)->ss_eport->eport_tx_frame(frm);
 318         }

 320         return (FCT_SUCCESS);
 321 }
______unchanged_portion_omitted_

 925 static fct_status_t
 926 fcoet_logo_fabric(fcoet_soft_state_t *ss)
 927 {
 928         fcoe_frame_t    *frm;
 929         uint32_t        req_payload_size = 16;
 930         uint16_t        xch_oxid, xch_rxid = 0xFFFF;

 932         frm = ss->ss_eport->eport_alloc_frame(ss->ss_eport,
```

```
 933             req_payload_size + FCFH_SIZE, NULL);
 934         if (frm == NULL) {
 935                 ASSERT(0);
 936                 return (FCT_FAILURE);
 937         } else {
 938                 fcoet_init_tfm(frm, NULL);
 939                 bzero(frm->frm_payload, frm->frm_payload_size);
 940         }
 941         xch_oxid = atomic_inc_16_nv(&ss->ss_next_sol_oxid);
 941         xch_oxid = atomic_add_16_nv(&ss->ss_next_sol_oxid, 1);
 942         if (xch_oxid == 0xFFFF) {
 943                 xch_oxid = atomic_inc_16_nv(&ss->ss_next_sol_oxid);
 943                 xch_oxid = atomic_add_16_nv(&ss->ss_next_sol_oxid, 1);
 944         }
 945         FFM_R_CTL(0x22, frm);
 946         FRM2TFM(frm)->tfm_rctl = 0x22;
 947         FFM_TYPE(0x01, frm);
 948         FFM_F_CTL(0x290000, frm);
 949         FFM_OXID(xch_oxid, frm);
 950         FFM_RXID(xch_rxid, frm);
 951         FFM_S_ID(ss->ss_link_info.portid, frm);
 952         FFM_D_ID(0xfffffe, frm);

 954         FCOE_V2B_1(0x5, frm->frm_payload);
 955         FCOE_V2B_3(ss->ss_link_info.portid, frm->frm_payload + 5);
 956         bcopy(ss->ss_eport->eport_portwwn, frm->frm_payload + 8, 8);
 957         ss->ss_eport->eport_tx_frame(frm);

 959         return (FCT_SUCCESS);

 961 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   82554 Mon Jul 28 07:44:32 2014
new/usr/src/uts/common/io/comstar/port/fct/discovery.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 791 /*
 792  * Handles both solicited and unsolicited elses. Can be called inside
 793  * interrupt context.
 794  */
 795 void
 796 fct_handle_els(fct_cmd_t *cmd)
 797 {
 798         fct_local_port_t        *port = cmd->cmd_port;
 799         fct_i_local_port_t *iport =
 800             (fct_i_local_port_t *)port->port_fct_private;
 801         fct_i_cmd_t             *icmd = (fct_i_cmd_t *)cmd->cmd_fct_private;
 802         fct_els_t               *els  = (fct_els_t *)cmd->cmd_specific;
 803         fct_remote_port_t       *rp;
 804         fct_i_remote_port_t     *irp;
 805         uint16_t                 cmd_slot;
 806         uint8_t                  op;

 808         op = els->els_req_payload[0];
 809         icmd->icmd_start_time = ddi_get_lbolt();
 810         if (cmd->cmd_type == FCT_CMD_RCVD_ELS) {
 811                 icmd->icmd_flags |= ICMD_KNOWN_TO_FCA;
 812         }
 813         stmf_trace(iport->iport_alias, "Posting %ssol ELS %x (%s) rp_id=%x"
 814             " lp_id=%x", (cmd->cmd_type == FCT_CMD_RCVD_ELS) ? "un" : "",
 815             op, FCT_ELS_NAME(op), cmd->cmd_rportid,
 816             cmd->cmd_lportid);

 818         rw_enter(&iport->iport_lock, RW_READER);
 819 start_els_posting:;
 820         /* Make sure local port is sane */
 821         if ((iport->iport_link_state & S_LINK_ONLINE) == 0) {
 822                 rw_exit(&iport->iport_lock);
 823                 stmf_trace(iport->iport_alias, "ELS %x not posted becasue"
 824                     "port state was %x", els->els_req_payload[0],
 825                     iport->iport_link_state);
 826                 fct_queue_cmd_for_termination(cmd, FCT_LOCAL_PORT_OFFLINE);
 827                 return;
 828         }

 830         /* Weed out any bad initiators in case of N2N topology */
 831         if ((cmd->cmd_type == FCT_CMD_RCVD_ELS) &&
 832             (els->els_req_payload[0] == ELS_OP_PLOGI) &&
 833             (iport->iport_link_state == PORT_STATE_LINK_INIT_START) &&
 834             (iport->iport_link_info.port_topology == PORT_TOPOLOGY_PT_TO_PT)) {
 835                 int state;
 836                 int killit = 0;

 838                 mutex_enter(&iport->iport_worker_lock);
 839                 state = iport->iport_li_state & LI_STATE_MASK;
 840                 /*
 841                  * We dont allow remote port to plogi in N2N if we have not yet
 842                  * resolved the topology.
 843                  */
 844                 if (state <= LI_STATE_FINI_TOPOLOGY) {
 845                         killit = 1;
 846                         stmf_trace(iport->iport_alias, "port %x is trying to "
 847                             "PLOGI in N2N topology, While we have not resolved"
 848                             " the topology. Dropping...", cmd->cmd_rportid);
 849                 } else if (state <= LI_STATE_N2N_PLOGI) {
```



```
 850                         if (fct_lport_has_bigger_wwn(iport)) {
 851                                 killit = 1;
 852                                 stmf_trace(iport->iport_alias, "port %x is "
 853                                     "trying to PLOGI in N2N topology, even "
 854                                     "though it has smaller PWWN",
 855                                     cmd->cmd_rportid);
 856                         } else {
 857                                 /*
 858                                  * Remote port is assigning us a PORTID as
 859                                  * a part of PLOGI.
 860                                  */
 861                                 iport->iport_link_info.portid =
 862                                     cmd->cmd_lportid;
 863                         }
 864                 }
 865                 mutex_exit(&iport->iport_worker_lock);
 866                 if (killit) {
 867                         rw_exit(&iport->iport_lock);
 868                         fct_queue_cmd_for_termination(cmd,
 869                             FCT_LOCAL_PORT_OFFLINE);
 870                         return;
 871                 }
 872         }

 874         /*
 875          * For all unsolicited ELSes that are not FLOGIs, our portid
 876          * has been established by now. Sometimes port IDs change due to
 877          * link resets but remote ports may still send ELSes using the
 878          * old IDs. Kill those right here.
 879          */
 880         if ((cmd->cmd_type == FCT_CMD_RCVD_ELS) &&
 881             (els->els_req_payload[0] != ELS_OP_FLOGI)) {
 882                 if (cmd->cmd_lportid != iport->iport_link_info.portid) {
 883                         rw_exit(&iport->iport_lock);
 884                         stmf_trace(iport->iport_alias, "Rcvd %s with "
 885                             "wrong lportid %x, expecting %x. Killing ELS.",
 886                             FCT_ELS_NAME(op), cmd->cmd_lportid,
 887                             iport->iport_link_info.portid);
 888                         fct_queue_cmd_for_termination(cmd,
 889                             FCT_NOT_FOUND);
 890                         return;
 891                 }
 892         }

 894         /*
 895          * We always lookup by portid. port handles are too
 896          * unreliable at this stage.
 897          */
 898         irp = fct_portid_to_portptr(iport, cmd->cmd_rportid);
 899         if (els->els_req_payload[0] == ELS_OP_PLOGI) {
 900                 if (irp == NULL) {
 901                         /* drop the lock while we do allocations */
 902                         rw_exit(&iport->iport_lock);
 903                         rp = fct_alloc(FCT_STRUCT_REMOTE_PORT,
 904                             port->port_fca_rp_private_size, 0);
 905                         if (rp == NULL) {
 906                                 fct_queue_cmd_for_termination(cmd,
 907                                     FCT_ALLOC_FAILURE);
 908                                 return;
 909                         }
 910                         irp = (fct_i_remote_port_t *)rp->rp_fct_private;
 911                         rw_init(&irp->irp_lock, 0, RW_DRIVER, 0);
 912                         irp->irp_rp = rp;
 913                         irp->irp_portid = cmd->cmd_rportid;
 914                         rp->rp_port = port;
 915                         rp->rp_id = cmd->cmd_rportid;
```

```
916                         rp->rp_handle = FCT_HANDLE_NONE;
917                         /*
918                          * Grab port lock as writer since we are going
919                          * to modify the local port struct.
920                          */
921                         rw_enter(&iport->iport_lock, RW_WRITER);
922                         /* Make sure nobody created the struct except us */
923                         if (fct_portid_to_portptr(iport, cmd->cmd_rportid)) {
924                                 /* Oh well, free it */
925                                 fct_free(rp);
926                         } else {
927                                 fct_queue_rp(iport, irp);
928                         }
929                         rw_downgrade(&iport->iport_lock);
930                         /* Start over becasue we dropped the lock */
931                         goto start_els_posting;
932                 }

934                 /* A PLOGI is by default a logout of previous session */
935                 irp->irp_deregister_timer = ddi_get_lbolt() +
936                     drv_usectohz(USEC_DEREG_RP_TIMEOUT);
937                 irp->irp_dereg_count = 0;
938                 fct_post_to_discovery_queue(iport, irp, NULL);

940                 /* A PLOGI also invalidates any RSCNs related to this rp */
941                 atomic_inc_32(&irp->irp_rscn_counter);
941                 atomic_add_32(&irp->irp_rscn_counter, 1);
942         } else {
943                 /*
944                  * For everything else, we have (or be able to lookup) a
945                  * valid port pointer.
946                  */
947                 if (irp == NULL) {
948                         rw_exit(&iport->iport_lock);
949                         if (cmd->cmd_type == FCT_CMD_RCVD_ELS) {
950                                 /* XXX Throw a logout to the initiator */
951                                 stmf_trace(iport->iport_alias, "ELS %x "
952                                     "received from %x without a session",
953                                     els->els_req_payload[0], cmd->cmd_rportid);
954                         } else {
955                                 stmf_trace(iport->iport_alias, "Sending ELS %x "
956                                     "to %x without a session",
957                                     els->els_req_payload[0], cmd->cmd_rportid);
958                         }
959                         fct_queue_cmd_for_termination(cmd, FCT_NOT_LOGGED_IN);
960                         return;
961                 }
962         }
963         cmd->cmd_rp = rp = irp->irp_rp;

965         /*
966          * Lets get a slot for this els
967          */
968         if (!(icmd->icmd_flags & ICMD_IMPLICIT)) {
969                 cmd_slot = fct_alloc_cmd_slot(iport, cmd);
970                 if (cmd_slot == FCT_SLOT_EOL) {
971                         /* This should not have happened */
972                         rw_exit(&iport->iport_lock);
973                         stmf_trace(iport->iport_alias,
974                             "ran out of xchg resources");
975                         fct_queue_cmd_for_termination(cmd,
976                             FCT_NO_XCHG_RESOURCE);
977                         return;
978                 }
979         } else {
980                 /*
```



```
981                  * Tell the framework that fct_cmd_free() can decrement the
982                  * irp_nonfcp_xchg_count variable.
983                  */
984                 atomic_or_32(&icmd->icmd_flags, ICMD_IMPLICIT_CMD_HAS_RESOURCE);
985         }
986         atomic_inc_16(&irp->irp_nonfcp_xchg_count);
986         atomic_add_16(&irp->irp_nonfcp_xchg_count, 1);

988         /*
989          * Grab the remote port lock while we modify the port state.
990          * we should not drop the fca port lock (as a reader) until we
991          * modify the remote port state.
992          */
993         rw_enter(&irp->irp_lock, RW_WRITER);
994         if ((op == ELS_OP_PLOGI) || (op == ELS_OP_PRLI) ||
995             (op == ELS_OP_LOGO) || (op == ELS_OP_PRLO) ||
996             (op == ELS_OP_TPRLO)) {
997                 uint32_t rf = IRP_PRLI_DONE;
998                 if ((op == ELS_OP_PLOGI) || (op == ELS_OP_LOGO)) {
999                         rf |= IRP_PLOGI_DONE;
1000                        if (irp->irp_flags & IRP_PLOGI_DONE)
1001                                atomic_dec_32(&iport->iport_nrps_login);
1001                                atomic_add_32(&iport->iport_nrps_login, -1);
1002                }
1003                atomic_inc_16(&irp->irp_sa_elses_count);
1003                atomic_add_16(&irp->irp_sa_elses_count, 1);
1004                atomic_and_32(&irp->irp_flags, ~rf);
1005                atomic_or_32(&icmd->icmd_flags, ICMD_SESSION_AFFECTING);
1006        } else {
1007                atomic_inc_16(&irp->irp_nsa_elses_count);
1007                atomic_add_16(&irp->irp_nsa_elses_count, 1);
1008        }

1010        fct_post_to_discovery_queue(iport, irp, icmd);

1012        rw_exit(&irp->irp_lock);
1013        rw_exit(&iport->iport_lock);
1014 }
______unchanged_portion_omitted_

1101 fct_status_t
1102 fct_register_remote_port(fct_local_port_t *port, fct_remote_port_t *rp,
1103                                 fct_cmd_t *cmd)
1104 {
1105         fct_status_t ret;
1106         fct_i_local_port_t      *iport;
1107         fct_i_remote_port_t     *irp;
1108         int                     i;
1109         char                    info[FCT_INFO_LEN];

1111         iport = (fct_i_local_port_t *)port->port_fct_private;
1112         irp = (fct_i_remote_port_t *)rp->rp_fct_private;

1114         if ((ret = port->port_register_remote_port(port, rp, cmd)) !=
1115             FCT_SUCCESS)
1116                 return (ret);

1118         rw_enter(&iport->iport_lock, RW_WRITER);
1119         rw_enter(&irp->irp_lock, RW_WRITER);
1120         if (rp->rp_handle != FCT_HANDLE_NONE) {
1121                 if (rp->rp_handle >= port->port_max_logins) {
1122                         (void) snprintf(info, sizeof (info),
1123                             "fct_register_remote_port: FCA "
1124                             "returned a handle (%d) for portid %x which is "
1125                             "out of range (max logins = %d)", rp->rp_handle,
1126                             rp->rp_id, port->port_max_logins);
```

```
1127                        goto hba_fatal_err;
1128                }
1129                if ((iport->iport_rp_slots[rp->rp_handle] != NULL) &&
1130                    (iport->iport_rp_slots[rp->rp_handle] != irp)) {
1131                        fct_i_remote_port_t *t_irp =
1132                            iport->iport_rp_slots[rp->rp_handle];
1133                        (void) snprintf(info, sizeof (info),
1134                            "fct_register_remote_port: "
1135                            "FCA returned a handle %d for portid %x "
1136                            "which was already in use for a different "
1137                            "portid (%x)", rp->rp_handle, rp->rp_id,
1138                            t_irp->irp_rp->rp_id);
1139                        goto hba_fatal_err;
1140                }
1141        } else {
1142                /* Pick a handle for this port */
1143                for (i = 0; i < port->port_max_logins; i++) {
1144                        if (iport->iport_rp_slots[i] == NULL) {
1145                                break;
1146                        }
1147                }
1148                if (i == port->port_max_logins) {
1149                        /* This is really pushing it. */
1150                        (void) snprintf(info, sizeof (info),
1151                            "fct_register_remote_port "
1152                            "Cannot register portid %x because all the "
1153                            "handles are used up", rp->rp_id);
1154                        goto hba_fatal_err;
1155                }
1156                rp->rp_handle = i;
1157        }
1158        /* By this time rport_handle is valid */
1159        if ((irp->irp_flags & IRP_HANDLE_OPENED) == 0) {
1160                iport->iport_rp_slots[rp->rp_handle] = irp;
1161                atomic_or_32(&irp->irp_flags, IRP_HANDLE_OPENED);
1162        }
1163        (void) atomic_inc_64_nv(&iport->iport_last_change);
1163        (void) atomic_add_64_nv(&iport->iport_last_change, 1);
1164        fct_log_remote_port_event(port, ESC_SUNFC_TARGET_ADD,
1165            rp->rp_pwwn, rp->rp_id);

1167 register_rp_done:;
1168        rw_exit(&irp->irp_lock);
1169        rw_exit(&iport->iport_lock);
1170        return (FCT_SUCCESS);

1172 hba_fatal_err:;
1173        rw_exit(&irp->irp_lock);
1174        rw_exit(&iport->iport_lock);
1175        /*
1176         * XXX Throw HBA fatal error event
1177         */
1178        (void) fct_port_shutdown(iport->iport_port,
1179            STMF_RFLAG_FATAL_ERROR | STMF_RFLAG_RESET, info);
1180        return (FCT_FAILURE);
1181 }

1183 fct_status_t
1184 fct_deregister_remote_port(fct_local_port_t *port, fct_remote_port_t *rp)
1185 {
1186        fct_status_t            ret   = FCT_SUCCESS;
1187        fct_i_local_port_t      *iport = PORT_TO_IPORT(port);
1188        fct_i_remote_port_t     *irp   = RP_TO_IRP(rp);

1190        if (irp->irp_snn) {
1191                kmem_free(irp->irp_snn, strlen(irp->irp_snn) + 1);
```



```
1192                irp->irp_snn = NULL;
1193        }
1194        if (irp->irp_spn) {
1195                kmem_free(irp->irp_spn, strlen(irp->irp_spn) + 1);
1196                irp->irp_spn = NULL;
1197        }

1199        if ((ret = port->port_deregister_remote_port(port, rp)) !=
1200            FCT_SUCCESS) {
1201                return (ret);
1202        }

1204        if (irp->irp_flags & IRP_HANDLE_OPENED) {
1205                atomic_and_32(&irp->irp_flags, ~IRP_HANDLE_OPENED);
1206                iport->iport_rp_slots[rp->rp_handle] = NULL;
1207        }
1208        (void) atomic_inc_64_nv(&iport->iport_last_change);
1208        (void) atomic_add_64_nv(&iport->iport_last_change, 1);
1209        fct_log_remote_port_event(port, ESC_SUNFC_TARGET_REMOVE,
1210            rp->rp_pwwn, rp->rp_id);

1212        return (FCT_SUCCESS);
1213 }
______unchanged_portion_omitted_

1377 disc_action_t
1378 fct_process_plogi(fct_i_cmd_t *icmd)
1379 {
1380        fct_cmd_t               *cmd = icmd->icmd_cmd;
1381        fct_remote_port_t       *rp = cmd->cmd_rp;
1382        fct_local_port_t        *port = cmd->cmd_port;
1383        fct_i_local_port_t      *iport = (fct_i_local_port_t *)
1384            port->port_fct_private;
1385        fct_els_t               *els = (fct_els_t *)
1386            cmd->cmd_specific;
1387        fct_i_remote_port_t     *irp = (fct_i_remote_port_t *)
1388            rp->rp_fct_private;
1389        uint8_t                 *p;
1390        fct_status_t             ret;
1391        uint8_t                  cmd_type   = cmd->cmd_type;
1392        uint32_t                 icmd_flags = icmd->icmd_flags;
1393        clock_t                  end_time;
1394        char                     info[FCT_INFO_LEN];

1396        DTRACE_FC_4(rport__login__start,
1397            fct_cmd_t, cmd,
1398            fct_local_port_t, port,
1399            fct_i_remote_port_t, irp,
1400            int, (cmd_type != FCT_CMD_RCVD_ELS));

1402        /* Drain I/Os */
1403        if ((irp->irp_nonfcp_xchg_count + irp->irp_fcp_xchg_count) > 1) {
1404                /* Trigger cleanup if necessary */
1405                if ((irp->irp_flags & IRP_SESSION_CLEANUP) == 0) {
1406                        stmf_trace(iport->iport_alias, "handling PLOGI rp_id"
1407                            " %x. Triggering cleanup", cmd->cmd_rportid);
1408                        /* Cleanup everything except elses */
1409                        if (fct_trigger_rport_cleanup(irp, ~(cmd->cmd_type))) {
1410                                atomic_or_32(&irp->irp_flags,
1411                                    IRP_SESSION_CLEANUP);
1412                        } else {
1413                                /* XXX: handle this */
1414                                /* EMPTY */
1415                        }
1416                }
```

```
1418                 end_time = icmd->icmd_start_time +
1419                     drv_usectohz(USEC_ELS_TIMEOUT);
1420                 if (ddi_get_lbolt() > end_time) {
1421                         (void) snprintf(info, sizeof (info),
1422                             "fct_process_plogi: unable to "
1423                             "clean up I/O. iport-%p, icmd-%p", (void *)iport,
1424                             (void *)icmd);
1425                         (void) fct_port_shutdown(iport->iport_port,
1426                             STMF_RFLAG_FATAL_ERROR | STMF_RFLAG_RESET, info);

1428                         return (DISC_ACTION_DELAY_RESCAN);
1429                 }

1431                 if ((ddi_get_lbolt() & 0x7f) == 0) {
1432                         stmf_trace(iport->iport_alias, "handling"
1433                             " PLOGI rp_id %x, waiting for cmds to"
1434                             " drain", cmd->cmd_rportid);
1435                 }
1436                 return (DISC_ACTION_DELAY_RESCAN);
1437         }
1438         atomic_and_32(&irp->irp_flags, ~IRP_SESSION_CLEANUP);

1440         /* Session can only be terminated after all the I/Os have drained */
1441         if (irp->irp_flags & IRP_SCSI_SESSION_STARTED) {
1442                 stmf_deregister_scsi_session(iport->iport_port->port_lport,
1443                     irp->irp_session);
1444                 stmf_free(irp->irp_session);
1445                 irp->irp_session = NULL;
1446                 atomic_and_32(&irp->irp_flags, ~IRP_SCSI_SESSION_STARTED);
1447         }

1449         if (cmd->cmd_type == FCT_CMD_RCVD_ELS) {
1450                 els->els_resp_size = els->els_req_size;
1451                 p = els->els_resp_payload = (uint8_t *)kmem_zalloc(
1452                     els->els_resp_size, KM_SLEEP);
1453                 els->els_resp_alloc_size = els->els_resp_size;
1454                 bcopy(els->els_req_payload, p, els->els_resp_size);
1455                 p[0] = ELS_OP_ACC;
1456                 bcopy(p+20, rp->rp_pwwn, 8);
1457                 bcopy(p+28, rp->rp_nwwn, 8);
1458                 bcopy(port->port_pwwn, p+20, 8);
1459                 bcopy(port->port_nwwn, p+28, 8);
1460                 fct_wwn_to_str(rp->rp_pwwn_str, rp->rp_pwwn);
1461                 fct_wwn_to_str(rp->rp_nwwn_str, rp->rp_nwwn);
1462                 fct_wwn_to_str(port->port_pwwn_str, port->port_pwwn);
1463                 fct_wwn_to_str(port->port_nwwn_str, port->port_nwwn);

1465                 stmf_wwn_to_devid_desc((scsi_devid_desc_t *)irp->irp_id,
1466                     rp->rp_pwwn, PROTOCOL_FIBRE_CHANNEL);
1467         }

1469         ret = fct_register_remote_port(port, rp, cmd);
1470         fct_dequeue_els(irp);
1471         if ((ret == FCT_SUCCESS) && !(icmd->icmd_flags & ICMD_IMPLICIT)) {
1472                 if (cmd->cmd_type == FCT_CMD_RCVD_ELS) {
1473                         ret = port->port_send_cmd_response(cmd, 0);
1474                         if ((ret == FCT_SUCCESS) && IPORT_IN_NS_TOPO(iport) &&
1475                             !FC_WELL_KNOWN_ADDR(irp->irp_portid)) {
1476                                 fct_cmd_t *ct_cmd = fct_create_solct(port,
1477                                     rp, NS_GSNN_NN, fct_gsnn_cb);
1478                                 if (ct_cmd) {
1479                                         fct_post_to_solcmd_queue(port, ct_cmd);
1480                                 }
1481                                 ct_cmd = fct_create_solct(port, rp,
1482                                     NS_GSPN_ID, fct_gspn_cb);
1483                                 if (ct_cmd)
```



```
1484                                         fct_post_to_solcmd_queue(port, ct_cmd);
1485                                 ct_cmd = fct_create_solct(port, rp,
1486                                     NS_GCS_ID, fct_gcs_cb);
1487                                 if (ct_cmd)
1488                                         fct_post_to_solcmd_queue(port, ct_cmd);
1489                                 ct_cmd = fct_create_solct(port, rp,
1490                                     NS_GFT_ID, fct_gft_cb);
1491                                 if (ct_cmd)
1492                                         fct_post_to_solcmd_queue(port, ct_cmd);
1493                         }
1494                 } else {
1495                         /*
1496                          * The reason we set this flag is to prevent
1497                          * killing a PRLI while we have not yet processed
1498                          * a response to PLOGI. Because the initiator
1499                          * will send a PRLI as soon as it responds to PLOGI.
1500                          * Check fct_process_els() for more info.
1501                          */
1502                         atomic_or_32(&irp->irp_flags,
1503                             IRP_SOL_PLOGI_IN_PROGRESS);
1504                         atomic_or_32(&icmd->icmd_flags, ICMD_KNOWN_TO_FCA);
1505                         ret = port->port_send_cmd(cmd);
1506                         if (ret != FCT_SUCCESS) {
1507                                 atomic_and_32(&icmd->icmd_flags,
1508                                     ~ICMD_KNOWN_TO_FCA);
1509                                 atomic_and_32(&irp->irp_flags,
1510                                     ~IRP_SOL_PLOGI_IN_PROGRESS);
1511                         }
1512                 }
1513         }
1514         atomic_dec_16(&irp->irp_sa_elses_count);
1514         atomic_add_16(&irp->irp_sa_elses_count, -1);

1516         if (ret == FCT_SUCCESS) {
1517                 if (cmd_type == FCT_CMD_RCVD_ELS) {
1518                         atomic_or_32(&irp->irp_flags, IRP_PLOGI_DONE);
1519                         atomic_inc_32(&iport->iport_nrps_login);
1519                         atomic_add_32(&iport->iport_nrps_login, 1);
1520                         if (irp->irp_deregister_timer)
1521                                 irp->irp_deregister_timer = 0;
1522                 }
1523                 if (icmd_flags & ICMD_IMPLICIT) {
1524                         DTRACE_FC_5(rport__login__end,
1525                             fct_cmd_t, cmd,
1526                             fct_local_port_t, port,
1527                             fct_i_remote_port_t, irp,
1528                             int, (cmd_type != FCT_CMD_RCVD_ELS),
1529                             int, FCT_SUCCESS);

1531                         p = els->els_resp_payload;
1532                         p[0] = ELS_OP_ACC;
1533                         cmd->cmd_comp_status = FCT_SUCCESS;
1534                         fct_send_cmd_done(cmd, FCT_SUCCESS, FCT_IOF_FCA_DONE);
1535                 }
1536         } else {
1537                 DTRACE_FC_5(rport__login__end,
1538                     fct_cmd_t, cmd,
1539                     fct_local_port_t, port,
1540                     fct_i_remote_port_t, irp,
1541                     int, (cmd_type != FCT_CMD_RCVD_ELS),
1542                     int, ret);

1544                 fct_queue_cmd_for_termination(cmd, ret);
1545         }

1547         /* Do not touch cmd here as it may have been freed */
```

```
1549         return (DISC_ACTION_RESCAN);
1550 }

1552 uint8_t fct_prli_temp[] = { 0x20, 0x10, 0, 0x14, 8, 0, 0x20, 0, 0, 0, 0, 0,
1553                                 0, 0, 0, 0 };

1555 disc_action_t
1556 fct_process_prli(fct_i_cmd_t *icmd)
1557 {
1558         fct_cmd_t               *cmd   = icmd->icmd_cmd;
1559         fct_remote_port_t       *rp    = cmd->cmd_rp;
1560         fct_local_port_t        *port  = cmd->cmd_port;
1561         fct_i_local_port_t      *iport = (fct_i_local_port_t *)
1562             port->port_fct_private;
1563         fct_els_t               *els   = (fct_els_t *)
1564             cmd->cmd_specific;
1565         fct_i_remote_port_t     *irp   = (fct_i_remote_port_t *)
1566             rp->rp_fct_private;
1567         stmf_scsi_session_t     *ses   = NULL;
1568         fct_status_t             ret;
1569         clock_t                  end_time;
1570         char                     info[FCT_INFO_LEN];

1572         /* We dont support solicited PRLIs yet */
1573         ASSERT(cmd->cmd_type == FCT_CMD_RCVD_ELS);

1575         if (irp->irp_flags & IRP_SOL_PLOGI_IN_PROGRESS) {
1576                 /*
1577                  * Dont process the PRLI yet. Let the framework process the
1578                  * PLOGI completion 1st. This should be very quick because
1579                  * the reason we got the PRLI is because the initiator
1580                  * has responded to PLOGI already.
1581                  */
1582                 /* XXX: Probably need a timeout here */
1583                 return (DISC_ACTION_DELAY_RESCAN);
1584         }
1585         /* The caller has made sure that login is done */

1587         /* Make sure the process is fcp in this case */
1588         if ((els->els_req_size != 20) || (bcmp(els->els_req_payload,
1589             fct_prli_temp, 16))) {
1590                 if (els->els_req_payload[4] != 0x08)
1591                         stmf_trace(iport->iport_alias, "PRLI received from"
1592                             " %x for unknown FC-4 type %x", cmd->cmd_rportid,
1593                             els->els_req_payload[4]);
1594                 else
1595                         stmf_trace(iport->iport_alias, "Rejecting PRLI from %x "
1596                             " pld sz %d, prli_flags %x", cmd->cmd_rportid,
1597                             els->els_req_size, els->els_req_payload[6]);

1599                 fct_dequeue_els(irp);
1600                 atomic_dec_16(&irp->irp_sa_elses_count);
1600                 atomic_add_16(&irp->irp_sa_elses_count, -1);
1601                 ret = fct_send_accrjt(cmd, ELS_OP_LSRJT, 3, 0x2c);
1602                 goto prli_end;
1603         }

1605         if (irp->irp_fcp_xchg_count) {
1606                 /* Trigger cleanup if necessary */
1607                 if ((irp->irp_flags & IRP_FCP_CLEANUP) == 0) {
1608                         stmf_trace(iport->iport_alias, "handling PRLI from"
1609                             " %x. Triggering cleanup", cmd->cmd_rportid);
1610                         if (fct_trigger_rport_cleanup(irp, FCT_CMD_FCP_XCHG)) {
1611                                 atomic_or_32(&irp->irp_flags, IRP_FCP_CLEANUP);
1612                         } else {
```



```
1613                                 /* XXX: handle this */
1614                                 /* EMPTY */
1615                         }
1616                 }

1618                 end_time = icmd->icmd_start_time +
1619                     drv_usectohz(USEC_ELS_TIMEOUT);
1620                 if (ddi_get_lbolt() > end_time) {
1621                         (void) snprintf(info, sizeof (info),
1622                             "fct_process_prli: unable to clean "
1623                             "up I/O. iport-%p, icmd-%p", (void *)iport,
1624                             (void *)icmd);
1625                         (void) fct_port_shutdown(iport->iport_port,
1626                             STMF_RFLAG_FATAL_ERROR | STMF_RFLAG_RESET, info);

1628                         return (DISC_ACTION_DELAY_RESCAN);
1629                 }

1631                 if ((ddi_get_lbolt() & 0x7f) == 0) {
1632                         stmf_trace(iport->iport_alias, "handling"
1633                             " PRLI from %x, waiting for cmds to"
1634                             " drain", cmd->cmd_rportid);
1635                 }
1636                 return (DISC_ACTION_DELAY_RESCAN);
1637         }
1638         atomic_and_32(&irp->irp_flags, ~IRP_FCP_CLEANUP);

1640         /* Session can only be terminated after all the I/Os have drained */
1641         if (irp->irp_flags & IRP_SCSI_SESSION_STARTED) {
1642                 stmf_deregister_scsi_session(iport->iport_port->port_lport,
1643                     irp->irp_session);
1644                 stmf_free(irp->irp_session);
1645                 irp->irp_session = NULL;
1646                 atomic_and_32(&irp->irp_flags, ~IRP_SCSI_SESSION_STARTED);
1647         }

1649         /* All good, lets start a session */
1650         ses = (stmf_scsi_session_t *)stmf_alloc(STMF_STRUCT_SCSI_SESSION, 0, 0);
1651         if (ses) {
1652                 ses->ss_port_private = irp;
1653                 ses->ss_rport_id = (scsi_devid_desc_t *)irp->irp_id;
1654                 ses->ss_lport = port->port_lport;
1655                 if (stmf_register_scsi_session(port->port_lport, ses) !=
1656                     STMF_SUCCESS) {
1657                         stmf_free(ses);
1658                         ses = NULL;
1659                 } else {
1660                         irp->irp_session = ses;
1661                         irp->irp_session->ss_rport_alias = irp->irp_snn;

1663                         /*
1664                          * The reason IRP_SCSI_SESSION_STARTED is different
1665                          * from IRP_PRLI_DONE is that we clear IRP_PRLI_DONE
1666                          * inside interrupt context. We dont want to deregister
1667                          * the session from an interrupt.
1668                          */
1669                         atomic_or_32(&irp->irp_flags, IRP_SCSI_SESSION_STARTED);
1670                 }
1671         }

1673         fct_dequeue_els(irp);
1674         atomic_dec_16(&irp->irp_sa_elses_count);
1674         atomic_add_16(&irp->irp_sa_elses_count, -1);
1675         if (ses == NULL) {
1676                 /* fail PRLI */
1677                 ret = fct_send_accrjt(cmd, ELS_OP_LSRJT, 3, 0);
```

```
1678         } else {
1679                 /* accept PRLI */
1680                 els->els_resp_payload = (uint8_t *)kmem_zalloc(20, KM_SLEEP);
1681                 bcopy(fct_prli_temp, els->els_resp_payload, 20);
1682                 els->els_resp_payload[0] = 2;
1683                 els->els_resp_payload[6] = 0x21;

1685                 /* XXX the two bytes below need to set as per capabilities */
1686                 els->els_resp_payload[18] = 0;
1687                 els->els_resp_payload[19] = 0x12;

1689                 els->els_resp_size = els->els_resp_alloc_size = 20;
1690                 if ((ret = port->port_send_cmd_response(cmd, 0)) !=
1691                     FCT_SUCCESS) {
1692                         stmf_deregister_scsi_session(port->port_lport, ses);
1693                         stmf_free(irp->irp_session);
1694                         irp->irp_session = NULL;
1695                         atomic_and_32(&irp->irp_flags,
1696                             ~IRP_SCSI_SESSION_STARTED);
1697                 } else {
1698                         /* Mark that PRLI is done */
1699                         atomic_or_32(&irp->irp_flags, IRP_PRLI_DONE);
1700                 }
1701         }

1703 prli_end:;
1704         if (ret != FCT_SUCCESS)
1705                 fct_queue_cmd_for_termination(cmd, ret);

1707         return (DISC_ACTION_RESCAN);
1708 }

1710 disc_action_t
1711 fct_process_logo(fct_i_cmd_t *icmd)
1712 {
1713         fct_cmd_t               *cmd   = icmd->icmd_cmd;
1714         fct_remote_port_t       *rp    = cmd->cmd_rp;
1715         fct_local_port_t        *port  = cmd->cmd_port;
1716         fct_i_local_port_t      *iport = (fct_i_local_port_t *)
1717             port->port_fct_private;
1718         fct_i_remote_port_t     *irp   = (fct_i_remote_port_t *)
1719             rp->rp_fct_private;
1720         fct_status_t             ret;
1721         char                     info[FCT_INFO_LEN];
1722         clock_t                  end_time;

1724         DTRACE_FC_4(rport__logout__start,
1725             fct_cmd_t, cmd,
1726             fct_local_port_t, port,
1727             fct_i_remote_port_t, irp,
1728             int, (cmd->cmd_type != FCT_CMD_RCVD_ELS));

1730         /* Drain I/Os */
1731         if ((irp->irp_nonfcp_xchg_count + irp->irp_fcp_xchg_count) > 1) {
1732                 /* Trigger cleanup if necessary */
1733                 if ((irp->irp_flags & IRP_SESSION_CLEANUP) == 0) {
1734                         stmf_trace(iport->iport_alias, "handling LOGO rp_id"
1735                             " %x. Triggering cleanup", cmd->cmd_rportid);
1736                         /* Cleanup everything except elses */
1737                         if (fct_trigger_rport_cleanup(irp, ~(cmd->cmd_type))) {
1738                                 atomic_or_32(&irp->irp_flags,
1739                                     IRP_SESSION_CLEANUP);
1740                         } else {
1741                                 /* XXX: need more handling */
1742                                 return (DISC_ACTION_DELAY_RESCAN);
1743                         }
```



```
1744                 }

1746                 end_time = icmd->icmd_start_time +
1747                     drv_usectohz(USEC_ELS_TIMEOUT);
1748                 if (ddi_get_lbolt() > end_time) {
1749                         (void) snprintf(info, sizeof (info),
1750                             "fct_process_logo: unable to clean "
1751                             "up I/O. iport-%p, icmd-%p", (void *)iport,
1752                             (void *)icmd);
1753                         (void) fct_port_shutdown(iport->iport_port,
1754                             STMF_RFLAG_FATAL_ERROR | STMF_RFLAG_RESET, info);

1756                         return (DISC_ACTION_DELAY_RESCAN);
1757                 }

1759                 if ((ddi_get_lbolt() & 0x7f) == 0) {
1760                         stmf_trace(iport->iport_alias, "handling"
1761                             " LOGO rp_id %x, waiting for cmds to"
1762                             " drain", cmd->cmd_rportid);
1763                 }
1764                 return (DISC_ACTION_DELAY_RESCAN);
1765         }
1766         atomic_and_32(&irp->irp_flags, ~IRP_SESSION_CLEANUP);

1768         /* Session can only be terminated after all the I/Os have drained */
1769         if (irp->irp_flags & IRP_SCSI_SESSION_STARTED) {
1770                 stmf_deregister_scsi_session(iport->iport_port->port_lport,
1771                     irp->irp_session);
1772                 stmf_free(irp->irp_session);
1773                 irp->irp_session = NULL;
1774                 atomic_and_32(&irp->irp_flags, ~IRP_SCSI_SESSION_STARTED);
1775         }

1777         fct_dequeue_els(irp);
1778         atomic_dec_16(&irp->irp_sa_elses_count);
1778         atomic_add_16(&irp->irp_sa_elses_count, -1);

1780         /* don't send response if this is an implicit logout cmd */
1781         if (!(icmd->icmd_flags & ICMD_IMPLICIT)) {
1782                 if (cmd->cmd_type == FCT_CMD_RCVD_ELS) {
1783                         ret = fct_send_accrjt(cmd, ELS_OP_ACC, 0, 0);
1784                 } else {
1785                         atomic_or_32(&icmd->icmd_flags, ICMD_KNOWN_TO_FCA);
1786                         ret = port->port_send_cmd(cmd);
1787                         if (ret != FCT_SUCCESS) {
1788                                 atomic_and_32(&icmd->icmd_flags,
1789                                     ~ICMD_KNOWN_TO_FCA);
1790                         }
1791                 }

1793                 if (ret != FCT_SUCCESS) {
1794                         fct_queue_cmd_for_termination(cmd, ret);
1795                 }

1797                 DTRACE_FC_4(rport__logout__end,
1798                     fct_cmd_t, cmd,
1799                     fct_local_port_t, port,
1800                     fct_i_remote_port_t, irp,
1801                     int, (cmd->cmd_type != FCT_CMD_RCVD_ELS));

1803         } else {
1804                 DTRACE_FC_4(rport__logout__end,
1805                     fct_cmd_t, cmd,
1806                     fct_local_port_t, port,
1807                     fct_i_remote_port_t, irp,
1808                     int, (cmd->cmd_type != FCT_CMD_RCVD_ELS));
```

```
1810                 fct_cmd_free(cmd);
1811         }

1813         irp->irp_deregister_timer = ddi_get_lbolt() +
1814             drv_usectohz(USEC_DEREG_RP_TIMEOUT);
1815         irp->irp_dereg_count = 0;

1817         /* Do not touch cmd here as it may have been freed */

1819         ASSERT(irp->irp_flags & IRP_IN_DISCOVERY_QUEUE);

1821         return (DISC_ACTION_RESCAN);
1822 }

1824 disc_action_t
1825 fct_process_prlo(fct_i_cmd_t *icmd)
1826 {
1827         fct_cmd_t               *cmd   = icmd->icmd_cmd;
1828         fct_remote_port_t       *rp    = cmd->cmd_rp;
1829         fct_local_port_t        *port  = cmd->cmd_port;
1830         fct_i_local_port_t      *iport = (fct_i_local_port_t *)
1831             port->port_fct_private;
1832         fct_i_remote_port_t     *irp   = (fct_i_remote_port_t *)
1833             rp->rp_fct_private;
1834         fct_status_t             ret;
1835         clock_t                  end_time;
1836         char                     info[FCT_INFO_LEN];

1838         /* We do not support solicited PRLOs yet */
1839         ASSERT(cmd->cmd_type == FCT_CMD_RCVD_ELS);

1841         /* Drain I/Os */
1842         if (irp->irp_fcp_xchg_count) {
1843                 /* Trigger cleanup if necessary */
1844                 if ((irp->irp_flags & IRP_FCP_CLEANUP) == 0) {
1845                         stmf_trace(iport->iport_alias, "handling LOGO from"
1846                             " %x. Triggering cleanup", cmd->cmd_rportid);
1847                         /* Cleanup everything except elses */
1848                         if (fct_trigger_rport_cleanup(irp, FCT_CMD_FCP_XCHG)) {
1849                                 atomic_or_32(&irp->irp_flags,
1850                                     IRP_FCP_CLEANUP);
1851                         } else {
1852                                 /* XXX: need more handling */
1853                                 return (DISC_ACTION_DELAY_RESCAN);
1854                         }
1855                 }

1857                 end_time = icmd->icmd_start_time +
1858                     drv_usectohz(USEC_ELS_TIMEOUT);
1859                 if (ddi_get_lbolt() > end_time) {
1860                         (void) snprintf(info, sizeof (info),
1861                             "fct_process_prlo: unable to "
1862                             "clean up I/O. iport-%p, icmd-%p", (void *)iport,
1863                             (void *)icmd);
1864                         (void) fct_port_shutdown(iport->iport_port,
1865                             STMF_RFLAG_FATAL_ERROR | STMF_RFLAG_RESET, info);

1867                         return (DISC_ACTION_DELAY_RESCAN);
1868                 }

1870                 if ((ddi_get_lbolt() & 0x7f) == 0) {
1871                         stmf_trace(iport->iport_alias, "handling"
1872                             " PRLO from %x, waiting for cmds to"
1873                             " drain", cmd->cmd_rportid);
1874                 }
```



```
1875                         return (DISC_ACTION_DELAY_RESCAN);
1876         }
1877         atomic_and_32(&irp->irp_flags, ~IRP_FCP_CLEANUP);

1879         /* Session can only be terminated after all the I/Os have drained */
1880         if (irp->irp_flags & IRP_SCSI_SESSION_STARTED) {
1881                 stmf_deregister_scsi_session(iport->iport_port->port_lport,
1882                     irp->irp_session);
1883                 stmf_free(irp->irp_session);
1884                 irp->irp_session = NULL;
1885                 atomic_and_32(&irp->irp_flags, ~IRP_SCSI_SESSION_STARTED);
1886         }

1888         fct_dequeue_els(irp);
1889         atomic_dec_16(&irp->irp_sa_elses_count);
1889         atomic_add_16(&irp->irp_sa_elses_count, -1);
1890         ret = fct_send_accrjt(cmd, ELS_OP_ACC, 0, 0);
1891         if (ret != FCT_SUCCESS)
1892                 fct_queue_cmd_for_termination(cmd, ret);

1894         return (DISC_ACTION_RESCAN);
1895 }

1897 disc_action_t
1898 fct_process_rcvd_adisc(fct_i_cmd_t *icmd)
1899 {
1900         fct_cmd_t               *cmd = icmd->icmd_cmd;
1901         fct_remote_port_t       *rp = cmd->cmd_rp;
1902         fct_local_port_t        *port = cmd->cmd_port;
1903         fct_i_local_port_t      *iport = (fct_i_local_port_t *)
1904             port->port_fct_private;
1905         fct_els_t               *els = (fct_els_t *)
1906             cmd->cmd_specific;
1907         fct_i_remote_port_t     *irp = (fct_i_remote_port_t *)
1908             rp->rp_fct_private;
1909         uint8_t                 *p;
1910         uint32_t                *q;
1911         fct_status_t            ret;

1913         fct_dequeue_els(irp);
1914         atomic_dec_16(&irp->irp_nsa_elses_count);
1914         atomic_add_16(&irp->irp_nsa_elses_count, -1);

1916         /* Validate the adisc request */
1917         p = els->els_req_payload;
1918         q = (uint32_t *)p;
1919         if ((els->els_req_size != 28) || (bcmp(rp->rp_pwwn, p + 8, 8)) ||
1920             (bcmp(rp->rp_nwwn, p + 16, 8))) {
1921                 ret = fct_send_accrjt(cmd, ELS_OP_LSRJT, 3, 0);
1922         } else {
1923                 rp->rp_hard_address = BE_32(q[1]);
1924                 els->els_resp_size = els->els_resp_alloc_size = 28;
1925                 els->els_resp_payload = (uint8_t *)kmem_zalloc(28, KM_SLEEP);
1926                 bcopy(p, els->els_resp_payload, 28);
1927                 p = els->els_resp_payload;
1928                 q = (uint32_t *)p;
1929                 p[0] = ELS_OP_ACC;
1930                 q[1] = BE_32(port->port_hard_address);
1931                 bcopy(port->port_pwwn, p + 8, 8);
1932                 bcopy(port->port_nwwn, p + 16, 8);
1933                 q[6] = BE_32(iport->iport_link_info.portid);
1934                 ret = port->port_send_cmd_response(cmd, 0);
1935         }
1936         if (ret != FCT_SUCCESS) {
1937                 fct_queue_cmd_for_termination(cmd, ret);
1938         }
```

```
1940         return (DISC_ACTION_RESCAN);
1941 }

1943 disc_action_t
1944 fct_process_unknown_els(fct_i_cmd_t *icmd)
1945 {
1946         fct_i_local_port_t      *iport = ICMD_TO_IPORT(icmd);
1947         fct_status_t             ret   = FCT_FAILURE;
1948         uint8_t                  op    = 0;

1950         ASSERT(icmd->icmd_cmd->cmd_type == FCT_CMD_RCVD_ELS);
1951         fct_dequeue_els(ICMD_TO_IRP(icmd));
1952         atomic_dec_16(&ICMD_TO_IRP(icmd)->irp_nsa_elses_count);
1952         atomic_add_16(&ICMD_TO_IRP(icmd)->irp_nsa_elses_count, -1);
1953         op = ICMD_TO_ELS(icmd)->els_req_payload[0];
1954         stmf_trace(iport->iport_alias, "Rejecting unknown unsol els %x (%s)",
1955             op, FCT_ELS_NAME(op));
1956         ret = fct_send_accrjt(icmd->icmd_cmd, ELS_OP_LSRJT, 1, 0);
1957         if (ret != FCT_SUCCESS) {
1958                 fct_queue_cmd_for_termination(icmd->icmd_cmd, ret);
1959         }

1961         return (DISC_ACTION_RESCAN);
1962 }

1964 disc_action_t
1965 fct_process_rscn(fct_i_cmd_t *icmd)
1966 {
1967         fct_i_local_port_t      *iport = ICMD_TO_IPORT(icmd);
1968         fct_status_t             ret   = FCT_FAILURE;
1969         uint8_t                  op    = 0;
1970         uint8_t                 *rscn_req_payload;
1971         uint32_t                 rscn_req_size;

1973         fct_dequeue_els(ICMD_TO_IRP(icmd));
1974         atomic_dec_16(&ICMD_TO_IRP(icmd)->irp_nsa_elses_count);
1974         atomic_add_16(&ICMD_TO_IRP(icmd)->irp_nsa_elses_count, -1);
1975         if (icmd->icmd_cmd->cmd_type == FCT_CMD_RCVD_ELS) {
1976                 op = ICMD_TO_ELS(icmd)->els_req_payload[0];
1977                 stmf_trace(iport->iport_alias, "Accepting RSCN %x (%s)",
1978                     op, FCT_ELS_NAME(op));
1979                 rscn_req_size = ICMD_TO_ELS(icmd)->els_req_size;
1980                 rscn_req_payload = kmem_alloc(rscn_req_size, KM_SLEEP);
1981                 bcopy(ICMD_TO_ELS(icmd)->els_req_payload, rscn_req_payload,
1982                     rscn_req_size);
1983                 ret = fct_send_accrjt(icmd->icmd_cmd, ELS_OP_ACC, 1, 0);
1984                 if (ret != FCT_SUCCESS) {
1985                         fct_queue_cmd_for_termination(icmd->icmd_cmd, ret);
1986                 } else {
1987                         if (fct_rscn_options & RSCN_OPTION_VERIFY) {
1988                                 fct_rscn_verify(iport, rscn_req_payload,
1989                                     rscn_req_size);
1990                         }
1991                 }

1993                 kmem_free(rscn_req_payload, rscn_req_size);
1994         } else {
1995                 ASSERT(0);
1996         }

1998         return (DISC_ACTION_RESCAN);
1999 }

2001 disc_action_t
2002 fct_process_els(fct_i_local_port_t *iport, fct_i_remote_port_t *irp)
```



```
2003 {
2004         fct_i_cmd_t     *cmd_to_abort = NULL;
2005         fct_i_cmd_t     **ppcmd, *icmd;
2006         fct_cmd_t       *cmd;
2007         fct_els_t       *els;
2008         int             dq;
2009         disc_action_t   ret = DISC_ACTION_NO_WORK;
2010         uint8_t         op;

2012         mutex_exit(&iport->iport_worker_lock);

2014         /*
2015          * Do some cleanup based on the following.
2016          * - We can only have one session affecting els pending.
2017          * - If any session affecting els is pending no other els is allowed.
2018          * - If PLOGI is not done, nothing except PLOGI or LOGO is allowed.
2019          * NOTE: If port is down the cleanup is done outside of this
2020          *       function.
2021          * NOTE: There is a side effect, if a sa ELS (non PLOGI) is received
2022          * while a PLOGI is pending, it will kill itself and the PLOGI.
2023          * which is probably ok.
2024          */
2025         rw_enter(&irp->irp_lock, RW_WRITER);
2026         ppcmd = &irp->irp_els_list;
2027         while ((*ppcmd) != NULL) {
2028                 int special_prli_cond = 0;
2029                 dq = 0;

2031                 els = (fct_els_t *)((*ppcmd)->icmd_cmd)->cmd_specific;

2033                 if (((*ppcmd)->icmd_cmd->cmd_type == FCT_CMD_RCVD_ELS) &&
2034                     (els->els_req_payload[0] == ELS_OP_PRLI) &&
2035                     (irp->irp_flags & IRP_SOL_PLOGI_IN_PROGRESS)) {
2036                         /*
2037                          * The initiator sent a PRLI right after responding
2038                          * to PLOGI and we have not yet finished processing
2039                          * the PLOGI completion. We should not kill the PRLI
2040                          * as the initiator may not retry it.
2041                          */
2042                         special_prli_cond = 1;
2043                 }

2045                 if ((*ppcmd)->icmd_flags & ICMD_BEING_ABORTED) {
2046                         dq = 1;
2047                 } else if (irp->irp_sa_elses_count > 1) {
2048                         dq = 1;
2049                         /* This els might have set the CLEANUP flag */
2050                         atomic_and_32(&irp->irp_flags, ~IRP_SESSION_CLEANUP);
2051                         stmf_trace(iport->iport_alias, "Killing ELS %x cond 1",
2052                             els->els_req_payload[0]);
2053                 } else if (irp->irp_sa_elses_count &&
2054                     (((*ppcmd)->icmd_flags & ICMD_SESSION_AFFECTING) == 0)) {
2055                         stmf_trace(iport->iport_alias, "Killing ELS %x cond 2",
2056                             els->els_req_payload[0]);
2057                         dq = 1;
2058                 } else if (((irp->irp_flags & IRP_PLOGI_DONE) == 0) &&
2059                     (els->els_req_payload[0] != ELS_OP_PLOGI) &&
2060                     (els->els_req_payload[0] != ELS_OP_LOGO) &&
2061                     (special_prli_cond == 0)) {
2062                         stmf_trace(iport->iport_alias, "Killing ELS %x cond 3",
2063                             els->els_req_payload[0]);
2064                         dq = 1;
2065                 }

2067                 if (dq) {
2068                         fct_i_cmd_t *c = (*ppcmd)->icmd_next;
```

```
2070                        if ((*ppcmd)->icmd_flags & ICMD_SESSION_AFFECTING)
2071                                atomic_dec_16(&irp->irp_sa_elses_count);
2071                                atomic_add_16(&irp->irp_sa_elses_count, -1);
2072                        else
2073                                atomic_dec_16(&irp->irp_nsa_elses_count);
2073                                atomic_add_16(&irp->irp_nsa_elses_count, -1);
2074                        (*ppcmd)->icmd_next = cmd_to_abort;
2075                        cmd_to_abort = *ppcmd;
2076                        *ppcmd = c;
2077                } else {
2078                        ppcmd = &((*ppcmd)->icmd_next);
2079                }
2080        }
2081        rw_exit(&irp->irp_lock);

2083        while (cmd_to_abort) {
2084                fct_i_cmd_t *c = cmd_to_abort->icmd_next;

2086                atomic_and_32(&cmd_to_abort->icmd_flags, ~ICMD_IN_IRP_QUEUE);
2087                fct_queue_cmd_for_termination(cmd_to_abort->icmd_cmd,
2088                    FCT_ABORTED);
2089                cmd_to_abort = c;
2090        }

2092        /*
2093         * pick from the top of the queue
2094         */
2095        icmd = irp->irp_els_list;
2096        if (icmd == NULL) {
2097                /*
2098                 * The cleanup took care of everything.
2099                 */

2101                mutex_enter(&iport->iport_worker_lock);
2102                return (DISC_ACTION_RESCAN);
2103        }

2105        cmd = icmd->icmd_cmd;
2106        els = ICMD_TO_ELS(icmd);
2107        op = els->els_req_payload[0];
2108        if ((icmd->icmd_flags & ICMD_ELS_PROCESSING_STARTED) == 0) {
2109                stmf_trace(iport->iport_alias, "Processing %ssol ELS %x (%s) "
2110                    "rp_id=%x", (cmd->cmd_type == FCT_CMD_RCVD_ELS) ? "un" : "",
2111                    op, FCT_ELS_NAME(op), cmd->cmd_rportid);
2112                atomic_or_32(&icmd->icmd_flags, ICMD_ELS_PROCESSING_STARTED);
2113        }

2115        if (op == ELS_OP_PLOGI) {
2116                ret |= fct_process_plogi(icmd);
2117        } else if (op == ELS_OP_PRLI) {
2118                ret |= fct_process_prli(icmd);
2119        } else if (op == ELS_OP_LOGO) {
2120                ret |= fct_process_logo(icmd);
2121        } else if ((op == ELS_OP_PRLO) || (op == ELS_OP_TPRLO)) {
2122                ret |= fct_process_prlo(icmd);
2123        } else if (cmd->cmd_type == FCT_CMD_SOL_ELS) {
2124                fct_status_t s;
2125                fct_local_port_t *port = iport->iport_port;

2127                fct_dequeue_els(irp);
2128                atomic_dec_16(&irp->irp_nsa_elses_count);
2128                atomic_add_16(&irp->irp_nsa_elses_count, -1);
2129                atomic_or_32(&icmd->icmd_flags, ICMD_KNOWN_TO_FCA);
2130                if ((s = port->port_send_cmd(cmd)) != FCT_SUCCESS) {
2131                        atomic_and_32(&icmd->icmd_flags, ~ICMD_KNOWN_TO_FCA);
```



```
2132                        fct_queue_cmd_for_termination(cmd, s);
2133                        stmf_trace(iport->iport_alias, "Solicited els "
2134                            "transport failed, ret = %llx", s);
2135                }
2136        } else if (op == ELS_OP_ADISC) {
2137                ret |= fct_process_rcvd_adisc(icmd);
2138        } else if (op == ELS_OP_RSCN) {
2139                (void) fct_process_rscn(icmd);
2140        } else {
2141                (void) fct_process_unknown_els(icmd);
2142        }

2144        /*
2145         * This if condition will be false if a sa ELS trigged a cleanup
2146         * and set the ret = DISC_ACTION_DELAY_RESCAN. In that case we should
2147         * keep it that way.
2148         */
2149        if (ret == DISC_ACTION_NO_WORK) {
2150                /*
2151                 * Since we dropped the lock, we will force a rescan. The
2152                 * only exception is if someone returned
2153                 * DISC_ACTION_DELAY_RESCAN, in which case that should be the
2154                 * return value.
2155                 */
2156                ret = DISC_ACTION_RESCAN;
2157        }

2159        mutex_enter(&iport->iport_worker_lock);
2160        return (ret);
2161 }

2163 void
2164 fct_handle_sol_els_completion(fct_i_local_port_t *iport, fct_i_cmd_t *icmd)
2165 {
2166        fct_i_remote_port_t     *irp = NULL;
2167        fct_els_t               *els = ICMD_TO_ELS(icmd);
2168        uint8_t                 op  = els->els_req_payload[0];

2170        if (icmd->icmd_cmd->cmd_rp) {
2171                irp = ICMD_TO_IRP(icmd);
2172        }
2173        if (icmd->icmd_cmd->cmd_rp &&
2174            (icmd->icmd_cmd->cmd_comp_status == FCT_SUCCESS) &&
2175            (els->els_req_payload[0] == ELS_OP_PLOGI)) {
2176                bcopy(els->els_resp_payload + 20, irp->irp_rp->rp_pwwn, 8);
2177                bcopy(els->els_resp_payload + 28, irp->irp_rp->rp_nwwn, 8);

2179                stmf_wwn_to_devid_desc((scsi_devid_desc_t *)irp->irp_id,
2180                    irp->irp_rp->rp_pwwn, PROTOCOL_FIBRE_CHANNEL);
2181                atomic_or_32(&irp->irp_flags, IRP_PLOGI_DONE);
2182                atomic_inc_32(&iport->iport_nrps_login);
2182                atomic_add_32(&iport->iport_nrps_login, 1);
2183                if (irp->irp_deregister_timer) {
2184                        irp->irp_deregister_timer = 0;
2185                        irp->irp_dereg_count = 0;
2186                }
2187        }

2189        if (irp && (els->els_req_payload[0] == ELS_OP_PLOGI)) {
2190                atomic_and_32(&irp->irp_flags, ~IRP_SOL_PLOGI_IN_PROGRESS);
2191        }
2192        atomic_or_32(&icmd->icmd_flags, ICMD_CMD_COMPLETE);
2193        stmf_trace(iport->iport_alias, "Sol ELS %x (%s) completed with "
2194            "status %llx, did/%x", op, FCT_ELS_NAME(op),
2195            icmd->icmd_cmd->cmd_comp_status, icmd->icmd_cmd->cmd_rportid);
2196 }
```

```
2198 static disc_action_t
2199 fct_check_cmdlist(fct_i_local_port_t *iport)
2200 {
2201         int             num_to_release, ndx;
2202         fct_i_cmd_t     *icmd;
2203         uint32_t        total, max_active;

2205         ASSERT(MUTEX_HELD(&iport->iport_worker_lock));

2207         total = iport->iport_total_alloced_ncmds;
2208         max_active = iport->iport_max_active_ncmds;

2210         if (total <= max_active)
2211                 return (DISC_ACTION_NO_WORK);
2212         /*
2213          * Everytime, we release half of the difference
2214          */
2215         num_to_release = (total + 1 - max_active) / 2;

2217         mutex_exit(&iport->iport_worker_lock);
2218         for (ndx = 0; ndx < num_to_release; ndx++) {
2219                 mutex_enter(&iport->iport_cached_cmd_lock);
2220                 icmd = iport->iport_cached_cmdlist;
2221                 if (icmd == NULL) {
2222                         mutex_exit(&iport->iport_cached_cmd_lock);
2223                         break;
2224                 }
2225                 iport->iport_cached_cmdlist = icmd->icmd_next;
2226                 iport->iport_cached_ncmds--;
2227                 mutex_exit(&iport->iport_cached_cmd_lock);
2228                 atomic_dec_32(&iport->iport_total_alloced_ncmds);
2228                 atomic_add_32(&iport->iport_total_alloced_ncmds, -1);
2229                 fct_free(icmd->icmd_cmd);
2230         }
2231         mutex_enter(&iport->iport_worker_lock);
2232         return (DISC_ACTION_RESCAN);
2233 }
```

*unchanged_portion_omitted_*

```
2314 void
2315 fct_handle_solct(fct_cmd_t *cmd)
2316 {
2317         fct_status_t            ret      = FCT_SUCCESS;
2318         fct_i_cmd_t             *icmd    = CMD_TO_ICMD(cmd);
2319         fct_i_local_port_t      *iport   = ICMD_TO_IPORT(icmd);
2320         fct_i_remote_port_t     *irp     = ICMD_TO_IRP(icmd);

2322         ASSERT(cmd->cmd_type == FCT_CMD_SOL_CT);
2323         rw_enter(&iport->iport_lock, RW_READER);
2324         /*
2325          * Let's make sure local port is sane
2326          */
2327         if ((iport->iport_link_state & S_LINK_ONLINE) == 0) {
2328                 rw_exit(&iport->iport_lock);

2330                 stmf_trace(iport->iport_alias, "fct_transport_solct: "
2331                     "solcmd-%p transport failed, becasue port state was %x",
2332                     cmd, iport->iport_link_state);
2333                 fct_queue_cmd_for_termination(cmd, FCT_LOCAL_PORT_OFFLINE);
2334                 return;
2335         }

2337         /*
2338          * Let's make sure we have plogi-ed to name server
2339          */
```



```
2340         rw_enter(&irp->irp_lock, RW_READER);
2341         if (!(irp->irp_flags & IRP_PLOGI_DONE)) {
2342                 rw_exit(&irp->irp_lock);
2343                 rw_exit(&iport->iport_lock);

2345                 stmf_trace(iport->iport_alias, "fct_transport_solct: "
2346                     "Must login to name server first - cmd-%p", cmd);
2347                 fct_queue_cmd_for_termination(cmd, FCT_NOT_LOGGED_IN);
2348                 return;
2349         }

2351         /*
2352          * Let's get a slot for this solcmd
2353          */
2354         if (fct_alloc_cmd_slot(iport, cmd) == FCT_SLOT_EOL) {
2355                 rw_exit(&irp->irp_lock);
2356                 rw_exit(&iport->iport_lock);

2358                 stmf_trace(iport->iport_alias, "fct_transport_solcmd: "
2359                     "ran out of xchg resources - cmd-%p", cmd);
2360                 fct_queue_cmd_for_termination(cmd, FCT_NO_XCHG_RESOURCE);
2361                 return;
2362         }

2364         if (fct_netbuf_to_value(ICMD_TO_CT(icmd)->ct_req_payload + 8, 2) ==
2365             NS_GID_PN) {
2366                 fct_i_remote_port_t     *query_irp = NULL;

2368                 query_irp = fct_lookup_irp_by_portwwn(iport,
2369                     ICMD_TO_CT(icmd)->ct_req_payload + 16);
2370                 if (query_irp) {
2371                         atomic_and_32(&query_irp->irp_flags, ~IRP_RSCN_QUEUED);
2372                 }
2373         }
2374         rw_exit(&irp->irp_lock);
2375         rw_exit(&iport->iport_lock);

2377         atomic_inc_16(&irp->irp_nonfcp_xchg_count);
2377         atomic_add_16(&irp->irp_nonfcp_xchg_count, 1);
2378         atomic_or_32(&icmd->icmd_flags, ICMD_KNOWN_TO_FCA);
2379         icmd->icmd_start_time = ddi_get_lbolt();
2380         ret = iport->iport_port->port_send_cmd(cmd);
2381         if (ret != FCT_SUCCESS) {
2382                 atomic_and_32(&icmd->icmd_flags, ~ICMD_KNOWN_TO_FCA);
2383                 fct_queue_cmd_for_termination(cmd, ret);
2384         }
2385 }
```

*unchanged_portion_omitted_*

```
2757 #ifdef  lint
2758 #define FCT_VERIFY_RSCN()       _NOTE(EMPTY)
2759 #else
2760 #define FCT_VERIFY_RSCN()                                               \
2761 do {                                                                    \
2762         ct_cmd = fct_create_solct(port, irp->irp_rp, NS_GID_PN,         \
2763             fct_gid_cb);                                                \
2764         if (ct_cmd) {                                                   \
2765                 uint32_t cnt;                                           \
2766                 cnt = atomic_inc_32_nv(&irp->irp_rscn_counter); \
2766                 cnt = atomic_add_32_nv(&irp->irp_rscn_counter, 1);      \
2767                 CMD_TO_ICMD(ct_cmd)->icmd_cb_private =                  \
2768                     INT2PTR(cnt, void *);                               \
2769                 irp->irp_flags |= IRP_RSCN_QUEUED;                      \
2770                 fct_post_to_solcmd_queue(port, ct_cmd);                 \
2771         }                                                               \
2772 } while (0)
```

*unchanged_portion_omitted_*

```
**********************************************************
   98679 Mon Jul 28 07:44:32 2014
new/usr/src/uts/common/io/comstar/port/fct/fct.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1131 static uint32_t taskq_cntr = 0;

1133 fct_status_t
1134 fct_register_local_port(fct_local_port_t *port)
1135 {
1136         fct_i_local_port_t      *iport;
1137         stmf_local_port_t       *lport;
1138         fct_cmd_slot_t          *slot;
1139         int                     i;
1140         char                    taskq_name[FCT_TASKQ_NAME_LEN];

1142         iport = (fct_i_local_port_t *)port->port_fct_private;
1143         if (port->port_fca_version != FCT_FCA_MODREV_1) {
1144                 cmn_err(CE_WARN,
1145                     "fct: %s driver version mismatch",
1146                     port->port_default_alias);
1147                 return (FCT_FAILURE);
1148         }
1149         if (port->port_default_alias) {
1150                 int l = strlen(port->port_default_alias);

1152                 if (l < 16) {
1153                         iport->iport_alias = iport->iport_alias_mem;
1154                 } else {
1155                         iport->iport_alias =
1156                             (char *)kmem_zalloc(l+1, KM_SLEEP);
1157                 }
1158                 (void) strcpy(iport->iport_alias, port->port_default_alias);
1159         } else {
1160                 iport->iport_alias = NULL;
1161         }
1162         stmf_wwn_to_devid_desc((scsi_devid_desc_t *)iport->iport_id,
1163             port->port_pwwn, PROTOCOL_FIBRE_CHANNEL);
1164         (void) snprintf(taskq_name, sizeof (taskq_name), "stmf_fct_taskq_%d",
1165             atomic_inc_32_nv(&taskq_cntr));
1165             atomic_add_32_nv(&taskq_cntr, 1));
1166         if ((iport->iport_worker_taskq = ddi_taskq_create(NULL,
1167             taskq_name, 1, TASKQ_DEFAULTPRI, 0)) == NULL) {
1168                 return (FCT_FAILURE);
1169         }
1170         mutex_init(&iport->iport_worker_lock, NULL, MUTEX_DRIVER, NULL);
1171         cv_init(&iport->iport_worker_cv, NULL, CV_DRIVER, NULL);
1172         rw_init(&iport->iport_lock, NULL, RW_DRIVER, NULL);
1173         sema_init(&iport->iport_rls_sema, 0, NULL, SEMA_DRIVER, NULL);

1175         /* Remote port mgmt */
1176         iport->iport_rp_slots = (fct_i_remote_port_t **)kmem_zalloc(
1177             port->port_max_logins * sizeof (fct_i_remote_port_t *), KM_SLEEP);
1178         iport->iport_rp_tb = kmem_zalloc(rportid_table_size *
1179             sizeof (fct_i_remote_port_t *), KM_SLEEP);

1181         /* fct_cmds for SCSI traffic */
1182         iport->iport_total_alloced_ncmds = 0;
1183         iport->iport_cached_ncmds = 0;
1184         port->port_fca_fcp_cmd_size =
1185             (port->port_fca_fcp_cmd_size + 7) & ~7;
1186         iport->iport_cached_cmdlist = NULL;
1187         mutex_init(&iport->iport_cached_cmd_lock, NULL, MUTEX_DRIVER, NULL);
```



```
1189         /* Initialize cmd slots */
1190         iport->iport_cmd_slots = (fct_cmd_slot_t *)kmem_zalloc(
1191             port->port_max_xchges * sizeof (fct_cmd_slot_t), KM_SLEEP);
1192         iport->iport_next_free_slot = 0;
1193         for (i = 0; i < port->port_max_xchges; ) {
1194                 slot = &iport->iport_cmd_slots[i];
1195                 slot->slot_no = (uint16_t)i;
1196                 slot->slot_next = (uint16_t)(++i);
1197         }
1198         slot->slot_next = FCT_SLOT_EOL;
1199         iport->iport_nslots_free = port->port_max_xchges;

1201         iport->iport_task_green_limit =
1202             (port->port_max_xchges * FCT_TASK_GREEN_LIMIT) / 100;
1203         iport->iport_task_yellow_limit =
1204             (port->port_max_xchges * FCT_TASK_YELLOW_LIMIT) / 100;
1205         iport->iport_task_red_limit =
1206             (port->port_max_xchges * FCT_TASK_RED_LIMIT) / 100;

1208         /* Start worker thread */
1209         atomic_and_32(&iport->iport_flags, ~IPORT_TERMINATE_WORKER);
1210         (void) ddi_taskq_dispatch(iport->iport_worker_taskq,
1211             fct_port_worker, port, DDI_SLEEP);
1212         /* Wait for taskq to start */
1213         while ((iport->iport_flags & IPORT_WORKER_RUNNING) == 0) {
1214                 delay(1);
1215         }

1217         lport = port->port_lport;
1218         lport->lport_id = (scsi_devid_desc_t *)iport->iport_id;
1219         lport->lport_alias = iport->iport_alias;
1220         lport->lport_pp = port->port_pp;
1221         port->port_fds->fds_ds->ds_alloc_data_buf = fct_alloc_dbuf;
1222         port->port_fds->fds_ds->ds_free_data_buf = fct_free_dbuf;
1223         port->port_fds->fds_ds->ds_setup_dbuf = fct_setup_dbuf;
1224         port->port_fds->fds_ds->ds_teardown_dbuf = fct_teardown_dbuf;
1225         lport->lport_ds = port->port_fds->fds_ds;
1226         lport->lport_xfer_data = fct_xfer_scsi_data;
1227         lport->lport_send_status = fct_send_scsi_status;
1228         lport->lport_task_free = fct_scsi_task_free;
1229         lport->lport_abort = fct_scsi_abort;
1230         lport->lport_ctl = fct_ctl;
1231         lport->lport_info = fct_info;
1232         lport->lport_event_handler = fct_event_handler;
1233         /* set up as alua participating port */
1234         stmf_set_port_alua(lport);
1235         if (stmf_register_local_port(port->port_lport) != FCT_SUCCESS) {
1236                 goto fct_regport_fail1;
1237         }
1238         (void) stmf_lport_add_event(lport, LPORT_EVENT_INITIAL_LUN_MAPPED);

1240         mutex_enter(&fct_global_mutex);
1241         iport->iport_next = fct_iport_list;
1242         iport->iport_prev = NULL;
1243         if (iport->iport_next)
1244                 iport->iport_next->iport_prev = iport;
1245         fct_iport_list = iport;
1246         mutex_exit(&fct_global_mutex);

1248         fct_init_kstats(iport);

1250         fct_log_local_port_event(port, ESC_SUNFC_PORT_ATTACH);

1252         return (FCT_SUCCESS);

1254 fct_regport_fail1:;
```

```
1255         /* Stop the taskq 1st */
1256         if (iport->iport_flags & IPORT_WORKER_RUNNING) {
1257                 atomic_or_32(&iport->iport_flags, IPORT_TERMINATE_WORKER);
1258                 cv_broadcast(&iport->iport_worker_cv);
1259                 while (iport->iport_flags & IPORT_WORKER_RUNNING) {
1260                         delay(1);
1261                 }
1262         }
1263         ddi_taskq_destroy(iport->iport_worker_taskq);
1264         if (iport->iport_rp_tb) {
1265                 kmem_free(iport->iport_rp_tb, rportid_table_size *
1266                     sizeof (fct_i_remote_port_t *));
1267         }
1268         return (FCT_FAILURE);
1269 }
______unchanged_portion_omitted_

1427 /*
1428  * Called with irp_lock and iport_lock held as writer.
1429  */
1430 void
1431 fct_deque_rp(fct_i_local_port_t *iport, fct_i_remote_port_t *irp)
1432 {
1433         fct_i_remote_port_t     *irp_next = NULL;
1434         fct_i_remote_port_t     *irp_last = NULL;
1435         int hash_key                      =
1436             FCT_PORTID_HASH_FUNC(irp->irp_portid);

1438         irp_next = iport->iport_rp_tb[hash_key];
1439         irp_last = NULL;
1440         while (irp_next != NULL) {
1441                 if (irp == irp_next) {
1442                         if (irp->irp_flags & IRP_PLOGI_DONE) {
1443                                 atomic_dec_32(&iport->iport_nrps_login);
1443                                 atomic_add_32(&iport->iport_nrps_login, -1);
1444                         }
1445                         atomic_and_32(&irp->irp_flags,
1446                             ~(IRP_PLOGI_DONE | IRP_PRLI_DONE));
1447                         break;
1448                 }
1449                 irp_last = irp_next;
1450                 irp_next = irp_next->irp_next;
1451         }

1453         if (irp_next) {
1454                 if (irp_last == NULL) {
1455                         iport->iport_rp_tb[hash_key] =
1456                             irp->irp_next;
1457                 } else {
1458                         irp_last->irp_next = irp->irp_next;
1459                 }
1460                 irp->irp_next = NULL;
1461                 iport->iport_nrps--;
1462         }
1463 }
______unchanged_portion_omitted_

1608 fct_cmd_t *
1609 fct_scsi_task_alloc(fct_local_port_t *port, uint16_t rp_handle,
1610                 uint32_t rportid, uint8_t *lun, uint16_t cdb_length,
1611                 uint16_t task_ext)
1612 {
1613         fct_cmd_t *cmd;
1614         fct_i_cmd_t *icmd;
1615         fct_i_local_port_t *iport =
1616             (fct_i_local_port_t *)port->port_fct_private;
```



```
1617         fct_i_remote_port_t *irp;
1618         scsi_task_t *task;
1619         fct_remote_port_t *rp;
1620         uint16_t cmd_slot;

1622         rw_enter(&iport->iport_lock, RW_READER);
1623         if ((iport->iport_link_state & S_LINK_ONLINE) == 0) {
1624                 rw_exit(&iport->iport_lock);
1625                 stmf_trace(iport->iport_alias, "cmd alloc called while the port"
1626                     " was offline");
1627                 return (NULL);
1628         }

1630         if (rp_handle == FCT_HANDLE_NONE) {
1631                 irp = fct_portid_to_portptr(iport, rportid);
1632                 if (irp == NULL) {
1633                         rw_exit(&iport->iport_lock);
1634                         stmf_trace(iport->iport_alias, "cmd received from "
1635                             "non existent port %x", rportid);
1636                         return (NULL);
1637                 }
1638         } else {
1639                 if ((rp_handle >= port->port_max_logins) ||
1640                     ((irp = iport->iport_rp_slots[rp_handle]) == NULL)) {
1641                         rw_exit(&iport->iport_lock);
1642                         stmf_trace(iport->iport_alias, "cmd received from "
1643                             "invalid port handle %x", rp_handle);
1644                         return (NULL);
1645                 }
1646         }
1647         rp = irp->irp_rp;

1649         rw_enter(&irp->irp_lock, RW_READER);
1650         if ((irp->irp_flags & IRP_PRLI_DONE) == 0) {
1651                 rw_exit(&irp->irp_lock);
1652                 rw_exit(&iport->iport_lock);
1653                 stmf_trace(iport->iport_alias, "cmd alloc called while fcp "
1654                     "login was not done. portid=%x, rp=%p", rp->rp_id, rp);
1655                 return (NULL);
1656         }

1658         mutex_enter(&iport->iport_cached_cmd_lock);
1659         if ((icmd = iport->iport_cached_cmdlist) != NULL) {
1660                 iport->iport_cached_cmdlist = icmd->icmd_next;
1661                 iport->iport_cached_ncmds--;
1662                 cmd = icmd->icmd_cmd;
1663         } else {
1664                 icmd = NULL;
1665         }
1666         mutex_exit(&iport->iport_cached_cmd_lock);
1667         if (icmd == NULL) {
1668                 cmd = (fct_cmd_t *)fct_alloc(FCT_STRUCT_CMD_FCP_XCHG,
1669                     port->port_fca_fcp_cmd_size, 0);
1670                 if (cmd == NULL) {
1671                         rw_exit(&irp->irp_lock);
1672                         rw_exit(&iport->iport_lock);
1673                         stmf_trace(iport->iport_alias, "Ran out of "
1674                             "memory, port=%p", port);
1675                         return (NULL);
1676                 }

1678                 icmd = (fct_i_cmd_t *)cmd->cmd_fct_private;
1679                 icmd->icmd_next = NULL;
1680                 cmd->cmd_port = port;
1681                 atomic_inc_32(&iport->iport_total_alloced_ncmds);
1681                 atomic_add_32(&iport->iport_total_alloced_ncmds, 1);
```

```
1682         }

1684         /*
1685          * The accuracy of iport_max_active_ncmds is not important
1686          */
1687         if ((iport->iport_total_alloced_ncmds - iport->iport_cached_ncmds) >
1688             iport->iport_max_active_ncmds) {
1689                 iport->iport_max_active_ncmds =
1690                     iport->iport_total_alloced_ncmds -
1691                     iport->iport_cached_ncmds;
1692         }

1694         /* Lets get a slot */
1695         cmd_slot = fct_alloc_cmd_slot(iport, cmd);
1696         if (cmd_slot == FCT_SLOT_EOL) {
1697                 rw_exit(&irp->irp_lock);
1698                 rw_exit(&iport->iport_lock);
1699                 stmf_trace(iport->iport_alias, "Ran out of xchg resources");
1700                 cmd->cmd_handle = 0;
1701                 fct_cmd_free(cmd);
1702                 return (NULL);
1703         }
1704         atomic_inc_16(&irp->irp_fcp_xchg_count);
1704         atomic_add_16(&irp->irp_fcp_xchg_count, 1);
1705         cmd->cmd_rp = rp;
1706         icmd->icmd_flags |= ICMD_IN_TRANSITION | ICMD_KNOWN_TO_FCA;
1707         rw_exit(&irp->irp_lock);
1708         rw_exit(&iport->iport_lock);

1710         icmd->icmd_start_time = ddi_get_lbolt();

1712         cmd->cmd_specific = stmf_task_alloc(port->port_lport, irp->irp_session,
1713             lun, cdb_length, task_ext);
1714         if ((task = (scsi_task_t *)cmd->cmd_specific) != NULL) {
1715                 task->task_port_private = cmd;
1716                 return (cmd);
1717         }

1719         fct_cmd_free(cmd);

1721         return (NULL);
1722 }
_____unchanged_portion_omitted_

1812 /*
1813  * This function bypasses fct_handle_els()
1814  */
1815 void
1816 fct_post_implicit_logo(fct_cmd_t *cmd)
1817 {
1818         fct_local_port_t *port = cmd->cmd_port;
1819         fct_i_local_port_t *iport =
1820             (fct_i_local_port_t *)port->port_fct_private;
1821         fct_i_cmd_t *icmd = (fct_i_cmd_t *)cmd->cmd_fct_private;
1822         fct_remote_port_t *rp = cmd->cmd_rp;
1823         fct_i_remote_port_t *irp = (fct_i_remote_port_t *)rp->rp_fct_private;

1825         icmd->icmd_start_time = ddi_get_lbolt();

1827         rw_enter(&irp->irp_lock, RW_WRITER);
1828         atomic_or_32(&icmd->icmd_flags, ICMD_IMPLICIT_CMD_HAS_RESOURCE);
1829         atomic_inc_16(&irp->irp_nonfcp_xchg_count);
1830         atomic_inc_16(&irp->irp_sa_elses_count);
1829         atomic_add_16(&irp->irp_nonfcp_xchg_count, 1);
1830         atomic_add_16(&irp->irp_sa_elses_count, 1);
1831         /*
```



```
1832          * An implicit LOGO can also be posted to a irp where a PLOGI might
1833          * be in process. That PLOGI will reset this flag and decrement the
1834          * iport_nrps_login counter.
1835          */
1836         if (irp->irp_flags & IRP_PLOGI_DONE) {
1837                 atomic_dec_32(&iport->iport_nrps_login);
1837                 atomic_add_32(&iport->iport_nrps_login, -1);
1838         }
1839         atomic_and_32(&irp->irp_flags, ~(IRP_PLOGI_DONE | IRP_PRLI_DONE));
1840         atomic_or_32(&icmd->icmd_flags, ICMD_SESSION_AFFECTING);
1841         fct_post_to_discovery_queue(iport, irp, icmd);
1842         rw_exit(&irp->irp_lock);
1843 }

1845 /*
1846  * called with iport_lock held, return the slot number
1847  */
1848 uint16_t
1849 fct_alloc_cmd_slot(fct_i_local_port_t *iport, fct_cmd_t *cmd)
1850 {
1851         uint16_t cmd_slot;
1852         uint32_t old, new;
1853         fct_i_cmd_t *icmd = (fct_i_cmd_t *)cmd->cmd_fct_private;

1855         do {
1856                 old = iport->iport_next_free_slot;
1857                 cmd_slot = old & 0xFFFF;
1858                 if (cmd_slot == FCT_SLOT_EOL)
1859                         return (cmd_slot);
1860                 /*
1861                  * We use high order 16 bits as a counter which keeps on
1862                  * incrementing to avoid ABA issues with atomic lists.
1863                  */
1864                 new = ((old + (0x10000)) & 0xFFFF0000);
1865                 new |= iport->iport_cmd_slots[cmd_slot].slot_next;
1866         } while (atomic_cas_32(&iport->iport_next_free_slot, old, new) != old);

1868         atomic_dec_16(&iport->iport_nslots_free);
1868         atomic_add_16(&iport->iport_nslots_free, -1);
1869         iport->iport_cmd_slots[cmd_slot].slot_cmd = icmd;
1870         cmd->cmd_handle = (uint32_t)cmd_slot | 0x80000000 |
1871             (((uint32_t)(iport->iport_cmd_slots[cmd_slot].slot_uniq_cntr))
1872             << 24);
1873         return (cmd_slot);
1874 }
_____unchanged_portion_omitted_

2028 void
2029 fct_cmd_free(fct_cmd_t *cmd)
2030 {
2031         char                    info[FCT_INFO_LEN];
2032         fct_i_cmd_t             *icmd = (fct_i_cmd_t *)cmd->cmd_fct_private;
2033         fct_local_port_t        *port = cmd->cmd_port;
2034         fct_i_local_port_t      *iport = (fct_i_local_port_t *)
2035             port->port_fct_private;
2036         fct_i_remote_port_t     *irp = NULL;
2037         int                     do_abts_acc = 0;
2038         uint32_t                old, new;

2040         ASSERT(!mutex_owned(&iport->iport_worker_lock));
2041         /* Give the slot back */
2042         if (CMD_HANDLE_VALID(cmd->cmd_handle)) {
2043                 uint16_t n = CMD_HANDLE_SLOT_INDEX(cmd->cmd_handle);
2044                 fct_cmd_slot_t *slot;

2046                 /*
```

```
2047                 * If anything went wrong, grab the lock as writer. This is
2048                 * probably unnecessary.
2049                 */
2050                if ((cmd->cmd_comp_status != FCT_SUCCESS) ||
2051                    (icmd->icmd_flags & ICMD_ABTS_RECEIVED)) {
2052                        rw_enter(&iport->iport_lock, RW_WRITER);
2053                } else {
2054                        rw_enter(&iport->iport_lock, RW_READER);
2055                }

2057                if ((icmd->icmd_flags & ICMD_ABTS_RECEIVED) &&
2058                    (cmd->cmd_link != NULL)) {
2059                        do_abts_acc = 1;
2060                }

2062                /* XXX Validate slot before freeing */

2064                slot = &iport->iport_cmd_slots[n];
2065                slot->slot_uniq_cntr++;
2066                slot->slot_cmd = NULL;
2067                do {
2068                        old = iport->iport_next_free_slot;
2069                        slot->slot_next = old & 0xFFFF;
2070                        new = (old + 0x10000) & 0xFFFF0000;
2071                        new |= slot->slot_no;
2072                } while (atomic_cas_32(&iport->iport_next_free_slot,
2073                    old, new) != old);
2074                cmd->cmd_handle = 0;
2075                atomic_inc_16(&iport->iport_nslots_free);
2075                atomic_add_16(&iport->iport_nslots_free, 1);
2076                if (cmd->cmd_rp) {
2077                        irp = (fct_i_remote_port_t *)
2078                            cmd->cmd_rp->rp_fct_private;
2079                        if (cmd->cmd_type == FCT_CMD_FCP_XCHG)
2080                                atomic_dec_16(&irp->irp_fcp_xchg_count);
2080                                atomic_add_16(&irp->irp_fcp_xchg_count, -1);
2081                        else
2082                                atomic_dec_16(&irp->irp_nonfcp_xchg_count);
2082                                atomic_add_16(&irp->irp_nonfcp_xchg_count, -1);
2083                }
2084                rw_exit(&iport->iport_lock);
2085        } else if ((icmd->icmd_flags & ICMD_IMPLICIT) &&
2086            (icmd->icmd_flags & ICMD_IMPLICIT_CMD_HAS_RESOURCE)) {
2087                /* for implicit cmd, no cmd slot is used */
2088                if (cmd->cmd_rp) {
2089                        irp = (fct_i_remote_port_t *)
2090                            cmd->cmd_rp->rp_fct_private;
2091                        if (cmd->cmd_type == FCT_CMD_FCP_XCHG)
2092                                atomic_dec_16(&irp->irp_fcp_xchg_count);
2092                                atomic_add_16(&irp->irp_fcp_xchg_count, -1);
2093                        else
2094                                atomic_dec_16(&irp->irp_nonfcp_xchg_count);
2094                                atomic_add_16(&irp->irp_nonfcp_xchg_count, -1);
2095                }
2096        }

2098        if (do_abts_acc) {
2099                fct_cmd_t *lcmd = cmd->cmd_link;
2100                fct_fill_abts_acc(lcmd);
2101                if (port->port_send_cmd_response(lcmd,
2102                    FCT_IOF_FORCE_FCA_DONE) != FCT_SUCCESS) {
2103                        /*
2104                         * XXX Throw HBA fatal error event
2105                         * Later shutdown svc will terminate the ABTS in the end
2106                         */
2107                        (void) snprintf(info, sizeof (info),
```



```
2108                            "fct_cmd_free: iport-%p, ABTS_ACC"
2109                            " port_send_cmd_response failed", (void *)iport);
2110                        (void) fct_port_shutdown(iport->iport_port,
2111                            STMF_RFLAG_FATAL_ERROR | STMF_RFLAG_RESET, info);
2112                        return;
2113                } else {
2114                        fct_cmd_free(lcmd);
2115                        cmd->cmd_link = NULL;
2116                }
2117        }

2119        /* Free the cmd */
2120        if (cmd->cmd_type == FCT_CMD_FCP_XCHG) {
2121                if (iport->iport_cached_ncmds < max_cached_ncmds) {
2122                        icmd->icmd_flags = 0;
2123                        mutex_enter(&iport->iport_cached_cmd_lock);
2124                        icmd->icmd_next = iport->iport_cached_cmdlist;
2125                        iport->iport_cached_cmdlist = icmd;
2126                        iport->iport_cached_ncmds++;
2127                        mutex_exit(&iport->iport_cached_cmd_lock);
2128                } else {
2129                        atomic_dec_32(&iport->iport_total_alloced_ncmds);
2129                        atomic_add_32(&iport->iport_total_alloced_ncmds, -1);
2130                        fct_free(cmd);
2131                }
2132        } else {
2133                fct_free(cmd);
2134        }
2135 }
```

_____unchanged_portion_omitted_

```
**********************************************************
  180892 Mon Jul 28 07:44:33 2014
new/usr/src/uts/common/io/comstar/port/qlt/qlt.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1674 /*
1675  * All of these ioctls are unstable interfaces which are meant to be used
1676  * in a controlled lab env. No formal testing will be (or needs to be) done
1677  * for these ioctls. Specially note that running with an additional
1678  * uploaded firmware is not supported and is provided here for test
1679  * purposes only.
1680  */
1681 /* ARGSUSED */
1682 static int
1683 qlt_ioctl(dev_t dev, int cmd, intptr_t data, int mode,
1684     cred_t *credp, int *rval)
1685 {
1686         qlt_state_t     *qlt;
1687         int             ret = 0;
1688 #ifdef _LITTLE_ENDIAN
1689         int             i;
1690 #endif
1691         stmf_iocdata_t  *iocd;
1692         void            *ibuf = NULL;
1693         void            *obuf = NULL;
1694         uint32_t        *intp;
1695         qlt_fw_info_t   *fwi;
1696         mbox_cmd_t      *mcp;
1697         fct_status_t    st;
1698         char            info[QLT_INFO_LEN];
1699         fct_status_t    ret2;

1701         if (drv_priv(credp) != 0)
1702                 return (EPERM);

1704         qlt = ddi_get_soft_state(qlt_state, (int32_t)getminor(dev));
1705         ret = stmf_copyin_iocdata(data, mode, &iocd, &ibuf, &obuf);
1706         if (ret)
1707                 return (ret);
1708         iocd->stmf_error = 0;

1710         switch (cmd) {
1711         case QLT_IOCTL_FETCH_FWDUMP:
1712                 if (iocd->stmf_obuf_size < QLT_FWDUMP_BUFSIZE) {
1713                         EL(qlt, "FETCH_FWDUMP obuf_size=%d < %d\n",
1714                             iocd->stmf_obuf_size, QLT_FWDUMP_BUFSIZE);
1715                         ret = EINVAL;
1716                         break;
1717                 }
1718                 mutex_enter(&qlt->qlt_ioctl_lock);
1719                 if (!(qlt->qlt_ioctl_flags & QLT_FWDUMP_ISVALID)) {
1720                         mutex_exit(&qlt->qlt_ioctl_lock);
1721                         ret = ENODATA;
1722                         EL(qlt, "no fwdump\n");
1723                         iocd->stmf_error = QLTIO_NO_DUMP;
1724                         break;
1725                 }
1726                 if (qlt->qlt_ioctl_flags & QLT_FWDUMP_INPROGRESS) {
1727                         mutex_exit(&qlt->qlt_ioctl_lock);
1728                         ret = EBUSY;
1729                         EL(qlt, "fwdump inprogress\n");
1730                         iocd->stmf_error = QLTIO_DUMP_INPROGRESS;
1731                         break;
1732                 }
```



```
1733                 if (qlt->qlt_ioctl_flags & QLT_FWDUMP_FETCHED_BY_USER) {
1734                         mutex_exit(&qlt->qlt_ioctl_lock);
1735                         ret = EEXIST;
1736                         EL(qlt, "fwdump already fetched\n");
1737                         iocd->stmf_error = QLTIO_ALREADY_FETCHED;
1738                         break;
1739                 }
1740                 bcopy(qlt->qlt_fwdump_buf, obuf, QLT_FWDUMP_BUFSIZE);
1741                 qlt->qlt_ioctl_flags |= QLT_FWDUMP_FETCHED_BY_USER;
1742                 mutex_exit(&qlt->qlt_ioctl_lock);

1744                 break;

1746         case QLT_IOCTL_TRIGGER_FWDUMP:
1747                 if (qlt->qlt_state != FCT_STATE_ONLINE) {
1748                         ret = EACCES;
1749                         iocd->stmf_error = QLTIO_NOT_ONLINE;
1750                         break;
1751                 }
1752                 (void) snprintf(info, sizeof (info), "qlt_ioctl: qlt-%p, "
1753                     "user triggered FWDUMP with RFLAG_RESET", (void *)qlt);
1754                 if ((ret2 = fct_port_shutdown(qlt->qlt_port,
1755                     STMF_RFLAG_USER_REQUEST | STMF_RFLAG_RESET |
1756                     STMF_RFLAG_COLLECT_DEBUG_DUMP, info)) != FCT_SUCCESS) {
1757                         EL(qlt, "TRIGGER_FWDUMP fct_port_shutdown status="
1758                             "%llxh\n", ret2);
1759                         ret = EIO;
1760                 }
1761                 break;
1762         case QLT_IOCTL_UPLOAD_FW:
1763                 if ((iocd->stmf_ibuf_size < 1024) ||
1764                     (iocd->stmf_ibuf_size & 3)) {
1765                         EL(qlt, "UPLOAD_FW ibuf_size=%d < 1024\n",
1766                             iocd->stmf_ibuf_size);
1767                         ret = EINVAL;
1768                         iocd->stmf_error = QLTIO_INVALID_FW_SIZE;
1769                         break;
1770                 }
1771                 intp = (uint32_t *)ibuf;
1772 #ifdef _LITTLE_ENDIAN
1773                 for (i = 0; (i << 2) < iocd->stmf_ibuf_size; i++) {
1774                         intp[i] = BSWAP_32(intp[i]);
1775                 }
1776 #endif
1777                 if (((intp[3] << 2) >= iocd->stmf_ibuf_size) ||
1778                     (((intp[intp[3] + 3] + intp[3]) << 2) !=
1779                     iocd->stmf_ibuf_size)) {
1780                         EL(qlt, "UPLOAD_FW fw_size=%d >= %d\n", intp[3] << 2,
1781                             iocd->stmf_ibuf_size);
1782                         ret = EINVAL;
1783                         iocd->stmf_error = QLTIO_INVALID_FW_SIZE;
1784                         break;
1785                 }
1786                 if ((qlt->qlt_81xx_chip && ((intp[8] & 8) == 0)) ||
1787                     (qlt->qlt_25xx_chip && ((intp[8] & 4) == 0)) ||
1788                     (!qlt->qlt_25xx_chip && !qlt->qlt_81xx_chip &&
1789                     ((intp[8] & 3) == 0))) {
1790                         EL(qlt, "UPLOAD_FW fw_type=%d\n", intp[8]);
1791                         ret = EACCES;
1792                         iocd->stmf_error = QLTIO_INVALID_FW_TYPE;
1793                         break;
1794                 }

1796                 /* Everything looks ok, lets copy this firmware */
1797                 if (qlt->fw_code01) {
1798                         kmem_free(qlt->fw_code01, (qlt->fw_length01 +
```

```
1799                            qlt->fw_length02) << 2);
1800                    qlt->fw_code01 = NULL;
1801            } else {
1802                    atomic_inc_32(&qlt_loaded_counter);
1802                    atomic_add_32(&qlt_loaded_counter, 1);
1803            }
1804            qlt->fw_length01 = intp[3];
1805            qlt->fw_code01 = (uint32_t *)kmem_alloc(iocd->stmf_ibuf_size,
1806                KM_SLEEP);
1807            bcopy(intp, qlt->fw_code01, iocd->stmf_ibuf_size);
1808            qlt->fw_addr01 = intp[2];
1809            qlt->fw_code02 = &qlt->fw_code01[intp[3]];
1810            qlt->fw_addr02 = qlt->fw_code02[2];
1811            qlt->fw_length02 = qlt->fw_code02[3];
1812            break;

1814    case QLT_IOCTL_CLEAR_FW:
1815            if (qlt->fw_code01) {
1816                    kmem_free(qlt->fw_code01, (qlt->fw_length01 +
1817                        qlt->fw_length02) << 2);
1818                    qlt->fw_code01 = NULL;
1819                    atomic_dec_32(&qlt_loaded_counter);
1819                    atomic_add_32(&qlt_loaded_counter, -1);
1820            }
1821            break;

1823    case QLT_IOCTL_GET_FW_INFO:
1824            if (iocd->stmf_obuf_size != sizeof (qlt_fw_info_t)) {
1825                    EL(qlt, "GET_FW_INFO obuf_size=%d != %d\n",
1826                        iocd->stmf_obuf_size, sizeof (qlt_fw_info_t));
1827                    ret = EINVAL;
1828                    break;
1829            }
1830            fwi = (qlt_fw_info_t *)obuf;
1831            if (qlt->qlt_stay_offline) {
1832                    fwi->fwi_stay_offline = 1;
1833            }
1834            if (qlt->qlt_state == FCT_STATE_ONLINE) {
1835                    fwi->fwi_port_active = 1;
1836            }
1837            fwi->fwi_active_major = qlt->fw_major;
1838            fwi->fwi_active_minor = qlt->fw_minor;
1839            fwi->fwi_active_subminor = qlt->fw_subminor;
1840            fwi->fwi_active_attr = qlt->fw_attr;
1841            if (qlt->fw_code01) {
1842                    fwi->fwi_fw_uploaded = 1;
1843                    fwi->fwi_loaded_major = (uint16_t)qlt->fw_code01[4];
1844                    fwi->fwi_loaded_minor = (uint16_t)qlt->fw_code01[5];
1845                    fwi->fwi_loaded_subminor = (uint16_t)qlt->fw_code01[6];
1846                    fwi->fwi_loaded_attr = (uint16_t)qlt->fw_code01[7];
1847            }
1848            if (qlt->qlt_81xx_chip) {
1849                    fwi->fwi_default_major = (uint16_t)fw8100_code01[4];
1850                    fwi->fwi_default_minor = (uint16_t)fw8100_code01[5];
1851                    fwi->fwi_default_subminor = (uint16_t)fw8100_code01[6];
1852                    fwi->fwi_default_attr = (uint16_t)fw8100_code01[7];
1853            } else if (qlt->qlt_25xx_chip) {
1854                    fwi->fwi_default_major = (uint16_t)fw2500_code01[4];
1855                    fwi->fwi_default_minor = (uint16_t)fw2500_code01[5];
1856                    fwi->fwi_default_subminor = (uint16_t)fw2500_code01[6];
1857                    fwi->fwi_default_attr = (uint16_t)fw2500_code01[7];
1858            } else {
1859                    fwi->fwi_default_major = (uint16_t)fw2400_code01[4];
1860                    fwi->fwi_default_minor = (uint16_t)fw2400_code01[5];
1861                    fwi->fwi_default_subminor = (uint16_t)fw2400_code01[6];
1862                    fwi->fwi_default_attr = (uint16_t)fw2400_code01[7];
```



```
1863            }
1864            break;

1866    case QLT_IOCTL_STAY_OFFLINE:
1867            if (!iocd->stmf_ibuf_size) {
1868                    EL(qlt, "STAY_OFFLINE ibuf_size=%d\n",
1869                        iocd->stmf_ibuf_size);
1870                    ret = EINVAL;
1871                    break;
1872            }
1873            if (*((char *)ibuf)) {
1874                    qlt->qlt_stay_offline = 1;
1875            } else {
1876                    qlt->qlt_stay_offline = 0;
1877            }
1878            break;

1880    case QLT_IOCTL_MBOX:
1881            if ((iocd->stmf_ibuf_size < sizeof (qlt_ioctl_mbox_t)) ||
1882                (iocd->stmf_obuf_size < sizeof (qlt_ioctl_mbox_t))) {
1883                    EL(qlt, "IOCTL_MBOX ibuf_size=%d, obuf_size=%d\n",
1884                        iocd->stmf_ibuf_size, iocd->stmf_obuf_size);
1885                    ret = EINVAL;
1886                    break;
1887            }
1888            mcp = qlt_alloc_mailbox_command(qlt, 0);
1889            if (mcp == NULL) {
1890                    EL(qlt, "IOCTL_MBOX mcp == NULL\n");
1891                    ret = ENOMEM;
1892                    break;
1893            }
1894            bcopy(ibuf, mcp, sizeof (qlt_ioctl_mbox_t));
1895            st = qlt_mailbox_command(qlt, mcp);
1896            bcopy(mcp, obuf, sizeof (qlt_ioctl_mbox_t));
1897            qlt_free_mailbox_command(qlt, mcp);
1898            if (st != QLT_SUCCESS) {
1899                    if ((st & (~((uint64_t)(0xFFFF)))) == QLT_MBOX_FAILED)
1900                            st = QLT_SUCCESS;
1901            }
1902            if (st != QLT_SUCCESS) {
1903                    EL(qlt, "IOCTL_MBOX status=%xh\n", st);
1904                    ret = EIO;
1905                    switch (st) {
1906                    case QLT_MBOX_NOT_INITIALIZED:
1907                            iocd->stmf_error = QLTIO_MBOX_NOT_INITIALIZED;
1908                            break;
1909                    case QLT_MBOX_BUSY:
1910                            iocd->stmf_error = QLTIO_CANT_GET_MBOXES;
1911                            break;
1912                    case QLT_MBOX_TIMEOUT:
1913                            iocd->stmf_error = QLTIO_MBOX_TIMED_OUT;
1914                            break;
1915                    case QLT_MBOX_ABORTED:
1916                            iocd->stmf_error = QLTIO_MBOX_ABORTED;
1917                            break;
1918                    }
1919            }
1920            break;

1922    case QLT_IOCTL_ELOG:
1923            qlt_dump_el_trace_buffer(qlt);
1924            break;

1926    default:
1927            EL(qlt, "Unknown ioctl-%xh\n", cmd);
1928            ret = ENOTTY;
```

```
1929         }

1931         if (ret == 0) {
1932                 ret = stmf_copyout_iocdata(data, mode, iocd, obuf);
1933         } else if (iocd->stmf_error) {
1934                 (void) stmf_copyout_iocdata(data, mode, iocd, obuf);
1935         }
1936         if (obuf) {
1937                 kmem_free(obuf, iocd->stmf_obuf_size);
1938                 obuf = NULL;
1939         }
1940         if (ibuf) {
1941                 kmem_free(ibuf, iocd->stmf_ibuf_size);
1942                 ibuf = NULL;
1943         }
1944         kmem_free(iocd, sizeof (stmf_iocdata_t));
1945         return (ret);
1946 }
```

*unchanged_portion_omitted_*

```
4658 #ifdef  DEBUG
4659 uint32_t qlt_drop_abort_counter = 0;
4660 #endif

4662 fct_status_t
4663 qlt_abort_cmd(struct fct_local_port *port, fct_cmd_t *cmd, uint32_t flags)
4664 {
4665         qlt_state_t *qlt = (qlt_state_t *)port->port_fca_private;

4667         if ((qlt->qlt_state == FCT_STATE_OFFLINE) ||
4668             (qlt->qlt_state == FCT_STATE_OFFLINING)) {
4669                 return (FCT_NOT_FOUND);
4670         }

4672 #ifdef DEBUG
4673         if (qlt_drop_abort_counter > 0) {
4674                 if (atomic_dec_32_nv(&qlt_drop_abort_counter) == 1)
4674                 if (atomic_add_32_nv(&qlt_drop_abort_counter, -1) == 1)
4675                         return (FCT_SUCCESS);
4676         }
4677 #endif

4679         if (cmd->cmd_type == FCT_CMD_FCP_XCHG) {
4680                 return (qlt_abort_unsol_scsi_cmd(qlt, cmd));
4681         }

4683         if (flags & FCT_IOF_FORCE_FCA_DONE) {
4684                 cmd->cmd_handle = 0;
4685         }

4687         if (cmd->cmd_type == FCT_CMD_RCVD_ABTS) {
4688                 return (qlt_send_abts_response(qlt, cmd, 1));
4689         }

4691         if (cmd->cmd_type == FCT_CMD_RCVD_ELS) {
4692                 return (qlt_abort_purex(qlt, cmd));
4693         }

4695         if ((cmd->cmd_type == FCT_CMD_SOL_ELS) ||
4696             (cmd->cmd_type == FCT_CMD_SOL_CT)) {
4697                 return (qlt_abort_sol_cmd(qlt, cmd));
4698         }
4699         EL(qlt, "cmd->cmd_type = %xh\n", cmd->cmd_type);

4701         ASSERT(0);
4702         return (FCT_FAILURE);
```



```
4703 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   45363 Mon Jul 28 07:44:33 2014
new/usr/src/uts/common/io/comstar/stmf/lun_map.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 238 /*
 239  * destroy lun map for session
 240  */
 241 /* ARGSUSED */
 242 stmf_status_t
 243 stmf_session_destroy_lun_map(stmf_i_local_port_t *ilport,
 244                 stmf_i_scsi_session_t *iss)
 245 {
 246         stmf_lun_map_t *sm;
 247         stmf_i_lu_t *ilu;
 248         uint16_t n;
 249         stmf_lun_map_ent_t *ent;

 251         ASSERT(mutex_owned(&stmf_state.stmf_lock));
 252         /*
 253          * to avoid conflict with updating session's map,
 254          * which only grab stmf_lock
 255          */
 256         sm = iss->iss_sm;
 257         iss->iss_sm = NULL;
 258         iss->iss_hg = NULL;
 259         if (sm->lm_nentries) {
 260                 for (n = 0; n < sm->lm_nentries; n++) {
 261                         if ((ent = (stmf_lun_map_ent_t *)sm->lm_plus[n])
 262                             != NULL) {
 263                                 if (ent->ent_itl_datap) {
 264                                         stmf_do_itl_dereg(ent->ent_lu,
 265                                             ent->ent_itl_datap,
 266                                             STMF_ITL_REASON_IT_NEXUS_LOSS);
 267                                 }
 268                                 ilu = (stmf_i_lu_t *)
 269                                     ent->ent_lu->lu_stmf_private;
 270                                 atomic_dec_32(&ilu->ilu_ref_cnt);
 270                                 atomic_add_32(&ilu->ilu_ref_cnt, -1);
 271                                 kmem_free(sm->lm_plus[n],
 272                                     sizeof (stmf_lun_map_ent_t));
 273                         }
 274                 }
 275                 kmem_free(sm->lm_plus,
 276                     sizeof (stmf_lun_map_ent_t *) * sm->lm_nentries);
 277         }

 279         kmem_free(sm, sizeof (*sm));
 280         return (STMF_SUCCESS);
 281 }
_____unchanged_portion_omitted_
 391 /*
 392  * add lu to a session, stmf_lock is already held
 393  */
 394 stmf_status_t
 395 stmf_add_lu_to_session(stmf_i_local_port_t *ilport,
 396                 stmf_i_scsi_session_t   *iss,
 397                 stmf_lu_t *lu,
 398                 uint8_t *lu_nbr)
 399 {
 400         stmf_lun_map_t *sm = iss->iss_sm;
 401         stmf_status_t ret;
 402         stmf_i_lu_t *ilu = (stmf_i_lu_t *)lu->lu_stmf_private;
 403         stmf_lun_map_ent_t *lun_map_ent;
```



```
 404         uint32_t new_flags = 0;
 405         uint16_t luNbr =
 406             ((uint16_t)lu_nbr[1] | (((uint16_t)(lu_nbr[0] & 0x3F)) << 8));

 408         ASSERT(mutex_owned(&stmf_state.stmf_lock));
 409         ASSERT(!stmf_get_ent_from_map(sm, luNbr));

 411         if ((sm->lm_nluns == 0) &&
 412             ((iss->iss_flags & ISS_BEING_CREATED) == 0)) {
 413                 new_flags = ISS_GOT_INITIAL_LUNS;
 414                 atomic_or_32(&ilport->ilport_flags, ILPORT_SS_GOT_INITIAL_LUNS);
 415                 stmf_state.stmf_process_initial_luns = 1;
 416         }

 418         lun_map_ent = (stmf_lun_map_ent_t *)
 419             kmem_zalloc(sizeof (stmf_lun_map_ent_t), KM_SLEEP);
 420         lun_map_ent->ent_lu = lu;
 421         ret = stmf_add_ent_to_map(sm, (void *)lun_map_ent, lu_nbr);
 422         ASSERT(ret == STMF_SUCCESS);
 423         atomic_inc_32(&ilu->ilu_ref_cnt);
 423         atomic_add_32(&ilu->ilu_ref_cnt, 1);
 424         /*
 425          * do not set lun inventory flag for standby port
 426          * as this would be handled from peer
 427          */
 428         if (ilport->ilport_standby == 0) {
 429                 new_flags |= ISS_LUN_INVENTORY_CHANGED;
 430         }
 431         atomic_or_32(&iss->iss_flags, new_flags);
 432         return (STMF_SUCCESS);
 433 }

 435 /*
 436  * remvoe lu from a session, stmf_lock is already held
 437  */
 438 /* ARGSUSED */
 439 stmf_status_t
 440 stmf_remove_lu_from_session(stmf_i_local_port_t *ilport,
 441                 stmf_i_scsi_session_t *iss,
 442                 stmf_lu_t *lu,
 443                 uint8_t *lu_nbr)
 444 {
 445         stmf_status_t ret;
 446         stmf_i_lu_t *ilu;
 447         stmf_lun_map_t *sm = iss->iss_sm;
 448         stmf_lun_map_ent_t *lun_map_ent;
 449         uint16_t luNbr =
 450             ((uint16_t)lu_nbr[1] | (((uint16_t)(lu_nbr[0] & 0x3F)) << 8));

 452         ASSERT(mutex_owned(&stmf_state.stmf_lock));
 453         lun_map_ent = stmf_get_ent_from_map(sm, luNbr);
 454         ASSERT(lun_map_ent && lun_map_ent->ent_lu == lu);

 456         ilu = (stmf_i_lu_t *)lu->lu_stmf_private;

 458         ret = stmf_remove_ent_from_map(sm, lu_nbr);
 459         ASSERT(ret == STMF_SUCCESS);
 460         atomic_dec_32(&ilu->ilu_ref_cnt);
 460         atomic_add_32(&ilu->ilu_ref_cnt, -1);
 461         iss->iss_flags |= ISS_LUN_INVENTORY_CHANGED;
 462         if (lun_map_ent->ent_itl_datap) {
 463                 stmf_do_itl_dereg(lu, lun_map_ent->ent_itl_datap,
 464                     STMF_ITL_REASON_USER_REQUEST);
 465         }
 466         kmem_free((void *)lun_map_ent, sizeof (stmf_lun_map_ent_t));
 467         return (STMF_SUCCESS);
```

```
 468 }
______unchanged_portion_omitted_

 676 void
 677 stmf_append_id(stmf_id_list_t *idlist, stmf_id_data_t *id)
 678 {
 679         id->id_next = NULL;

 681         if ((id->id_prev = idlist->idl_tail) == NULL) {
 682                 idlist->idl_head = idlist->idl_tail = id;
 683         } else {
 684                 idlist->idl_tail->id_next = id;
 685                 idlist->idl_tail = id;
 686         }
 687         atomic_inc_32(&idlist->id_count);
 687         atomic_add_32(&idlist->id_count, 1);
 688 }

 690 void
 691 stmf_remove_id(stmf_id_list_t *idlist, stmf_id_data_t *id)
 692 {
 693         if (id->id_next) {
 694                 id->id_next->id_prev = id->id_prev;
 695         } else {
 696                 idlist->idl_tail = id->id_prev;
 697         }

 699         if (id->id_prev) {
 700                 id->id_prev->id_next = id->id_next;
 701         } else {
 702                 idlist->idl_head = id->id_next;
 703         }
 704         atomic_dec_32(&idlist->id_count);
 704         atomic_add_32(&idlist->id_count, -1);
 705 }
______unchanged_portion_omitted_
```

```
**********************************************************
  215743 Mon Jul 28 07:44:33 2014
new/usr/src/uts/common/io/comstar/stmf/stmf.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_


2060 static int
2061 stmf_set_alua_state(stmf_alua_state_desc_t *alua_state)
2062 {
2063         stmf_i_local_port_t *ilport;
2064         stmf_i_lu_t *ilu;
2065         stmf_lu_t *lu;
2066         stmf_ic_msg_status_t ic_ret;
2067         stmf_ic_msg_t *ic_reg_lun, *ic_reg_port;
2068         stmf_local_port_t *lport;
2069         int ret = 0;

2071         if (alua_state->alua_state > 1 || alua_state->alua_node > 1) {
2072                 return (EINVAL);
2073         }

2075         mutex_enter(&stmf_state.stmf_lock);
2076         if (alua_state->alua_state == 1) {
2077                 if (pppt_modload() == STMF_FAILURE) {
2078                         ret = EIO;
2079                         goto err;
2080                 }
2081                 if (alua_state->alua_node != 0) {
2082                         /* reset existing rtpids to new base */
2083                         stmf_rtpid_counter = 255;
2084                 }
2085                 stmf_state.stmf_alua_node = alua_state->alua_node;
2086                 stmf_state.stmf_alua_state = 1;
2087                 /* register existing local ports with ppp */
2088                 for (ilport = stmf_state.stmf_ilportlist; ilport != NULL;
2089                     ilport = ilport->ilport_next) {
2090                         /* skip standby ports and non-alua participants */
2091                         if (ilport->ilport_standby == 1 ||
2092                             ilport->ilport_alua == 0) {
2093                                 continue;
2094                         }
2095                         if (alua_state->alua_node != 0) {
2096                                 ilport->ilport_rtpid =
2097                                     atomic_inc_16_nv(&stmf_rtpid_counter);
2097                                     atomic_add_16_nv(&stmf_rtpid_counter, 1);
2098                         }
2099                         lport = ilport->ilport_lport;
2100                         ic_reg_port = ic_reg_port_msg_alloc(
2101                             lport->lport_id, ilport->ilport_rtpid,
2102                             0, NULL, stmf_proxy_msg_id);
2103                         if (ic_reg_port) {
2104                                 ic_ret = ic_tx_msg(ic_reg_port);
2105                                 if (ic_ret == STMF_IC_MSG_SUCCESS) {
2106                                         ilport->ilport_reg_msgid =
2107                                             stmf_proxy_msg_id++;
2108                                 } else {
2109                                         cmn_err(CE_WARN,
2110                                             "error on port registration "
2111                                             "port - %s",
2112                                             ilport->ilport_kstat_tgt_name);
2113                                 }
2114                         }
2115                 }
2116                 /* register existing logical units */
```



```
2117                 for (ilu = stmf_state.stmf_ilulist; ilu != NULL;
2118                     ilu = ilu->ilu_next) {
2119                         if (ilu->ilu_access != STMF_LU_ACTIVE) {
2120                                 continue;
2121                         }
2122                         /* register with proxy module */
2123                         lu = ilu->ilu_lu;
2124                         if (lu->lu_lp && lu->lu_lp->lp_lpif_rev == LPIF_REV_2 &&
2125                             lu->lu_lp->lp_alua_support) {
2126                                 ilu->ilu_alua = 1;
2127                                 /* allocate the register message */
2128                                 ic_reg_lun = ic_reg_lun_msg_alloc(
2129                                     lu->lu_id->ident, lu->lu_lp->lp_name,
2130                                     lu->lu_proxy_reg_arg_len,
2131                                     (uint8_t *)lu->lu_proxy_reg_arg,
2132                                     stmf_proxy_msg_id);
2133                                 /* send the message */
2134                                 if (ic_reg_lun) {
2135                                         ic_ret = ic_tx_msg(ic_reg_lun);
2136                                         if (ic_ret == STMF_IC_MSG_SUCCESS) {
2137                                                 stmf_proxy_msg_id++;
2138                                         }
2139                                 }
2140                         }
2141                 }
2142         } else {
2143                 stmf_state.stmf_alua_state = 0;
2144         }

2146 err:
2147         mutex_exit(&stmf_state.stmf_lock);
2148         return (ret);
2149 }
______unchanged_portion_omitted_

3215 stmf_status_t
3216 stmf_register_local_port(stmf_local_port_t *lport)
3217 {
3218         stmf_i_local_port_t *ilport;
3219         stmf_state_change_info_t ssci;
3220         int start_workers = 0;

3222         mutex_enter(&stmf_state.stmf_lock);
3223         if (stmf_state.stmf_inventory_locked) {
3224                 mutex_exit(&stmf_state.stmf_lock);
3225                 return (STMF_BUSY);
3226         }
3227         ilport = (stmf_i_local_port_t *)lport->lport_stmf_private;
3228         rw_init(&ilport->ilport_lock, NULL, RW_DRIVER, NULL);

3230         ilport->ilport_instance =
3231             id_alloc_nosleep(stmf_state.stmf_ilport_inst_space);
3232         if (ilport->ilport_instance == -1) {
3233                 mutex_exit(&stmf_state.stmf_lock);
3234                 return (STMF_FAILURE);
3235         }
3236         ilport->ilport_next = stmf_state.stmf_ilportlist;
3237         ilport->ilport_prev = NULL;
3238         if (ilport->ilport_next)
3239                 ilport->ilport_next->ilport_prev = ilport;
3240         stmf_state.stmf_ilportlist = ilport;
3241         stmf_state.stmf_nlports++;
3242         if (lport->lport_pp) {
3243                 ((stmf_i_port_provider_t *)
3244                     (lport->lport_pp->pp_stmf_private))->ipp_npps++;
3245         }
```

```
3246         ilport->ilport_tg =
3247             stmf_lookup_group_for_target(lport->lport_id->ident,
3248             lport->lport_id->ident_length);

3250         /*
3251          * rtpid will/must be set if this is a standby port
3252          * only register ports that are not standby (proxy) ports
3253          * and ports that are alua participants (ilport_alua == 1)
3254          */
3255         if (ilport->ilport_standby == 0) {
3256                 ilport->ilport_rtpid = atomic_inc_16_nv(&stmf_rtpid_counter);
3256                 ilport->ilport_rtpid = atomic_add_16_nv(&stmf_rtpid_counter, 1);
3257         }

3259         if (stmf_state.stmf_alua_state == 1 &&
3260             ilport->ilport_standby == 0 &&
3261             ilport->ilport_alua == 1) {
3262                 stmf_ic_msg_t *ic_reg_port;
3263                 stmf_ic_msg_status_t ic_ret;
3264                 stmf_local_port_t *lport;
3265                 lport = ilport->ilport_lport;
3266                 ic_reg_port = ic_reg_port_msg_alloc(
3267                     lport->lport_id, ilport->ilport_rtpid,
3268                     0, NULL, stmf_proxy_msg_id);
3269                 if (ic_reg_port) {
3270                         ic_ret = ic_tx_msg(ic_reg_port);
3271                         if (ic_ret == STMF_IC_MSG_SUCCESS) {
3272                                 ilport->ilport_reg_msgid = stmf_proxy_msg_id++;
3273                         } else {
3274                                 cmn_err(CE_WARN, "error on port registration "
3275                                 "port - %s", ilport->ilport_kstat_tgt_name);
3276                         }
3277                 }
3278         }
3279         STMF_EVENT_ALLOC_HANDLE(ilport->ilport_event_hdl);
3280         stmf_create_kstat_lport(ilport);
3281         if (stmf_workers_state == STMF_WORKERS_DISABLED) {
3282                 stmf_workers_state = STMF_WORKERS_ENABLING;
3283                 start_workers = 1;
3284         }
3285         mutex_exit(&stmf_state.stmf_lock);

3287         if (start_workers)
3288                 stmf_worker_init();

3290         /*  the default state of LPORT */

3292         if (stmf_state.stmf_default_lport_state == STMF_STATE_OFFLINE) {
3293                 ilport->ilport_prev_state = STMF_STATE_OFFLINE;
3294         } else {
3295                 ilport->ilport_prev_state = STMF_STATE_ONLINE;
3296                 if (stmf_state.stmf_service_running) {
3297                         ssci.st_rflags = 0;
3298                         ssci.st_additional_info = NULL;
3299                         (void) stmf_ctl(STMF_CMD_LPORT_ONLINE, lport, &ssci);
3300                 }
3301         }

3303         /* XXX: Generate event */
3304         return (STMF_SUCCESS);
3305 }
```

*unchanged_portion_omitted_*

```
3523 /*
3524  * Port provider has to make sure that register/deregister session and
3525  * port are serialized calls.
```



```
3526  */
3527 stmf_status_t
3528 stmf_register_scsi_session(stmf_local_port_t *lport, stmf_scsi_session_t *ss)
3529 {
3530         stmf_i_scsi_session_t *iss;
3531         stmf_i_local_port_t *ilport = (stmf_i_local_port_t *)
3532             lport->lport_stmf_private;
3533         uint8_t          lun[8];

3535         /*
3536          * Port state has to be online to register a scsi session. It is
3537          * possible that we started an offline operation and a new SCSI
3538          * session started at the same time (in that case also we are going
3539          * to fail the registeration). But any other state is simply
3540          * a bad port provider implementation.
3541          */
3542         if (ilport->ilport_state != STMF_STATE_ONLINE) {
3543                 if (ilport->ilport_state != STMF_STATE_OFFLINING) {
3544                         stmf_trace(lport->lport_alias, "Port is trying to "
3545                             "register a session while the state is neither "
3546                             "online nor offlining");
3547                 }
3548                 return (STMF_FAILURE);
3549         }
3550         bzero(lun, 8);
3551         iss = (stmf_i_scsi_session_t *)ss->ss_stmf_private;
3552         if ((iss->iss_irport = stmf_irport_register(ss->ss_rport_id)) == NULL) {
3553                 stmf_trace(lport->lport_alias, "Could not register "
3554                     "remote port during session registration");
3555                 return (STMF_FAILURE);
3556         }

3558         iss->iss_flags |= ISS_BEING_CREATED;

3560         if (ss->ss_rport == NULL) {
3561                 iss->iss_flags |= ISS_NULL_TPTID;
3562                 ss->ss_rport = stmf_scsilib_devid_to_remote_port(
3563                     ss->ss_rport_id);
3564                 if (ss->ss_rport == NULL) {
3565                         iss->iss_flags &= ~(ISS_NULL_TPTID | ISS_BEING_CREATED);
3566                         stmf_trace(lport->lport_alias, "Device id to "
3567                             "remote port conversion failed");
3568                         return (STMF_FAILURE);
3569                 }
3570         } else {
3571                 if (!stmf_scsilib_tptid_validate(ss->ss_rport->rport_tptid,
3572                     ss->ss_rport->rport_tptid_sz, NULL)) {
3573                         iss->iss_flags &= ~ISS_BEING_CREATED;
3574                         stmf_trace(lport->lport_alias, "Remote port "
3575                             "transport id validation failed");
3576                         return (STMF_FAILURE);
3577                 }
3578         }

3580         /* sessions use the ilport_lock. No separate lock is required */
3581         iss->iss_lockp = &ilport->ilport_lock;

3583         if (iss->iss_sm != NULL)
3584                 cmn_err(CE_PANIC, "create lun map called with non NULL map");
3585         iss->iss_sm = (stmf_lun_map_t *)kmem_zalloc(sizeof (stmf_lun_map_t),
3586             KM_SLEEP);

3588         mutex_enter(&stmf_state.stmf_lock);
3589         rw_enter(&ilport->ilport_lock, RW_WRITER);
3590         (void) stmf_session_create_lun_map(ilport, iss);
3591         ilport->ilport_nsessions++;
```

```
3592         iss->iss_next = ilport->ilport_ss_list;
3593         ilport->ilport_ss_list = iss;
3594         rw_exit(&ilport->ilport_lock);
3595         mutex_exit(&stmf_state.stmf_lock);

3597         iss->iss_creation_time = ddi_get_time();
3598         ss->ss_session_id = atomic_inc_64_nv(&stmf_session_counter);
3598         ss->ss_session_id = atomic_add_64_nv(&stmf_session_counter, 1);
3599         iss->iss_flags &= ~ISS_BEING_CREATED;
3600         /* XXX should we remove ISS_LUN_INVENTORY_CHANGED on new session? */
3601         iss->iss_flags &= ~ISS_LUN_INVENTORY_CHANGED;
3602         DTRACE_PROBE2(session__online, stmf_local_port_t *, lport,
3603             stmf_scsi_session_t *, ss);
3604         return (STMF_SUCCESS);
3605 }
______unchanged_portion_omitted_

3782 void
3783 stmf_do_itl_dereg(stmf_lu_t *lu, stmf_itl_data_t *itl, uint8_t hdlrm_reason)
3784 {
3785         uint8_t old, new;

3787         do {
3788                 old = new = itl->itl_flags;
3789                 if (old & STMF_ITL_BEING_TERMINATED)
3790                         return;
3791                 new |= STMF_ITL_BEING_TERMINATED;
3792         } while (atomic_cas_8(&itl->itl_flags, old, new) != old);
3793         itl->itl_hdlrm_reason = hdlrm_reason;

3795         ASSERT(itl->itl_counter);

3797         if (atomic_dec_32_nv(&itl->itl_counter))
3797         if (atomic_add_32_nv(&itl->itl_counter, -1))
3798                 return;

3800         stmf_release_itl_handle(lu, itl);
3801 }
______unchanged_portion_omitted_

4014 /* ARGSUSED */
4015 struct scsi_task *
4016 stmf_task_alloc(struct stmf_local_port *lport, stmf_scsi_session_t *ss,
4017                         uint8_t *lun, uint16_t cdb_length_in, uint16_t ext_id)
4018 {
4019         stmf_lu_t *lu;
4020         stmf_i_scsi_session_t *iss;
4021         stmf_i_lu_t *ilu;
4022         stmf_i_scsi_task_t *itask;
4023         stmf_i_scsi_task_t **ppitask;
4024         scsi_task_t *task;
4025         uint8_t *l;
4026         stmf_lun_map_ent_t *lun_map_ent;
4027         uint16_t cdb_length;
4028         uint16_t luNbr;
4029         uint8_t new_task = 0;

4031         /*
4032          * We allocate 7 extra bytes for CDB to provide a cdb pointer which
4033          * is guaranteed to be 8 byte aligned. Some LU providers like OSD
4034          * depend upon this alignment.
4035          */
4036         if (cdb_length_in >= 16)
4037                 cdb_length = cdb_length_in + 7;
4038         else
4039                 cdb_length = 16 + 7;
```



```
4040         iss = (stmf_i_scsi_session_t *)ss->ss_stmf_private;
4041         luNbr = ((uint16_t)lun[1] | (((uint16_t)(lun[0] & 0x3F)) << 8));
4042         rw_enter(iss->iss_lockp, RW_READER);
4043         lun_map_ent =
4044             (stmf_lun_map_ent_t *)stmf_get_ent_from_map(iss->iss_sm, luNbr);
4045         if (!lun_map_ent) {
4046                 lu = dlun0;
4047         } else {
4048                 lu = lun_map_ent->ent_lu;
4049         }
4050         ilu = lu->lu_stmf_private;
4051         if (ilu->ilu_flags & ILU_RESET_ACTIVE) {
4052                 rw_exit(iss->iss_lockp);
4053                 return (NULL);
4054         }
4055         ASSERT(lu == dlun0 || (ilu->ilu_state != STMF_STATE_OFFLINING &&
4056             ilu->ilu_state != STMF_STATE_OFFLINE));
4057         do {
4058                 if (ilu->ilu_free_tasks == NULL) {
4059                         new_task = 1;
4060                         break;
4061                 }
4062                 mutex_enter(&ilu->ilu_task_lock);
4063                 for (ppitask = &ilu->ilu_free_tasks; (*ppitask != NULL) &&
4064                     ((*ppitask)->itask_cdb_buf_size < cdb_length);
4065                     ppitask = &((*ppitask)->itask_lu_free_next))
4066                         ;
4067                 if (*ppitask) {
4068                         itask = *ppitask;
4069                         *ppitask = (*ppitask)->itask_lu_free_next;
4070                         ilu->ilu_ntasks_free--;
4071                         if (ilu->ilu_ntasks_free < ilu->ilu_ntasks_min_free)
4072                                 ilu->ilu_ntasks_min_free = ilu->ilu_ntasks_free;
4073                 } else {
4074                         new_task = 1;
4075                 }
4076                 mutex_exit(&ilu->ilu_task_lock);
4077         /* CONSTCOND */
4078         } while (0);

4080         if (!new_task) {
4081                 /*
4082                  * Save the task_cdb pointer and zero per cmd fields.
4083                  * We know the task_cdb_length is large enough by task
4084                  * selection process above.
4085                  */
4086                 uint8_t *save_cdb;
4087                 uintptr_t t_start, t_end;

4089                 task = itask->itask_task;
4090                 save_cdb = task->task_cdb;      /* save */
4091                 t_start = (uintptr_t)&task->task_flags;
4092                 t_end = (uintptr_t)&task->task_extended_cmd;
4093                 bzero((void *)t_start, (size_t)(t_end - t_start));
4094                 task->task_cdb = save_cdb;      /* restore */
4095                 itask->itask_ncmds = 0;
4096         } else {
4097                 task = (scsi_task_t *)stmf_alloc(STMF_STRUCT_SCSI_TASK,
4098                     cdb_length, AF_FORCE_NOSLEEP);
4099                 if (task == NULL) {
4100                         rw_exit(iss->iss_lockp);
4101                         return (NULL);
4102                 }
4103                 task->task_lu = lu;
4104                 l = task->task_lun_no;
4105                 l[0] = lun[0];
```

```
4106                 l[1] = lun[1];
4107                 l[2] = lun[2];
4108                 l[3] = lun[3];
4109                 l[4] = lun[4];
4110                 l[5] = lun[5];
4111                 l[6] = lun[6];
4112                 l[7] = lun[7];
4113                 task->task_cdb = (uint8_t *)task->task_port_private;
4114                 if ((ulong_t)(task->task_cdb) & 7ul) {
4115                         task->task_cdb = (uint8_t *)(((ulong_t)
4116                             (task->task_cdb) + 7ul) & ~(7ul));
4117                 }
4118                 itask = (stmf_i_scsi_task_t *)task->task_stmf_private;
4119                 itask->itask_cdb_buf_size = cdb_length;
4120                 mutex_init(&itask->itask_audit_mutex, NULL, MUTEX_DRIVER, NULL);
4121         }
4122         task->task_session = ss;
4123         task->task_lport = lport;
4124         task->task_cdb_length = cdb_length_in;
4125         itask->itask_flags = ITASK_IN_TRANSITION;
4126         itask->itask_waitq_time = 0;
4127         itask->itask_lu_read_time = itask->itask_lu_write_time = 0;
4128         itask->itask_lport_read_time = itask->itask_lport_write_time = 0;
4129         itask->itask_read_xfer = itask->itask_write_xfer = 0;
4130         itask->itask_audit_index = 0;

4132         if (new_task) {
4133                 if (lu->lu_task_alloc(task) != STMF_SUCCESS) {
4134                         rw_exit(iss->iss_lockp);
4135                         stmf_free(task);
4136                         return (NULL);
4137                 }
4138                 mutex_enter(&ilu->ilu_task_lock);
4139                 if (ilu->ilu_flags & ILU_RESET_ACTIVE) {
4140                         mutex_exit(&ilu->ilu_task_lock);
4141                         rw_exit(iss->iss_lockp);
4142                         stmf_free(task);
4143                         return (NULL);
4144                 }
4145                 itask->itask_lu_next = ilu->ilu_tasks;
4146                 if (ilu->ilu_tasks)
4147                         ilu->ilu_tasks->itask_lu_prev = itask;
4148                 ilu->ilu_tasks = itask;
4149                 /* kmem_zalloc automatically makes itask->itask_lu_prev NULL */
4150                 ilu->ilu_ntasks++;
4151                 mutex_exit(&ilu->ilu_task_lock);
4152         }

4154         itask->itask_ilu_task_cntr = ilu->ilu_cur_task_cntr;
4155         atomic_inc_32(itask->itask_ilu_task_cntr);
4155         atomic_add_32(itask->itask_ilu_task_cntr, 1);
4156         itask->itask_start_time = ddi_get_lbolt();

4158         if ((lun_map_ent != NULL) && ((itask->itask_itl_datap =
4159             lun_map_ent->ent_itl_datap) != NULL)) {
4160                 atomic_inc_32(&itask->itask_itl_datap->itl_counter);
4160                 atomic_add_32(&itask->itask_itl_datap->itl_counter, 1);
4161                 task->task_lu_itl_handle = itask->itask_itl_datap->itl_handle;
4162         } else {
4163                 itask->itask_itl_datap = NULL;
4164                 task->task_lu_itl_handle = NULL;
4165         }

4167         rw_exit(iss->iss_lockp);
4168         return (task);
4169 }
```



```
4171 static void
4172 stmf_task_lu_free(scsi_task_t *task, stmf_i_scsi_session_t *iss)
4173 {
4174         stmf_i_scsi_task_t *itask =
4175             (stmf_i_scsi_task_t *)task->task_stmf_private;
4176         stmf_i_lu_t *ilu = (stmf_i_lu_t *)task->task_lu->lu_stmf_private;

4178         ASSERT(rw_lock_held(iss->iss_lockp));
4179         itask->itask_flags = ITASK_IN_FREE_LIST;
4180         itask->itask_proxy_msg_id = 0;
4181         mutex_enter(&ilu->ilu_task_lock);
4182         itask->itask_lu_free_next = ilu->ilu_free_tasks;
4183         ilu->ilu_free_tasks = itask;
4184         ilu->ilu_ntasks_free++;
4185         if (ilu->ilu_ntasks == ilu->ilu_ntasks_free)
4186                 cv_signal(&ilu->ilu_offline_pending_cv);
4187         mutex_exit(&ilu->ilu_task_lock);
4188         atomic_dec_32(itask->itask_ilu_task_cntr);
4188         atomic_add_32(itask->itask_ilu_task_cntr, -1);
4189 }
______unchanged_portion_omitted_

4394 void
4395 stmf_task_free(scsi_task_t *task)
4396 {
4397         stmf_local_port_t *lport = task->task_lport;
4398         stmf_i_scsi_task_t *itask = (stmf_i_scsi_task_t *)
4399             task->task_stmf_private;
4400         stmf_i_scsi_session_t *iss = (stmf_i_scsi_session_t *)
4401             task->task_session->ss_stmf_private;

4403         stmf_task_audit(itask, TE_TASK_FREE, CMD_OR_IOF_NA, NULL);

4405         stmf_free_task_bufs(itask, lport);
4406         stmf_itl_task_done(itask);
4407         DTRACE_PROBE2(stmf__task__end, scsi_task_t *, task,
4408             hrtime_t,
4409             itask->itask_done_timestamp - itask->itask_start_timestamp);
4410         if (itask->itask_itl_datap) {
4411                 if (atomic_dec_32_nv(&itask->itask_itl_datap->itl_counter) ==
4412                     0) {
4411                 if (atomic_add_32_nv(&itask->itask_itl_datap->itl_counter,
4412                     -1) == 0) {
4413                         stmf_release_itl_handle(task->task_lu,
4414                             itask->itask_itl_datap);
4415                 }
4416         }

4418         rw_enter(iss->iss_lockp, RW_READER);
4419         lport->lport_task_free(task);
4420         if (itask->itask_worker) {
4421                 atomic_dec_32(&stmf_cur_ntasks);
4422                 atomic_dec_32(&itask->itask_worker->worker_ref_count);
4421                 atomic_add_32(&stmf_cur_ntasks, -1);
4422                 atomic_add_32(&itask->itask_worker->worker_ref_count, -1);
4423         }
4424         /*
4425          * After calling stmf_task_lu_free, the task pointer can no longer
4426          * be trusted.
4427          */
4428         stmf_task_lu_free(task, iss);
4429         rw_exit(iss->iss_lockp);
4430 }

4432 void
```

```
4433 stmf_post_task(scsi_task_t *task, stmf_data_buf_t *dbuf)
4434 {
4435         stmf_i_scsi_task_t *itask = (stmf_i_scsi_task_t *)
4436             task->task_stmf_private;
4437         stmf_i_lu_t *ilu = (stmf_i_lu_t *)task->task_lu->lu_stmf_private;
4438         int nv;
4439         uint32_t old, new;
4440         uint32_t ct;
4441         stmf_worker_t *w, *w1;
4442         uint8_t tm;

4444         if (task->task_max_nbufs > 4)
4445                 task->task_max_nbufs = 4;
4446         task->task_cur_nbufs = 0;
4447         /* Latest value of currently running tasks */
4448         ct = atomic_inc_32_nv(&stmf_cur_ntasks);
4448         ct = atomic_add_32_nv(&stmf_cur_ntasks, 1);

4450         /* Select the next worker using round robin */
4451         nv = (int)atomic_inc_32_nv((uint32_t *)&stmf_worker_sel_counter);
4451         nv = (int)atomic_add_32_nv((uint32_t *)&stmf_worker_sel_counter, 1);
4452         if (nv >= stmf_nworkers_accepting_cmds) {
4453                 int s = nv;
4454                 do {
4455                         nv -= stmf_nworkers_accepting_cmds;
4456                 } while (nv >= stmf_nworkers_accepting_cmds);
4457                 if (nv < 0)
4458                         nv = 0;
4459                 /* Its ok if this cas fails */
4460                 (void) atomic_cas_32((uint32_t *)&stmf_worker_sel_counter,
4461                     s, nv);
4462         }
4463         w = &stmf_workers[nv];

4465         /*
4466          * A worker can be pinned by interrupt. So select the next one
4467          * if it has lower load.
4468          */
4469         if ((nv + 1) >= stmf_nworkers_accepting_cmds) {
4470                 w1 = stmf_workers;
4471         } else {
4472                 w1 = &stmf_workers[nv + 1];
4473         }
4474         if (w1->worker_queue_depth < w->worker_queue_depth)
4475                 w = w1;

4477         mutex_enter(&w->worker_lock);
4478         if (((w->worker_flags & STMF_WORKER_STARTED) == 0) ||
4479             (w->worker_flags & STMF_WORKER_TERMINATE)) {
4480                 /*
4481                  * Maybe we are in the middle of a change. Just go to
4482                  * the 1st worker.
4483                  */
4484                 mutex_exit(&w->worker_lock);
4485                 w = stmf_workers;
4486                 mutex_enter(&w->worker_lock);
4487         }
4488         itask->itask_worker = w;
4489         /*
4490          * Track max system load inside the worker as we already have the
4491          * worker lock (no point implementing another lock). The service
4492          * thread will do the comparisons and figure out the max overall
4493          * system load.
4494          */
4495         if (w->worker_max_sys_qdepth_pu < ct)
4496                 w->worker_max_sys_qdepth_pu = ct;
```



```
4498         do {
4499                 old = new = itask->itask_flags;
4500                 new |= ITASK_KNOWN_TO_TGT_PORT | ITASK_IN_WORKER_QUEUE;
4501                 if (task->task_mgmt_function) {
4502                         tm = task->task_mgmt_function;
4503                         if ((tm == TM_TARGET_RESET) ||
4504                             (tm == TM_TARGET_COLD_RESET) ||
4505                             (tm == TM_TARGET_WARM_RESET)) {
4506                                 new |= ITASK_DEFAULT_HANDLING;
4507                         }
4508                 } else if (task->task_cdb[0] == SCMD_REPORT_LUNS) {
4509                         new |= ITASK_DEFAULT_HANDLING;
4510                 }
4511                 new &= ~ITASK_IN_TRANSITION;
4512         } while (atomic_cas_32(&itask->itask_flags, old, new) != old);

4514         stmf_itl_task_start(itask);

4516         itask->itask_worker_next = NULL;
4517         if (w->worker_task_tail) {
4518                 w->worker_task_tail->itask_worker_next = itask;
4519         } else {
4520                 w->worker_task_head = itask;
4521         }
4522         w->worker_task_tail = itask;
4523         if (++(w->worker_queue_depth) > w->worker_max_qdepth_pu) {
4524                 w->worker_max_qdepth_pu = w->worker_queue_depth;
4525         }
4526         /* Measure task waitq time */
4527         itask->itask_waitq_enter_timestamp = gethrtime();
4528         atomic_inc_32(&w->worker_ref_count);
4528         atomic_add_32(&w->worker_ref_count, 1);
4529         itask->itask_cmd_stack[0] = ITASK_CMD_NEW_TASK;
4530         itask->itask_ncmds = 1;
4531         stmf_task_audit(itask, TE_TASK_START, CMD_OR_IOF_NA, dbuf);
4532         if (dbuf) {
4533                 itask->itask_allocated_buf_map = 1;
4534                 itask->itask_dbufs[0] = dbuf;
4535                 dbuf->db_handle = 0;
4536         } else {
4537                 itask->itask_allocated_buf_map = 0;
4538                 itask->itask_dbufs[0] = NULL;
4539         }

4541         if ((w->worker_flags & STMF_WORKER_ACTIVE) == 0) {
4542                 w->worker_signal_timestamp = gethrtime();
4543                 DTRACE_PROBE2(worker__signal, stmf_worker_t *, w,
4544                     scsi_task_t *, task);
4545                 cv_signal(&w->worker_cv);
4546         }
4547         mutex_exit(&w->worker_lock);

4549         /*
4550          * This can only happen if during stmf_task_alloc(), ILU_RESET_ACTIVE
4551          * was set between checking of ILU_RESET_ACTIVE and clearing of the
4552          * ITASK_IN_FREE_LIST flag. Take care of these "sneaked-in" tasks here.
4553          */
4554         if (ilu->ilu_flags & ILU_RESET_ACTIVE) {
4555                 stmf_abort(STMF_QUEUE_TASK_ABORT, task, STMF_ABORTED, NULL);
4556         }
4557 }
______unchanged_portion_omitted_


4577 /*
```

```
 * +++++++++++++++ ABORT LOGIC +++++++++++++++++++++
 * Once ITASK_BEING_ABORTED is set, ITASK_KNOWN_TO_LU can be reset already
 * i.e. before ITASK_BEING_ABORTED being set. But if it was not, it cannot
 * be reset until the LU explicitly calls stmf_task_lu_aborted(). Of course
 * the LU will make this call only if we call the LU's abort entry point.
 * we will only call that entry point if ITASK_KNOWN_TO_LU was set.
 *
 * Same logic applies for the port.
 *
 * Also ITASK_BEING_ABORTED will not be allowed to set if both KNOWN_TO_LU
 * and KNOWN_TO_TGT_PORT are reset.
 *
 * +++++++++++++++++++++++++++++++++++++++++++++
 */

stmf_status_t
stmf_xfer_data(scsi_task_t *task, stmf_data_buf_t *dbuf, uint32_t ioflags)
{
        stmf_status_t ret = STMF_SUCCESS;

        stmf_i_scsi_task_t *itask =
            (stmf_i_scsi_task_t *)task->task_stmf_private;

        stmf_task_audit(itask, TE_XFER_START, ioflags, dbuf);

        if (ioflags & STMF_IOF_LU_DONE) {
                uint32_t new, old;
                do {
                        new = old = itask->itask_flags;
                        if (new & ITASK_BEING_ABORTED)
                                return (STMF_ABORTED);
                        new &= ~ITASK_KNOWN_TO_LU;
                } while (atomic_cas_32(&itask->itask_flags, old, new) != old);
        }
        if (itask->itask_flags & ITASK_BEING_ABORTED)
                return (STMF_ABORTED);
#ifdef  DEBUG
        if (!(ioflags & STMF_IOF_STATS_ONLY) && stmf_drop_buf_counter > 0) {
                if (atomic_dec_32_nv((uint32_t *)&stmf_drop_buf_counter) ==
                atomic_add_32_nv((uint32_t *)&stmf_drop_buf_counter, -1) ==
                    1)
                        return (STMF_SUCCESS);
        }
#endif

        stmf_update_kstat_lu_io(task, dbuf);
        stmf_update_kstat_lport_io(task, dbuf);
        stmf_lport_xfer_start(itask, dbuf);
        if (ioflags & STMF_IOF_STATS_ONLY) {
                stmf_lport_xfer_done(itask, dbuf);
                return (STMF_SUCCESS);
        }

        dbuf->db_flags |= DB_LPORT_XFER_ACTIVE;
        ret = task->task_lport->lport_xfer_data(task, dbuf, ioflags);

        /*
         * Port provider may have already called the buffer callback in
         * which case dbuf->db_xfer_start_timestamp will be 0.
         */
        if (ret != STMF_SUCCESS) {
                dbuf->db_flags &= ~DB_LPORT_XFER_ACTIVE;
                if (dbuf->db_xfer_start_timestamp != 0)
                        stmf_lport_xfer_done(itask, dbuf);
        }
```



```
        return (ret);
}
______unchanged_portion_omitted_

stmf_status_t
stmf_scsilib_uniq_lu_id2(uint32_t company_id, uint32_t host_id,
    scsi_devid_desc_t *lu_id)
{
        uint8_t *p;
        struct timeval32 timestamp32;
        uint32_t *t = (uint32_t *)&timestamp32;
        struct ether_addr mac;
        uint8_t *e = (uint8_t *)&mac;
        int hid = (int)host_id;
        uint16_t gen_number;

        if (company_id == COMPANY_ID_NONE)
                company_id = COMPANY_ID_SUN;

        if (lu_id->ident_length != 0x10)
                return (STMF_INVALID_ARG);

        p = (uint8_t *)lu_id;

        gen_number = atomic_inc_16_nv(&stmf_lu_id_gen_number);
        gen_number = atomic_add_16_nv(&stmf_lu_id_gen_number, 1);

        p[0] = 0xf1; p[1] = 3; p[2] = 0; p[3] = 0x10;
        p[4] = ((company_id >> 20) & 0xf) | 0x60;
        p[5] = (company_id >> 12) & 0xff;
        p[6] = (company_id >> 4) & 0xff;
        p[7] = (company_id << 4) & 0xf0;
        if (hid == 0 && !localetheraddr((struct ether_addr *)NULL, &mac)) {
                hid = BE_32((int)zone_get_hostid(NULL));
        }
        if (hid != 0) {
                e[0] = (hid >> 24) & 0xff;
                e[1] = (hid >> 16) & 0xff;
                e[2] = (hid >> 8) & 0xff;
                e[3] = hid & 0xff;
                e[4] = e[5] = 0;
        }
        bcopy(e, p+8, 6);
        uniqtime32(&timestamp32);
        *t = BE_32(*t);
        bcopy(t, p+14, 4);
        p[18] = (gen_number >> 8) & 0xff;
        p[19] = gen_number & 0xff;

        return (STMF_SUCCESS);
}
______unchanged_portion_omitted_

void
stmf_worker_task(void *arg)
{
        stmf_worker_t *w;
        stmf_i_scsi_session_t *iss;
        scsi_task_t *task;
        stmf_i_scsi_task_t *itask;
        stmf_data_buf_t *dbuf;
        stmf_lu_t *lu;
        clock_t wait_timer = 0;
        clock_t wait_ticks, wait_delta = 0;
        uint32_t old, new;
        uint8_t curcmd;
```

```
6197         uint8_t abort_free;
6198         uint8_t wait_queue;
6199         uint8_t dec_qdepth;

6201         w = (stmf_worker_t *)arg;
6202         wait_ticks = drv_usectohz(10000);

6204         DTRACE_PROBE1(worker__create, stmf_worker_t, w);
6205         mutex_enter(&w->worker_lock);
6206         w->worker_flags |= STMF_WORKER_STARTED | STMF_WORKER_ACTIVE;
6207 stmf_worker_loop:;
6208         if ((w->worker_ref_count == 0) &&
6209             (w->worker_flags & STMF_WORKER_TERMINATE)) {
6210                 w->worker_flags &= ~(STMF_WORKER_STARTED |
6211                     STMF_WORKER_ACTIVE | STMF_WORKER_TERMINATE);
6212                 w->worker_tid = NULL;
6213                 mutex_exit(&w->worker_lock);
6214                 DTRACE_PROBE1(worker__destroy, stmf_worker_t, w);
6215                 thread_exit();
6216         }
6217         /* CONSTCOND */
6218         while (1) {
6219                 dec_qdepth = 0;
6220                 if (wait_timer && (ddi_get_lbolt() >= wait_timer)) {
6221                         wait_timer = 0;
6222                         wait_delta = 0;
6223                         if (w->worker_wait_head) {
6224                                 ASSERT(w->worker_wait_tail);
6225                                 if (w->worker_task_head == NULL)
6226                                         w->worker_task_head =
6227                                             w->worker_wait_head;
6228                                 else
6229                                         w->worker_task_tail->itask_worker_next =
6230                                             w->worker_wait_head;
6231                                 w->worker_task_tail = w->worker_wait_tail;
6232                                 w->worker_wait_head = w->worker_wait_tail =
6233                                     NULL;
6234                         }
6235                 }
6236                 if ((itask = w->worker_task_head) == NULL) {
6237                         break;
6238                 }
6239                 task = itask->itask_task;
6240                 DTRACE_PROBE2(worker__active, stmf_worker_t, w,
6241                     scsi_task_t *, task);
6242                 w->worker_task_head = itask->itask_worker_next;
6243                 if (w->worker_task_head == NULL)
6244                         w->worker_task_tail = NULL;

6246                 wait_queue = 0;
6247                 abort_free = 0;
6248                 if (itask->itask_ncmds > 0) {
6249                         curcmd = itask->itask_cmd_stack[itask->itask_ncmds - 1];
6250                 } else {
6251                         ASSERT(itask->itask_flags & ITASK_BEING_ABORTED);
6252                 }
6253                 do {
6254                         old = itask->itask_flags;
6255                         if (old & ITASK_BEING_ABORTED) {
6256                                 itask->itask_ncmds = 1;
6257                                 curcmd = itask->itask_cmd_stack[0] =
6258                                     ITASK_CMD_ABORT;
6259                                 goto out_itask_flag_loop;
6260                         } else if ((curcmd & ITASK_CMD_MASK) ==
6261                             ITASK_CMD_NEW_TASK) {
6262                                 /*
```



```
6263                                  * set ITASK_KSTAT_IN_RUNQ, this flag
6264                                  * will not reset until task completed
6265                                  */
6266                                 new = old | ITASK_KNOWN_TO_LU |
6267                                     ITASK_KSTAT_IN_RUNQ;
6268                         } else {
6269                                 goto out_itask_flag_loop;
6270                         }
6271                 } while (atomic_cas_32(&itask->itask_flags, old, new) != old);

6273 out_itask_flag_loop:

6275                 /*
6276                  * Decide if this task needs to go to a queue and/or if
6277                  * we can decrement the itask_cmd_stack.
6278                  */
6279                 if (curcmd == ITASK_CMD_ABORT) {
6280                         if (itask->itask_flags & (ITASK_KNOWN_TO_LU |
6281                             ITASK_KNOWN_TO_TGT_PORT)) {
6282                                 wait_queue = 1;
6283                         } else {
6284                                 abort_free = 1;
6285                         }
6286                 } else if ((curcmd & ITASK_CMD_POLL) &&
6287                     (itask->itask_poll_timeout > ddi_get_lbolt())) {
6288                         wait_queue = 1;
6289                 }

6291                 if (wait_queue) {
6292                         itask->itask_worker_next = NULL;
6293                         if (w->worker_wait_tail) {
6294                                 w->worker_wait_tail->itask_worker_next = itask;
6295                         } else {
6296                                 w->worker_wait_head = itask;
6297                         }
6298                         w->worker_wait_tail = itask;
6299                         if (wait_timer == 0) {
6300                                 wait_timer = ddi_get_lbolt() + wait_ticks;
6301                                 wait_delta = wait_ticks;
6302                         }
6303                 } else if ((--(itask->itask_ncmds)) != 0) {
6304                         itask->itask_worker_next = NULL;
6305                         if (w->worker_task_tail) {
6306                                 w->worker_task_tail->itask_worker_next = itask;
6307                         } else {
6308                                 w->worker_task_head = itask;
6309                         }
6310                         w->worker_task_tail = itask;
6311                 } else {
6312                         atomic_and_32(&itask->itask_flags,
6313                             ~ITASK_IN_WORKER_QUEUE);
6314                         /*
6315                          * This is where the queue depth should go down by
6316                          * one but we delay that on purpose to account for
6317                          * the call into the provider. The actual decrement
6318                          * happens after the worker has done its job.
6319                          */
6320                         dec_qdepth = 1;
6321                         itask->itask_waitq_time +=
6322                             gethrtime() - itask->itask_waitq_enter_timestamp;
6323                 }

6325                 /* We made it here means we are going to call LU */
6326                 if ((itask->itask_flags & ITASK_DEFAULT_HANDLING) == 0)
6327                         lu = task->task_lu;
6328                 else
```

```
6329                        lu = dlun0;
6330                dbuf = itask->itask_dbufs[ITASK_CMD_BUF_NDX(curcmd)];
6331                mutex_exit(&w->worker_lock);
6332                curcmd &= ITASK_CMD_MASK;
6333                stmf_task_audit(itask, TE_PROCESS_CMD, curcmd, dbuf);
6334                switch (curcmd) {
6335                case ITASK_CMD_NEW_TASK:
6336                        iss = (stmf_i_scsi_session_t *)
6337                            task->task_session->ss_stmf_private;
6338                        stmf_itl_lu_new_task(itask);
6339                        if (iss->iss_flags & ISS_LUN_INVENTORY_CHANGED) {
6340                                if (stmf_handle_cmd_during_ic(itask))
6341                                        break;
6342                        }
6343 #ifdef  DEBUG
6344                        if (stmf_drop_task_counter > 0) {
6345                                if (atomic_dec_32_nv((uint32_t *)&stmf_drop_task
6346                                    1) {
6345                                if (atomic_add_32_nv(
6346                                    (uint32_t *)&stmf_drop_task_counter,
6347                                    -1) == 1) {
6347                                        break;
6348                                }
6349                        }
6350 #endif
6351                        DTRACE_PROBE1(scsi__task__start, scsi_task_t *, task);
6352                        lu->lu_new_task(task, dbuf);
6353                        break;
6354                case ITASK_CMD_DATA_XFER_DONE:
6355                        lu->lu_dbuf_xfer_done(task, dbuf);
6356                        break;
6357                case ITASK_CMD_STATUS_DONE:
6358                        lu->lu_send_status_done(task);
6359                        break;
6360                case ITASK_CMD_ABORT:
6361                        if (abort_free) {
6362                                stmf_task_free(task);
6363                        } else {
6364                                stmf_do_task_abort(task);
6365                        }
6366                        break;
6367                case ITASK_CMD_POLL_LU:
6368                        if (!wait_queue) {
6369                                lu->lu_task_poll(task);
6370                        }
6371                        break;
6372                case ITASK_CMD_POLL_LPORT:
6373                        if (!wait_queue)
6374                                task->task_lport->lport_task_poll(task);
6375                        break;
6376                case ITASK_CMD_SEND_STATUS:
6377                /* case ITASK_CMD_XFER_DATA: */
6378                        break;
6379                }
6380                mutex_enter(&w->worker_lock);
6381                if (dec_qdepth) {
6382                        w->worker_queue_depth--;
6383                }
6384        }
6385        if ((w->worker_flags & STMF_WORKER_TERMINATE) && (wait_timer == 0)) {
6386                if (w->worker_ref_count == 0)
6387                        goto stmf_worker_loop;
6388                else {
6389                        wait_timer = ddi_get_lbolt() + 1;
6390                        wait_delta = 1;
6391                }
```



```
6392        }
6393        w->worker_flags &= ~STMF_WORKER_ACTIVE;
6394        if (wait_timer) {
6395                DTRACE_PROBE1(worker__timed__sleep, stmf_worker_t, w);
6396                (void) cv_reltimedwait(&w->worker_cv, &w->worker_lock,
6397                    wait_delta, TR_CLOCK_TICK);
6398        } else {
6399                DTRACE_PROBE1(worker__sleep, stmf_worker_t, w);
6400                cv_wait(&w->worker_cv, &w->worker_lock);
6401        }
6402        DTRACE_PROBE1(worker__wakeup, stmf_worker_t, w);
6403        w->worker_flags |= STMF_WORKER_ACTIVE;
6404        goto stmf_worker_loop;
6405 }
```

*unchanged_portion_omitted_*

****
14138 Mon Jul 28 07:44:34 2014
new/usr/src/uts/common/io/cxgbe/t4nex/t4_l2t.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
 396 /*
 397  * Called when an L2T entry has no more users.  The entry is left in the hash
 398  * table since it is likely to be reused but we also bump nfree to indicate
 399  * that the entry can be reallocated for a different neighbor.  We also drop
 400  * the existing neighbor reference in case the neighbor is going away and is
 401  * waiting on our reference.
 402  *
 403  * Because entries can be reallocated to other neighbors once their ref count
 404  * drops to 0 we need to take the entry's lock to avoid races with a new
 405  * incarnation.
 406  */
 407 static void
 408 t4_l2e_free(struct l2t_entry *e)
 409 {
 410         struct l2t_data *d;

 412         mutex_enter(&e->lock);
 413         /* LINTED: E_NOP_IF_STMT */
 414         if (atomic_read(&e->refcnt) == 0) {  /* hasn't been recycled */
 415                 /*
 416                  * Don't need to worry about the arpq, an L2T entry can't be
 417                  * released if any packets are waiting for resolution as we
 418                  * need to be able to communicate with the device to close a
 419                  * connection.
 420                  */
 421         }
 422         mutex_exit(&e->lock);

 424         d = container_of(e, struct l2t_data, l2tab[e->idx]);
 425         atomic_inc_uint(&d->nfree);
 425         atomic_add_int(&d->nfree, 1);

 427 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   54720 Mon Jul 28 07:44:34 2014
new/usr/src/uts/common/io/dld/dld_str.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 595 /*
 596  * Create a new dld_str_t object.
 597  */
 598 dld_str_t *
 599 dld_str_create(queue_t *rq, uint_t type, major_t major, t_uscalar_t style)
 600 {
 601         dld_str_t       *dsp;
 602         int             err;

 604         /*
 605          * Allocate an object from the cache.
 606          */
 607         atomic_inc_32(&str_count);
 607         atomic_add_32(&str_count, 1);
 608         dsp = kmem_cache_alloc(str_cachep, KM_SLEEP);

 610         /*
 611          * Allocate the dummy mblk for flow-control.
 612          */
 613         dsp->ds_tx_flow_mp = allocb(1, BPRI_HI);
 614         if (dsp->ds_tx_flow_mp == NULL) {
 615                 kmem_cache_free(str_cachep, dsp);
 616                 atomic_dec_32(&str_count);
 616                 atomic_add_32(&str_count, -1);
 617                 return (NULL);
 618         }
 619         dsp->ds_type = type;
 620         dsp->ds_major = major;
 621         dsp->ds_style = style;

 623         /*
 624          * Initialize the queue pointers.
 625          */
 626         ASSERT(RD(rq) == rq);
 627         dsp->ds_rq = rq;
 628         dsp->ds_wq = WR(rq);
 629         rq->q_ptr = WR(rq)->q_ptr = (void *)dsp;

 631         /*
 632          * We want explicit control over our write-side STREAMS queue
 633          * where the dummy mblk gets added/removed for flow-control.
 634          */
 635         noenable(WR(rq));

 637         err = mod_hash_insert(str_hashp, STR_HASH_KEY(dsp->ds_minor),
 638             (mod_hash_val_t)dsp);
 639         ASSERT(err == 0);
 640         return (dsp);
 641 }

 643 /*
 644  * Destroy a dld_str_t object.
 645  */
 646 void
 647 dld_str_destroy(dld_str_t *dsp)
 648 {
 649         queue_t         *rq;
 650         queue_t         *wq;
 651         mod_hash_val_t  val;
```



```
 653         /*
 654          * Clear the queue pointers.
 655          */
 656         rq = dsp->ds_rq;
 657         wq = dsp->ds_wq;
 658         ASSERT(wq == WR(rq));
 659         rq->q_ptr = wq->q_ptr = NULL;
 660         dsp->ds_rq = dsp->ds_wq = NULL;

 662         ASSERT(dsp->ds_dlstate == DL_UNATTACHED);
 663         ASSERT(dsp->ds_sap == 0);
 664         ASSERT(dsp->ds_mh == NULL);
 665         ASSERT(dsp->ds_mch == NULL);
 666         ASSERT(dsp->ds_promisc == 0);
 667         ASSERT(dsp->ds_mph == NULL);
 668         ASSERT(dsp->ds_mip == NULL);
 669         ASSERT(dsp->ds_mnh == NULL);

 671         ASSERT(dsp->ds_polling == B_FALSE);
 672         ASSERT(dsp->ds_direct == B_FALSE);
 673         ASSERT(dsp->ds_lso == B_FALSE);
 674         ASSERT(dsp->ds_lso_max == 0);
 675         ASSERT(dsp->ds_passivestate != DLD_ACTIVE);

 677         /*
 678          * Reinitialize all the flags.
 679          */
 680         dsp->ds_notifications = 0;
 681         dsp->ds_passivestate = DLD_UNINITIALIZED;
 682         dsp->ds_mode = DLD_UNITDATA;
 683         dsp->ds_native = B_FALSE;
 684         dsp->ds_nonip = B_FALSE;

 686         ASSERT(dsp->ds_datathr_cnt == 0);
 687         ASSERT(dsp->ds_pending_head == NULL);
 688         ASSERT(dsp->ds_pending_tail == NULL);
 689         ASSERT(!dsp->ds_dlpi_pending);

 691         ASSERT(dsp->ds_dlp == NULL);
 692         ASSERT(dsp->ds_dmap == NULL);
 693         ASSERT(dsp->ds_rx == NULL);
 694         ASSERT(dsp->ds_rx_arg == NULL);
 695         ASSERT(dsp->ds_next == NULL);
 696         ASSERT(dsp->ds_head == NULL);

 698         /*
 699          * Free the dummy mblk if exists.
 700          */
 701         if (dsp->ds_tx_flow_mp != NULL) {
 702                 freeb(dsp->ds_tx_flow_mp);
 703                 dsp->ds_tx_flow_mp = NULL;
 704         }

 706         (void) mod_hash_remove(str_hashp, STR_HASH_KEY(dsp->ds_minor), &val);
 707         ASSERT(dsp == (dld_str_t *)val);

 709         /*
 710          * Free the object back to the cache.
 711          */
 712         kmem_cache_free(str_cachep, dsp);
 713         atomic_dec_32(&str_count);
 713         atomic_add_32(&str_count, -1);
 714 }
______unchanged_portion_omitted_
```

```
**********************************************************
   23959 Mon Jul 28 07:44:34 2014
new/usr/src/uts/common/io/dls/dls_link.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 332 /* ARGSUSED */
 333 void
 334 i_dls_link_rx(void *arg, mac_resource_handle_t mrh, mblk_t *mp,
 335     boolean_t loopback)
 336 {
 337         dls_link_t                      *dlp = arg;
 338         mod_hash_t                      *hash = dlp->dl_str_hash;
 339         mblk_t                          *nextp;
 340         mac_header_info_t               mhi;
 341         dls_head_t                      *dhp;
 342         dld_str_t                       *dsp;
 343         dld_str_t                       *ndsp;
 344         mblk_t                          *nmp;
 345         mod_hash_key_t                  key;
 346         uint_t                          npacket;
 347         boolean_t                       accepted;
 348         dls_rx_t                        ds_rx, nds_rx;
 349         void                            *ds_rx_arg, *nds_rx_arg;
 350         uint16_t                        vid;
 351         int                             err, rval;

 353         /*
 354          * Walk the packet chain.
 355          */
 356         for (; mp != NULL; mp = nextp) {
 357                 /*
 358                  * Wipe the accepted state.
 359                  */
 360                 accepted = B_FALSE;

 362                 DLS_PREPARE_PKT(dlp->dl_mh, mp, &mhi, err);
 363                 if (err != 0) {
 364                         atomic_inc_32(&(dlp->dl_unknowns));
 364                         atomic_add_32(&(dlp->dl_unknowns), 1);
 365                         nextp = mp->b_next;
 366                         mp->b_next = NULL;
 367                         freemsg(mp);
 368                         continue;
 369                 }

 371                 /*
 372                  * Grab the longest sub-chain we can process as a single
 373                  * unit.
 374                  */
 375                 nextp = i_dls_link_subchain(dlp, mp, &mhi, &npacket);
 376                 ASSERT(npacket != 0);

 378                 vid = VLAN_ID(mhi.mhi_tci);

 380                 if (mhi.mhi_istagged) {
 381                         /*
 382                          * If it is tagged traffic, send it upstream to
 383                          * all dld_str_t which are attached to the physical
 384                          * link and bound to SAP 0x8100.
 385                          */
 386                         if (i_dls_link_rx_func(dlp, mrh, &mhi, mp,
 387                             ETHERTYPE_VLAN, dls_accept) > 0) {
 388                                 accepted = B_TRUE;
 389                         }
```



```
 391                         /*
 392                          * Don't pass the packets up if they are tagged
 393                          * packets and:
 394                          *  - their VID and priority are both zero and the
 395                          *    original packet isn't using the PVID (invalid
 396                          *    packets).
 397                          *  - their sap is ETHERTYPE_VLAN and their VID is
 398                          *    zero as they have already been sent upstreams.
 399                          */
 400                         if ((vid == VLAN_ID_NONE && !mhi.mhi_ispvid &&
 401                             VLAN_PRI(mhi.mhi_tci) == 0) ||
 402                             (mhi.mhi_bindsap == ETHERTYPE_VLAN &&
 403                             vid == VLAN_ID_NONE)) {
 404                                 freemsgchain(mp);
 405                                 goto loop;
 406                         }
 407                 }

 409                 /*
 410                  * Construct a hash key from the VLAN identifier and the
 411                  * DLSAP.
 412                  */
 413                 key = MAKE_KEY(mhi.mhi_bindsap);

 415                 /*
 416                  * Search the has table for dld_str_t eligible to receive
 417                  * a packet chain for this DLSAP/VLAN combination.
 418                  */
 419                 if (mod_hash_find_cb_rval(hash, key, (mod_hash_val_t *)&dhp,
 420                     i_dls_head_hold, &rval) != 0 || (rval != 0)) {
 421                         freemsgchain(mp);
 422                         goto loop;
 423                 }

 425                 /*
 426                  * Find the first dld_str_t that will accept the sub-chain.
 427                  */
 428                 for (dsp = dhp->dh_list; dsp != NULL; dsp = dsp->ds_next)
 429                         if (dls_accept(dsp, &mhi, &ds_rx, &ds_rx_arg))
 430                                 break;

 432                 /*
 433                  * If we did not find any dld_str_t willing to accept the
 434                  * sub-chain then throw it away.
 435                  */
 436                 if (dsp == NULL) {
 437                         i_dls_head_rele(dhp);
 438                         freemsgchain(mp);
 439                         goto loop;
 440                 }

 442                 /*
 443                  * We have at least one acceptor.
 444                  */
 445                 accepted = B_TRUE;
 446                 for (;;) {
 447                         /*
 448                          * Find the next dld_str_t that will accept the
 449                          * sub-chain.
 450                          */
 451                         for (ndsp = dsp->ds_next; ndsp != NULL;
 452                             ndsp = ndsp->ds_next)
 453                                 if (dls_accept(ndsp, &mhi, &nds_rx,
 454                                     &nds_rx_arg))
 455                                         break;
```

```
457                         /*
458                          * If there are no more dld_str_t that are willing
459                          * to accept the sub-chain then we don't need to dup
460                          * it before handing it to the current one.
461                          */
462                         if (ndsp == NULL) {
463                                 ds_rx(ds_rx_arg, mrh, mp, &mhi);
464
465                                 /*
466                                  * Since there are no more dld_str_t, we're
467                                  * done.
468                                  */
469                                 break;
470                         }
471
472                         /*
473                          * There are more dld_str_t so dup the sub-chain.
474                          */
475                         if ((nmp = copymsgchain(mp)) != NULL)
476                                 ds_rx(ds_rx_arg, mrh, nmp, &mhi);
477
478                         dsp = ndsp;
479                         ds_rx = nds_rx;
480                         ds_rx_arg = nds_rx_arg;
481                 }
482
483                 /*
484                  * Release the hold on the dld_str_t chain now that we have
485                  * finished walking it.
486                  */
487                 i_dls_head_rele(dhp);
488
489 loop:
490                 /*
491                  * If there were no acceptors then add the packet count to the
492                  * 'unknown' count.
493                  */
494                 if (!accepted)
495                         atomic_add_32(&(dlp->dl_unknowns), npacket);
496         }
497 }
498
499 /* ARGSUSED */
500 void
501 dls_rx_vlan_promisc(void *arg, mac_resource_handle_t mrh, mblk_t *mp,
502     boolean_t loopback)
503 {
504         dld_str_t                       *dsp = arg;
505         dls_link_t                      *dlp = dsp->ds_dlp;
506         mac_header_info_t               mhi;
507         dls_rx_t                        ds_rx;
508         void                            *ds_rx_arg;
509         int                             err;
510
511         DLS_PREPARE_PKT(dlp->dl_mh, mp, &mhi, err);
512         if (err != 0)
513                 goto drop;
514
515         /*
516          * If there is promiscuous handle for vlan, we filter out the untagged
517          * pkts and pkts that are not for the primary unicast address.
518          */
519         if (dsp->ds_vlan_mph != NULL) {
520                 uint8_t prim_addr[MAXMACADDRLEN];
521                 size_t  addr_length = dsp->ds_mip->mi_addr_length;
```



```
522
523                 if (!(mhi.mhi_istagged))
524                         goto drop;
525                 ASSERT(dsp->ds_mh != NULL);
526                 mac_unicast_primary_get(dsp->ds_mh, (uint8_t *)prim_addr);
527                 if (memcmp(mhi.mhi_daddr, prim_addr, addr_length) != 0)
528                         goto drop;
529
530                 if (!dls_accept(dsp, &mhi, &ds_rx, &ds_rx_arg))
531                         goto drop;
532
533                 ds_rx(ds_rx_arg, NULL, mp, &mhi);
534                 return;
535         }
536
537 drop:
538         atomic_inc_32(&dlp->dl_unknowns);
538         atomic_add_32(&dlp->dl_unknowns, 1);
539         freemsg(mp);
540 }
541
542 /* ARGSUSED */
543 void
544 dls_rx_promisc(void *arg, mac_resource_handle_t mrh, mblk_t *mp,
545     boolean_t loopback)
546 {
547         dld_str_t                       *dsp = arg;
548         dls_link_t                      *dlp = dsp->ds_dlp;
549         mac_header_info_t               mhi;
550         dls_rx_t                        ds_rx;
551         void                            *ds_rx_arg;
552         int                             err;
553         dls_head_t                      *dhp;
554         mod_hash_key_t                  key;
555
556         DLS_PREPARE_PKT(dlp->dl_mh, mp, &mhi, err);
557         if (err != 0)
558                 goto drop;
559
560         /*
561          * In order to filter out sap pkt that no dls channel listens, search
562          * the hash table trying to find a dld_str_t eligible to receive the pkt
563          */
564         if ((dsp->ds_promisc & DLS_PROMISC_SAP) == 0) {
565                 key = MAKE_KEY(mhi.mhi_bindsap);
566                 if (mod_hash_find(dsp->ds_dlp->dl_str_hash, key,
567                     (mod_hash_val_t *)&dhp) != 0)
568                         goto drop;
569         }
570
571         if (!dls_accept_promisc(dsp, &mhi, &ds_rx, &ds_rx_arg, loopback))
572                 goto drop;
573
574         ds_rx(ds_rx_arg, NULL, mp, &mhi);
575         return;
576
577 drop:
578         atomic_inc_32(&dlp->dl_unknowns);
578         atomic_add_32(&dlp->dl_unknowns, 1);
579         freemsg(mp);
580 }
______unchanged_portion_omitted_

696 /*
697  * Exported functions.
698  */
```

```
700 static int
701 dls_link_hold_common(const char *name, dls_link_t **dlpp, boolean_t create)
702 {
703         dls_link_t              *dlp;
704         int                     err;

706         /*
707          * Look up a dls_link_t corresponding to the given macname in the
708          * global hash table. The i_dls_link_hash itself is protected by the
709          * mod_hash package's internal lock which synchronizes
710          * find/insert/remove into the global mod_hash list. Assumes that
711          * inserts and removes are single threaded on a per mac end point
712          * by the mac perimeter.
713          */
714         if ((err = mod_hash_find(i_dls_link_hash, (mod_hash_key_t)name,
715             (mod_hash_val_t *)&dlp)) == 0)
716                 goto done;

718         ASSERT(err == MH_ERR_NOTFOUND);
719         if (!create)
720                 return (ENOENT);

722         /*
723          * We didn't find anything so we need to create one.
724          */
725         if ((err = i_dls_link_create(name, &dlp)) != 0)
726                 return (err);

728         /*
729          * Insert the dls_link_t.
730          */
731         err = mod_hash_insert(i_dls_link_hash, (mod_hash_key_t)dlp->dl_name,
732             (mod_hash_val_t)dlp);
733         ASSERT(err == 0);

735         atomic_inc_32(&i_dls_link_count);
735         atomic_add_32(&i_dls_link_count, 1);
736         ASSERT(i_dls_link_count != 0);

738 done:
739         ASSERT(MAC_PERIM_HELD(dlp->dl_mh));
740         /*
741          * Bump the reference count and hand back the reference.
742          */
743         dlp->dl_ref++;
744         *dlpp = dlp;
745         return (0);
746 }
```

_____***unchanged_portion_omitted_***

```
803 void
804 dls_link_rele(dls_link_t *dlp)
805 {
806         mod_hash_val_t  val;

808         ASSERT(MAC_PERIM_HELD(dlp->dl_mh));
809         /*
810          * Check if there are any more references.
811          */
812         if (--dlp->dl_ref == 0) {
813                 (void) mod_hash_remove(i_dls_link_hash,
814                     (mod_hash_key_t)dlp->dl_name, &val);
815                 ASSERT(dlp == (dls_link_t *)val);

817                 /*
```



```
818                  * Destroy the dls_link_t.
819                  */
820                 i_dls_link_destroy(dlp);
821                 ASSERT(i_dls_link_count > 0);
822                 atomic_dec_32(&i_dls_link_count);
822                 atomic_add_32(&i_dls_link_count, -1);
823         }
824 }
```

_____***unchanged_portion_omitted_***

****
2896 Mon Jul 28 07:44:34 2014
new/usr/src/uts/common/io/drm/drm_atomic.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****

```
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * \file drm_atomic.h
 * Atomic operations used in the DRM which may or may not be provided by the OS.
 *
 * \author Eric Anholt <anholt@FreeBSD.org>
 */

/*
 * Copyright 2004 Eric Anholt
 * Copyright (c) 2009, Intel Corporation.
 * All Rights Reserved.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and associated documentation files (the "Software"),
 * to deal in the Software without restriction, including without limitation
 * the rights to use, copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit persons to whom the
 * Software is furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice (including the next
 * paragraph) shall be included in all copies or substantial portions of the
 * Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL
 * VA LINUX SYSTEMS AND/OR ITS SUPPLIERS BE LIABLE FOR ANY CLAIM, DAMAGES OR
 * OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
 * ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 */

/* Many of these implementations are rather fake, but good enough. */



#ifndef _SYS_DRM_ATOMIC_H_
#define _SYS_DRM_ATOMIC_H_

#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/atomic.h>

#ifdef __LINT__
#undef inline
#define inline
#endif
typedef uint32_t        atomic_t;

#define atomic_set(p, v)        (*(p) = (v))
#define atomic_read(p)          (*(p))
#define atomic_inc(p)           atomic_inc_uint(p)
#define atomic_inc(p)           atomic_add_int(p, 1)
#define atomic_dec(p)           atomic_dec_uint(p)
#define atomic_add(n, p)        atomic_add_int(p, n)
```



```
#define atomic_sub(n, p)        atomic_add_int(p, -n)
#define atomic_set_int(p, bits) atomic_or_uint(p, bits)
#define atomic_clear_int(p, bits)       atomic_and_uint(p, ~(bits))
#define atomic_cmpset_int(p, c, n) \
        ((c == atomic_cas_uint(p, c, n)) ? 1 : 0)

#define set_bit(b, p) \
        atomic_set_int(((volatile uint_t *)(void *)p) + (b >> 5), \
        1 << (b & 0x1f))

#define clear_bit(b, p) \
        atomic_clear_int(((volatile uint_t *)(void *)p) + (b >> 5), \
        1 << (b & 0x1f))

#define test_bit(b, p) \
        (((volatile uint_t *)(void *)p)[b >> 5] & (1 << (b & 0x1f)))

/*
 * Note: this routine doesn't return old value. It return
 * 0 when succeeds, or -1 when fails.
 */
#ifdef _LP64
#define test_and_set_bit(b, p) \
        atomic_set_long_excl(((ulong_t *)(void *)p) + (b >> 6), (b & 0x3f))
#else
#define test_and_set_bit(b, p) \
        atomic_set_long_excl(((ulong_t *)(void *)p) + (b >> 5), (b & 0x1f))
#endif

#ifdef  __cplusplus
}
```

_____*unchanged_portion_omitted_*

```
**********************************************************
   31843 Mon Jul 28 07:44:35 2014
new/usr/src/uts/common/io/fcoe/fcoe.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1148 static void
1149 fcoe_worker_frame(void *arg)
1150 {
1151         fcoe_worker_t   *w = (fcoe_worker_t *)arg;
1152         fcoe_i_frame_t  *fmi;
1153         int             ret;

1155         atomic_inc_32(&fcoe_nworkers_running);
1155         atomic_add_32(&fcoe_nworkers_running, 1);
1156         mutex_enter(&w->worker_lock);
1157         w->worker_flags |= FCOE_WORKER_STARTED | FCOE_WORKER_ACTIVE;
1158         while ((w->worker_flags & FCOE_WORKER_TERMINATE) == 0) {
1159                 /*
1160                  * loop through the frames
1161                  */
1162                 while (fmi = list_head(&w->worker_frm_list)) {
1163                         list_remove(&w->worker_frm_list, fmi);
1164                         mutex_exit(&w->worker_lock);
1165                         /*
1166                          * do the checksum
1167                          */
1168                         ret = fcoe_crc_verify(fmi->fmi_frame);
1169                         if (ret == FCOE_SUCCESS) {
1170                                 fmi->fmi_mac->fm_client.ect_rx_frame(
1171                                     fmi->fmi_frame);
1172                         } else {
1173                                 fcoe_release_frame(fmi->fmi_frame);
1174                         }
1175                         mutex_enter(&w->worker_lock);
1176                         w->worker_ntasks--;
1177                 }
1178                 w->worker_flags &= ~FCOE_WORKER_ACTIVE;
1179                 cv_wait(&w->worker_cv, &w->worker_lock);
1180                 w->worker_flags |= FCOE_WORKER_ACTIVE;
1181         }
1182         w->worker_flags &= ~(FCOE_WORKER_STARTED | FCOE_WORKER_ACTIVE);
1183         mutex_exit(&w->worker_lock);
1184         atomic_dec_32(&fcoe_nworkers_running);
1184         atomic_add_32(&fcoe_nworkers_running, -1);
1185         list_destroy(&w->worker_frm_list);
1186 }
______unchanged_portion_omitted_
```

```
**********************************************************
  160919 Mon Jul 28 07:44:35 2014
new/usr/src/uts/common/io/fibre-channel/fca/emlxs/emlxs_sli3.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_


4615 /*
4616  *  emlxs_sli3_handle_ring_event()
4617  *
4618  *    Description: Process a Ring Attention.
4619  */
4620 static void
4621 emlxs_sli3_handle_ring_event(emlxs_hba_t *hba, int32_t ring_no,
4622     uint32_t ha_copy)
4623 {
4624         emlxs_port_t *port = &PPORT;
4625         SLIM2 *slim2p = (SLIM2 *)hba->sli.sli3.slim2.virt;
4626         CHANNEL *cp;
4627         RING *rp;
4628         IOCB *entry;
4629         IOCBQ *iocbq;
4630         IOCBQ local_iocbq;
4631         PGP *pgp;
4632         uint32_t count;
4633         volatile uint32_t chipatt;
4634         void *ioa2;
4635         uint32_t reg;
4636         uint32_t channel_no;
4637         off_t offset;
4638         IOCBQ *rsp_head = NULL;
4639         IOCBQ *rsp_tail = NULL;
4640         emlxs_buf_t *sbp = NULL;

4642         count = 0;
4643         rp = &hba->sli.sli3.ring[ring_no];
4644         cp = rp->channelp;
4645         channel_no = cp->channelno;

4647         /*
4648          * Isolate this ring's host attention bits
4649          * This makes all ring attention bits equal
4650          * to Ring0 attention bits
4651          */
4652         reg = (ha_copy >> (ring_no * 4)) & 0x0f;

4654         /*
4655          * Gather iocb entries off response ring.
4656          * Ensure entry is owned by the host.
4657          */
4658         pgp = (PGP *)&slim2p->mbx.us.s2.port[ring_no];
4659         offset =
4660             (off_t)((uint64_t)((unsigned long)&(pgp->rspPutInx)) -
4661             (uint64_t)((unsigned long)slim2p));
4662         EMLXS_MPDATA_SYNC(hba->sli.sli3.slim2.dma_handle, offset, 4,
4663             DDI_DMA_SYNC_FORKERNEL);
4664         rp->fc_port_rspidx = BE_SWAP32(pgp->rspPutInx);

4666         /* While ring is not empty */
4667         while (rp->fc_rspidx != rp->fc_port_rspidx) {
4668                 HBASTATS.IocbReceived[channel_no]++;

4670                 /* Get the next response ring iocb */
4671                 entry =
4672                     (IOCB *)(((char *)rp->fc_rspringaddr +
```



```
4673                     (rp->fc_rspidx * hba->sli.sli3.iocb_rsp_size)));

4675                 /* DMA sync the response ring iocb for the adapter */
4676                 offset = (off_t)((uint64_t)((unsigned long)entry)
4677                     - (uint64_t)((unsigned long)slim2p));
4678                 EMLXS_MPDATA_SYNC(hba->sli.sli3.slim2.dma_handle, offset,
4679                     hba->sli.sli3.iocb_rsp_size, DDI_DMA_SYNC_FORKERNEL);

4681                 count++;

4683                 /* Copy word6 and word7 to local iocb for now */
4684                 iocbq = &local_iocbq;

4686                 BE_SWAP32_BCOPY((uint8_t *)entry + (sizeof (uint32_t) * 6),
4687                     (uint8_t *)iocbq + (sizeof (uint32_t) * 6),
4688                     (sizeof (uint32_t) * 2));

4690                 /* when LE is not set, entire Command has not been received */
4691                 if (!iocbq->iocb.ULPLE) {
4692                         /* This should never happen */
4693                         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_ring_error_msg,
4694                             "ulpLE is not set. "
4695                             "ring=%d iotag=%x cmd=%x status=%x",
4696                             channel_no, iocbq->iocb.ULPIOTAG,
4697                             iocbq->iocb.ULPCOMMAND, iocbq->iocb.ULPSTATUS);

4699                         goto next;
4700                 }

4702                 switch (iocbq->iocb.ULPCOMMAND) {
4703 #ifdef SFCT_SUPPORT
4704                 case CMD_CLOSE_XRI_CX:
4705                 case CMD_CLOSE_XRI_CN:
4706                 case CMD_ABORT_XRI_CX:
4707                         if (!port->tgt_mode) {
4708                                 sbp = NULL;
4709                                 break;
4710                         }

4712                         sbp =
4713                             emlxs_unregister_pkt(cp, iocbq->iocb.ULPIOTAG, 0);
4714                         break;
4715 #endif /* SFCT_SUPPORT */

4717                         /* Ring 0 registered commands */
4718                 case CMD_FCP_ICMND_CR:
4719                 case CMD_FCP_ICMND_CX:
4720                 case CMD_FCP_IREAD_CR:
4721                 case CMD_FCP_IREAD_CX:
4722                 case CMD_FCP_IWRITE_CR:
4723                 case CMD_FCP_IWRITE_CX:
4724                 case CMD_FCP_ICMND64_CR:
4725                 case CMD_FCP_ICMND64_CX:
4726                 case CMD_FCP_IREAD64_CR:
4727                 case CMD_FCP_IREAD64_CX:
4728                 case CMD_FCP_IWRITE64_CR:
4729                 case CMD_FCP_IWRITE64_CX:
4730 #ifdef SFCT_SUPPORT
4731                 case CMD_FCP_TSEND_CX:
4732                 case CMD_FCP_TSEND64_CX:
4733                 case CMD_FCP_TRECEIVE_CX:
4734                 case CMD_FCP_TRECEIVE64_CX:
4735                 case CMD_FCP_TRSP_CX:
4736                 case CMD_FCP_TRSP64_CX:
4737 #endif /* SFCT_SUPPORT */
```

```
4739                        /* Ring 1 registered commands */
4740                case CMD_XMIT_BCAST_CN:
4741                case CMD_XMIT_BCAST_CX:
4742                case CMD_XMIT_SEQUENCE_CX:
4743                case CMD_XMIT_SEQUENCE_CR:
4744                case CMD_XMIT_BCAST64_CN:
4745                case CMD_XMIT_BCAST64_CX:
4746                case CMD_XMIT_SEQUENCE64_CX:
4747                case CMD_XMIT_SEQUENCE64_CR:
4748                case CMD_CREATE_XRI_CR:
4749                case CMD_CREATE_XRI_CX:

4751                        /* Ring 2 registered commands */
4752                case CMD_ELS_REQUEST_CR:
4753                case CMD_ELS_REQUEST_CX:
4754                case CMD_XMIT_ELS_RSP_CX:
4755                case CMD_ELS_REQUEST64_CR:
4756                case CMD_ELS_REQUEST64_CX:
4757                case CMD_XMIT_ELS_RSP64_CX:

4759                        /* Ring 3 registered commands */
4760                case CMD_GEN_REQUEST64_CR:
4761                case CMD_GEN_REQUEST64_CX:

4763                        sbp =
4764                            emlxs_unregister_pkt(cp, iocbq->iocb.ULPIOTAG, 0);
4765                        break;

4767                default:
4768                        sbp = NULL;
4769                }

4771                /* If packet is stale, then drop it. */
4772                if (sbp == STALE_PACKET) {
4773                        cp->hbaCmplCmd_sbp++;
4774                        /* Copy entry to the local iocbq */
4775                        BE_SWAP32_BCOPY((uint8_t *)entry,
4776                            (uint8_t *)iocbq, hba->sli.sli3.iocb_rsp_size);

4778                        EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_iocb_stale_msg,
4779                            "channelno=%d iocb=%p cmd=%x status=%x "
4780                            "error=%x iotag=%x context=%x info=%x",
4781                            channel_no, iocbq, (uint8_t)iocbq->iocb.ULPCOMMAND,
4782                            iocbq->iocb.ULPSTATUS,
4783                            (uint8_t)iocbq->iocb.un.grsp.perr.statLocalError,
4784                            (uint16_t)iocbq->iocb.ULPIOTAG,
4785                            (uint16_t)iocbq->iocb.ULPCONTEXT,
4786                            (uint8_t)iocbq->iocb.ULPRSVDBYTE);

4788                        goto next;
4789                }

4791                /*
4792                 * If a packet was found, then queue the packet's
4793                 * iocb for deferred processing
4794                 */
4795                else if (sbp) {
4796 #ifdef SFCT_SUPPORT
4797                        fct_cmd_t *fct_cmd;
4798                        emlxs_buf_t *cmd_sbp;

4800                        fct_cmd = sbp->fct_cmd;
4801                        if (fct_cmd) {
4802                                cmd_sbp =
4803                                    (emlxs_buf_t *)fct_cmd->cmd_fca_private;
4804                                mutex_enter(&cmd_sbp->fct_mtx);
```



```
4805                                EMLXS_FCT_STATE_CHG(fct_cmd, cmd_sbp,
4806                                    EMLXS_FCT_IOCB_COMPLETE);
4807                                mutex_exit(&cmd_sbp->fct_mtx);
4808                        }
4809 #endif /* SFCT_SUPPORT */
4810                        cp->hbaCmplCmd_sbp++;
4811                        atomic_dec_32(&hba->io_active);
4811                        atomic_add_32(&hba->io_active, -1);

4813                        /* Copy entry to sbp's iocbq */
4814                        iocbq = &sbp->iocbq;
4815                        BE_SWAP32_BCOPY((uint8_t *)entry,
4816                            (uint8_t *)iocbq, hba->sli.sli3.iocb_rsp_size);

4818                        iocbq->next = NULL;

4820                        /*
4821                         * If this is NOT a polled command completion
4822                         * or a driver allocated pkt, then defer pkt
4823                         * completion.
4824                         */
4825                        if (!(sbp->pkt_flags &
4826                            (PACKET_POLLED | PACKET_ALLOCATED))) {
4827                                /* Add the IOCB to the local list */
4828                                if (!rsp_head) {
4829                                        rsp_head = iocbq;
4830                                } else {
4831                                        rsp_tail->next = iocbq;
4832                                }

4834                                rsp_tail = iocbq;

4836                                goto next;
4837                        }
4838                } else {
4839                        cp->hbaCmplCmd++;
4840                        /* Copy entry to the local iocbq */
4841                        BE_SWAP32_BCOPY((uint8_t *)entry,
4842                            (uint8_t *)iocbq, hba->sli.sli3.iocb_rsp_size);

4844                        iocbq->next = NULL;
4845                        iocbq->bp = NULL;
4846                        iocbq->port = &PPORT;
4847                        iocbq->channel = cp;
4848                        iocbq->node = NULL;
4849                        iocbq->sbp = NULL;
4850                        iocbq->flag = 0;
4851                }

4853                /* process the channel event now */
4854                emlxs_proc_channel_event(hba, cp, iocbq);

4856 next:
4857                /* Increment the driver's local response get index */
4858                if (++rp->fc_rspidx >= rp->fc_numRiocb) {
4859                        rp->fc_rspidx = 0;
4860                }

4862        }       /* while (TRUE) */

4864        if (rsp_head) {
4865                mutex_enter(&cp->rsp_lock);
4866                if (cp->rsp_head == NULL) {
4867                        cp->rsp_head = rsp_head;
4868                        cp->rsp_tail = rsp_tail;
4869                } else {
```

```
4870                        cp->rsp_tail->next = rsp_head;
4871                        cp->rsp_tail = rsp_tail;
4872                }
4873                mutex_exit(&cp->rsp_lock);

4875                emlxs_thread_trigger2(&cp->intr_thread, emlxs_proc_channel, cp);
4876        }

4878        /* Check if at least one response entry was processed */
4879        if (count) {
4880                /* Update response get index for the adapter */
4881                if (hba->bus_type == SBUS_FC) {
4882                        slim2p->mbx.us.s2.host[channel_no].rspGetInx
4883                            = BE_SWAP32(rp->fc_rspidx);

4885                        /* DMA sync the index for the adapter */
4886                        offset = (off_t)
4887                            ((uint64_t)((unsigned long)&(slim2p->mbx.us.s2.
4888                            host[channel_no].rspGetInx))
4889                            - (uint64_t)((unsigned long)slim2p));
4890                        EMLXS_MPDATA_SYNC(hba->sli.sli3.slim2.dma_handle,
4891                            offset, 4, DDI_DMA_SYNC_FORDEV);
4892                } else {
4893                        ioa2 =
4894                            (void *)((char *)hba->sli.sli3.slim_addr +
4895                            hba->sli.sli3.hgp_ring_offset + (((channel_no * 2) +
4896                            1) * sizeof (uint32_t)));
4897                        WRITE_SLIM_ADDR(hba, (volatile uint32_t *)ioa2,
4898                            rp->fc_rspidx);
4899 #ifdef FMA_SUPPORT
4900                        /* Access handle validation */
4901                        EMLXS_CHK_ACC_HANDLE(hba,
4902                            hba->sli.sli3.slim_acc_handle);
4903 #endif  /* FMA_SUPPORT */
4904                }

4906                if (reg & HA_R0RE_REQ) {
4907                        /* HBASTATS.chipRingFree++; */

4909                        mutex_enter(&EMLXS_PORT_LOCK);

4911                        /* Tell the adapter we serviced the ring */
4912                        chipatt = ((CA_R0ATT | CA_R0RE_RSP) <<
4913                            (channel_no * 4));
4914                        WRITE_CSR_REG(hba, FC_CA_REG(hba), chipatt);

4916 #ifdef FMA_SUPPORT
4917                        /* Access handle validation */
4918                        EMLXS_CHK_ACC_HANDLE(hba, hba->sli.sli3.csr_acc_handle);
4919 #endif  /* FMA_SUPPORT */

4921                        mutex_exit(&EMLXS_PORT_LOCK);
4922                }
4923        }

4925        if ((reg & HA_R0CE_RSP) || hba->channel_tx_count) {
4926                /* HBASTATS.hostRingFree++; */

4928                /* Cmd ring may be available. Try sending more iocbs */
4929                emlxs_sli3_issue_iocb_cmd(hba, cp, 0);
4930        }

4932        /* HBASTATS.ringEvent++; */

4934        return;
```



```
4936 } /* emlxs_sli3_handle_ring_event() */
```

_____unchanged_portion_omitted_

```
5190 /* EMLXS_CMD_RING_LOCK must be held when calling this function */
5191 static void
5192 emlxs_sli3_issue_iocb(emlxs_hba_t *hba, RING *rp, IOCBQ *iocbq)
5193 {
5194        emlxs_port_t *port;
5195        IOCB *icmd;
5196        IOCB *iocb;
5197        emlxs_buf_t *sbp;
5198        off_t offset;
5199        uint32_t ringno;

5201        ringno = rp->ringno;
5202        sbp = iocbq->sbp;
5203        icmd = &iocbq->iocb;
5204        port = iocbq->port;

5206        HBASTATS.IocbIssued[ringno]++;

5208        /* Check for ULP pkt request */
5209        if (sbp) {
5210                mutex_enter(&sbp->mtx);

5212                if (sbp->node == NULL) {
5213                        /* Set node to base node by default */
5214                        iocbq->node = (void *)&port->node_base;
5215                        sbp->node = (void *)&port->node_base;
5216                }

5218                sbp->pkt_flags |= PACKET_IN_CHIPQ;
5219                mutex_exit(&sbp->mtx);

5221                atomic_inc_32(&hba->io_active);
5221                atomic_add_32(&hba->io_active, 1);

5223 #ifdef SFCT_SUPPORT
5224 #ifdef FCT_IO_TRACE
5225                if (sbp->fct_cmd) {
5226                        emlxs_fct_io_trace(port, sbp->fct_cmd,
5227                            EMLXS_FCT_IOCB_ISSUED);
5228                        emlxs_fct_io_trace(port, sbp->fct_cmd,
5229                            icmd->ULPCOMMAND);
5230                }
5231 #endif /* FCT_IO_TRACE */
5232 #endif /* SFCT_SUPPORT */

5234                rp->channelp->hbaSendCmd_sbp++;
5235                iocbq->channel = rp->channelp;
5236        } else {
5237                rp->channelp->hbaSendCmd++;
5238        }

5240        /* get the next available command ring iocb */
5241        iocb =
5242            (IOCB *)(((char *)rp->fc_cmdringaddr +
5243            (rp->fc_cmdidx * hba->sli.sli3.iocb_cmd_size)));

5245        /* Copy the local iocb to the command ring iocb */
5246        BE_SWAP32_BCOPY((uint8_t *)icmd, (uint8_t *)iocb,
5247            hba->sli.sli3.iocb_cmd_size);

5249        /* DMA sync the command ring iocb for the adapter */
5250        offset = (off_t)((uint64_t)((unsigned long)iocb)
```

```
5251             - (uint64_t)((unsigned long)hba->sli.sli3.slim2.virt));
5252         EMLXS_MPDATA_SYNC(hba->sli.sli3.slim2.dma_handle, offset,
5253             hba->sli.sli3.iocb_cmd_size, DDI_DMA_SYNC_FORDEV);

5255         /*
5256          * After this, the sbp / iocb should not be
5257          * accessed in the xmit path.
5258          */

5260         /* Free the local iocb if there is no sbp tracking it */
5261         if (!sbp) {
5262                 emlxs_mem_put(hba, MEM_IOCB, (void *)iocbq);
5263         }

5265         /* update local ring index to next available ring index */
5266         rp->fc_cmdidx =
5267             (rp->fc_cmdidx + 1 >= rp->fc_numCiocb) ? 0 : rp->fc_cmdidx + 1;


5270         return;

5272 } /* emlxs_sli3_issue_iocb() */
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  168075 Mon Jul 28 07:44:35 2014
new/usr/src/uts/common/io/fibre-channel/fca/emlxs/emlxs_sli4.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_
```

```
1825 static void
1826 emlxs_sli4_issue_iocb_cmd(emlxs_hba_t *hba, CHANNEL *cp, IOCBQ *iocbq)
1827 {
1828         emlxs_port_t *port = &PPORT;
1829         emlxs_buf_t *sbp;
1830         uint32_t channelno;
1831         int32_t throttle;
1832         emlxs_wqe_t *wqe;
1833         emlxs_wqe_t *wqeslot;
1834         WQ_DESC_t *wq;
1835         uint32_t flag;
1836         uint32_t wqdb;
1837         uint16_t next_wqe;
1838         off_t offset;


1841         channelno = cp->channelno;
1842         wq = (WQ_DESC_t *)cp->iopath;

1844 #ifdef SLI4_FASTPATH_DEBUG
1845         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_sli_detail_msg,
1846             "ISSUE WQE channel: %x  %p", channelno, wq);
1847 #endif

1849         throttle = 0;

1851         /* Check if FCP ring and adapter is not ready */
1852         /* We may use any ring for FCP_CMD */
1853         if (iocbq && (iocbq->flag & IOCB_FCP_CMD) && (hba->state != FC_READY)) {
1854                 if (!(iocbq->flag & IOCB_SPECIAL) || !iocbq->port ||
1855                     !(((emlxs_port_t *)iocbq->port)->tgt_mode)) {
1856                         emlxs_tx_put(iocbq, 1);
1857                         return;
1858                 }
1859         }

1861         /* Attempt to acquire CMD_RING lock */
1862         if (mutex_tryenter(&EMLXS_QUE_LOCK(channelno)) == 0) {
1863                 /* Queue it for later */
1864                 if (iocbq) {
1865                         if ((hba->io_count -
1866                             hba->channel_tx_count) > 10) {
1867                                 emlxs_tx_put(iocbq, 1);
1868                                 return;
1869                         } else {

1871                                 mutex_enter(&EMLXS_QUE_LOCK(channelno));
1872                         }
1873                 } else {
1874                         return;
1875                 }
1876         }
1877         /* EMLXS_QUE_LOCK acquired */

1879         /* Throttle check only applies to non special iocb */
1880         if (iocbq && (!(iocbq->flag & IOCB_SPECIAL))) {
```



```
1881                 /* Check if HBA is full */
1882                 throttle = hba->io_throttle - hba->io_active;
1883                 if (throttle <= 0) {
1884                         /* Hitting adapter throttle limit */
1885                         /* Queue it for later */
1886                         if (iocbq) {
1887                                 emlxs_tx_put(iocbq, 1);
1888                         }

1890                         goto busy;
1891                 }
1892         }

1894         /* Check to see if we have room for this WQE */
1895         next_wqe = wq->host_index + 1;
1896         if (next_wqe >= wq->max_index) {
1897                 next_wqe = 0;
1898         }

1900         if (next_wqe == wq->port_index) {
1901                 /* Queue it for later */
1902                 if (iocbq) {
1903                         emlxs_tx_put(iocbq, 1);
1904                 }
1905                 goto busy;
1906         }

1908         /*
1909          * We have a command ring slot available
1910          * Make sure we have an iocb to send
1911          */
1912         if (iocbq) {
1913                 mutex_enter(&EMLXS_TX_CHANNEL_LOCK);

1915                 /* Check if the ring already has iocb's waiting */
1916                 if (cp->nodeq.q_first != NULL) {
1917                         /* Put the current iocbq on the tx queue */
1918                         emlxs_tx_put(iocbq, 0);

1920                         /*
1921                          * Attempt to replace it with the next iocbq
1922                          * in the tx queue
1923                          */
1924                         iocbq = emlxs_tx_get(cp, 0);
1925                 }

1927                 mutex_exit(&EMLXS_TX_CHANNEL_LOCK);
1928         } else {
1929                 iocbq = emlxs_tx_get(cp, 1);
1930         }

1932 sendit:
1933         /* Process each iocbq */
1934         while (iocbq) {

1936                 wqe = &iocbq->wqe;
1937 #ifdef SLI4_FASTPATH_DEBUG
1938                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_sli_detail_msg,
1939                     "ISSUE QID %d WQE iotag: %x xri: %x", wq->qid,
1940                     wqe->RequestTag, wqe->XRITag);
1941 #endif

1943                 sbp = iocbq->sbp;
1944                 if (sbp) {
1945                         /* If exchange removed after wqe was prep'ed, drop it */
1946                         if (!(sbp->xrip)) {
```

```
1947                                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_sli_detail_msg,
1948                                    "Xmit WQE iotag: %x xri: %x aborted",
1949                                    wqe->RequestTag, wqe->XRITag);

1951                                /* Get next iocb from the tx queue */
1952                                iocbq = emlxs_tx_get(cp, 1);
1953                                continue;
1954                        }

1956                        if (sbp->pkt_flags & PACKET_DELAY_REQUIRED) {

1958                                /* Perform delay */
1959                                if ((channelno == hba->channel_els) &&
1960                                    !(iocbq->flag & IOCB_FCP_CMD)) {
1961                                        drv_usecwait(100000);
1962                                } else {
1963                                        drv_usecwait(20000);
1964                                }
1965                        }
1966                }

1968                /*
1969                 * At this point, we have a command ring slot available
1970                 * and an iocb to send
1971                 */
1972                wq->release_depth--;
1973                if (wq->release_depth == 0) {
1974                        wq->release_depth = WQE_RELEASE_DEPTH;
1975                        wqe->WQEC = 1;
1976                }


1979                HBASTATS.IocbIssued[channelno]++;

1981                /* Check for ULP pkt request */
1982                if (sbp) {
1983                        mutex_enter(&sbp->mtx);

1985                        if (sbp->node == NULL) {
1986                                /* Set node to base node by default */
1987                                iocbq->node = (void *)&port->node_base;
1988                                sbp->node = (void *)&port->node_base;
1989                        }

1991                        sbp->pkt_flags |= PACKET_IN_CHIPQ;
1992                        mutex_exit(&sbp->mtx);

1994                        atomic_inc_32(&hba->io_active);
1994                        atomic_add_32(&hba->io_active, 1);
1995                        sbp->xrip->flag |= EMLXS_XRI_PENDING_IO;
1996                }


1999                /* Free the local iocb if there is no sbp tracking it */
2000                if (sbp) {
2001 #ifdef SFCT_SUPPORT
2002 #ifdef FCT_IO_TRACE
2003                        if (sbp->fct_cmd) {
2004                                emlxs_fct_io_trace(port, sbp->fct_cmd,
2005                                    EMLXS_FCT_IOCB_ISSUED);
2006                                emlxs_fct_io_trace(port, sbp->fct_cmd,
2007                                    icmd->ULPCOMMAND);
2008                        }
2009 #endif /* FCT_IO_TRACE */
2010 #endif /* SFCT_SUPPORT */
2011                        cp->hbaSendCmd_sbp++;
```



```
2012                        iocbq->channel = cp;
2013                } else {
2014                        cp->hbaSendCmd++;
2015                }

2017                flag = iocbq->flag;

2019                /* Send the iocb */
2020                wqeslot = (emlxs_wqe_t *)wq->addr.virt;
2021                wqeslot += wq->host_index;

2023                wqe->CQId = wq->cqid;
2024                BE_SWAP32_BCOPY((uint8_t *)wqe, (uint8_t *)wqeslot,
2025                    sizeof (emlxs_wqe_t));
2026 #ifdef DEBUG_WQE
2027                emlxs_data_dump(port, "WQE", (uint32_t *)wqe, 18, 0);
2028 #endif
2029                offset = (off_t)((uint64_t)((unsigned long)
2030                    wq->addr.virt) -
2031                    (uint64_t)((unsigned long)
2032                    hba->sli.sli4.slim2.virt));

2034                EMLXS_MPDATA_SYNC(wq->addr.dma_handle, offset,
2035                    4096, DDI_DMA_SYNC_FORDEV);

2037                /* Ring the WQ Doorbell */
2038                wqdb = wq->qid;
2039                wqdb |= ((1 << 24) | (wq->host_index << 16));


2042                WRITE_BAR2_REG(hba, FC_WQDB_REG(hba), wqdb);
2043                wq->host_index = next_wqe;

2045 #ifdef SLI4_FASTPATH_DEBUG
2046                EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_sli_detail_msg,
2047                    "WQ RING: %08x", wqdb);
2048 #endif

2050                /*
2051                 * After this, the sbp / iocb / wqe should not be
2052                 * accessed in the xmit path.
2053                 */

2055                if (!sbp) {
2056                        emlxs_mem_put(hba, MEM_IOCB, (void *)iocbq);
2057                }

2059                if (iocbq && (!(flag & IOCB_SPECIAL))) {
2060                        /* Check if HBA is full */
2061                        throttle = hba->io_throttle - hba->io_active;
2062                        if (throttle <= 0) {
2063                                goto busy;
2064                        }
2065                }

2067                /* Check to see if we have room for another WQE */
2068                next_wqe++;
2069                if (next_wqe >= wq->max_index) {
2070                        next_wqe = 0;
2071                }

2073                if (next_wqe == wq->port_index) {
2074                        /* Queue it for later */
2075                        goto busy;
2076                }
```

```
2079                    /* Get the next iocb from the tx queue if there is one */
2080                    iocbq = emlxs_tx_get(cp, 1);
2081            }

2083            mutex_exit(&EMLXS_QUE_LOCK(channelno));

2085            return;

2087    busy:
2088            if (throttle <= 0) {
2089                    HBASTATS.IocbThrottled++;
2090            } else {
2091                    HBASTATS.IocbRingFull[channelno]++;
2092            }

2094            mutex_exit(&EMLXS_QUE_LOCK(channelno));

2096            return;

2098    } /* emlxs_sli4_issue_iocb_cmd() */
```

_____***unchanged_portion_omitted_***

```
3828    /*ARGSUSED*/
3829    static void
3830    emlxs_sli4_hba_flush_chipq(emlxs_hba_t *hba)
3831    {
3832    #ifdef SFCT_SUPPORT
3833    #ifdef FCT_IO_TRACE
3834            emlxs_port_t *port = &PPORT;
3835    #endif /* FCT_IO_TRACE */
3836    #endif /* SFCT_SUPPORT */
3837            CHANNEL *cp;
3838            emlxs_buf_t *sbp;
3839            IOCBQ *iocbq;
3840            uint16_t i;
3841            uint32_t trigger;
3842            CQE_CmplWQ_t cqe;

3844            mutex_enter(&EMLXS_FCTAB_LOCK);
3845            for (i = 0; i < hba->max_iotag; i++) {
3846                    sbp = hba->fc_table[i];
3847                    if (sbp == NULL || sbp == STALE_PACKET) {
3848                            continue;
3849                    }
3850                    hba->fc_table[i] = STALE_PACKET;
3851                    hba->io_count--;
3852                    sbp->iotag = 0;
3853                    mutex_exit(&EMLXS_FCTAB_LOCK);

3855                    cp = sbp->channel;
3856                    bzero(&cqe, sizeof (CQE_CmplWQ_t));
3857                    cqe.RequestTag = i;
3858                    cqe.Status = IOSTAT_LOCAL_REJECT;
3859                    cqe.Parameter = IOERR_SEQUENCE_TIMEOUT;

3861                    cp->hbaCmplCmd_sbp++;

3863    #ifdef SFCT_SUPPORT
3864    #ifdef FCT_IO_TRACE
3865                    if (sbp->fct_cmd) {
3866                            emlxs_fct_io_trace(port, sbp->fct_cmd,
3867                                EMLXS_FCT_IOCB_COMPLETE);
3868                    }
3869    #endif /* FCT_IO_TRACE */
```



```
3870    #endif /* SFCT_SUPPORT */

3872                    atomic_dec_32(&hba->io_active);
3872                    atomic_add_32(&hba->io_active, -1);

3874                    /* Copy entry to sbp's iocbq */
3875                    iocbq = &sbp->iocbq;
3876                    emlxs_CQE_to_IOCB(hba, &cqe, sbp);

3878                    iocbq->next = NULL;

3880                    /* Exchange is no longer busy on-chip, free it */
3881                    emlxs_sli4_free_xri(hba, sbp, sbp->xrip, 1);

3883                    if (!(sbp->pkt_flags &
3884                        (PACKET_POLLED | PACKET_ALLOCATED))) {
3885                            /* Add the IOCB to the channel list */
3886                            mutex_enter(&cp->rsp_lock);
3887                            if (cp->rsp_head == NULL) {
3888                                    cp->rsp_head = iocbq;
3889                                    cp->rsp_tail = iocbq;
3890                            } else {
3891                                    cp->rsp_tail->next = iocbq;
3892                                    cp->rsp_tail = iocbq;
3893                            }
3894                            mutex_exit(&cp->rsp_lock);
3895                            trigger = 1;
3896                    } else {
3897                            emlxs_proc_channel_event(hba, cp, iocbq);
3898                    }
3899                    mutex_enter(&EMLXS_FCTAB_LOCK);
3900            }
3901            mutex_exit(&EMLXS_FCTAB_LOCK);

3903            if (trigger) {
3904                    for (i = 0; i < hba->chan_count; i++) {
3905                            cp = &hba->chan[i];
3906                            if (cp->rsp_head != NULL) {
3907                                    emlxs_thread_trigger2(&cp->intr_thread,
3908                                        emlxs_proc_channel, cp);
3909                            }
3910                    }
3911            }

3913    } /* emlxs_sli4_hba_flush_chipq() */
```

_____***unchanged_portion_omitted_***

```
3945    /*ARGSUSED*/
3946    static void
3947    emlxs_sli4_process_wqe_cmpl(emlxs_hba_t *hba, CQ_DESC_t *cq, CQE_CmplWQ_t *cqe)
3948    {
3949            emlxs_port_t *port = &PPORT;
3950            CHANNEL *cp;
3951            emlxs_buf_t *sbp;
3952            IOCBQ *iocbq;
3953            uint16_t request_tag;
3954    #ifdef SFCT_SUPPORT
3955            fct_cmd_t *fct_cmd;
3956            emlxs_buf_t *cmd_sbp;
3957    #endif /* SFCT_SUPPORT */

3959            request_tag = cqe->RequestTag;

3961            /* 1 to 1 mapping between CQ and channel */
3962            cp = cq->channelp;
```

```
3964         mutex_enter(&EMLXS_FCTAB_LOCK);
3965         sbp = hba->fc_table[request_tag];
3966         atomic_dec_32(&hba->io_active);
3966         atomic_add_32(&hba->io_active, -1);

3968         if (sbp == STALE_PACKET) {
3969                 cp->hbaCmplCmd_sbp++;
3970                 mutex_exit(&EMLXS_FCTAB_LOCK);
3971                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_sli_detail_msg,
3972                     "CQ ENTRY: Stale sbp. tag=%x. Dropping...", request_tag);
3973                 return;
3974         }

3976         if (!sbp || !(sbp->xrip)) {
3977                 cp->hbaCmplCmd++;
3978                 mutex_exit(&EMLXS_FCTAB_LOCK);
3979                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_sli_detail_msg,
3980                     "CQ ENTRY: NULL sbp %p. tag=%x. Dropping...",
3981                     sbp, request_tag);
3982                 return;
3983         }

3985 #ifdef SLI4_FASTPATH_DEBUG
3986         EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_sli_detail_msg,
3987             "CQ ENTRY: process wqe compl");
3988 #endif

3990         cp->hbaCmplCmd_sbp++;

3992         /* Copy entry to sbp's iocbq */
3993         iocbq = &sbp->iocbq;
3994         emlxs_CQE_to_IOCB(hba, cqe, sbp);

3996         iocbq->next = NULL;

3998         if (cqe->XB) {
3999                 /* Mark exchange as ABORT in progress */
4000                 sbp->xrip->flag &= ~EMLXS_XRI_PENDING_IO;
4001                 sbp->xrip->flag |= EMLXS_XRI_ABORT_INP;

4003                 EMLXS_MSGF(EMLXS_CONTEXT, &emlxs_sli_detail_msg,
4004                     "CQ ENTRY: ABORT INP: tag=%x xri=%x", request_tag,
4005                     sbp->xrip->XRI);

4007                 emlxs_sli4_free_xri(hba, sbp, 0, 0);
4008         } else {
4009                 /* Exchange is no longer busy on-chip, free it */
4010                 emlxs_sli4_free_xri(hba, sbp, sbp->xrip, 0);
4011         }

4013         mutex_exit(&EMLXS_FCTAB_LOCK);

4015 #ifdef SFCT_SUPPORT
4016         fct_cmd = sbp->fct_cmd;
4017         if (fct_cmd) {
4018                 cmd_sbp = (emlxs_buf_t *)fct_cmd->cmd_fca_private;
4019                 mutex_enter(&cmd_sbp->fct_mtx);
4020                 EMLXS_FCT_STATE_CHG(fct_cmd, cmd_sbp, EMLXS_FCT_IOCB_COMPLETE);
4021                 mutex_exit(&cmd_sbp->fct_mtx);
4022         }
4023 #endif /* SFCT_SUPPORT */

4025         /*
4026          * If this is NOT a polled command completion
4027          * or a driver allocated pkt, then defer pkt
```



```
4028          * completion.
4029          */
4030         if (!(sbp->pkt_flags &
4031             (PACKET_POLLED | PACKET_ALLOCATED))) {
4032                 /* Add the IOCB to the channel list */
4033                 mutex_enter(&cp->rsp_lock);
4034                 if (cp->rsp_head == NULL) {
4035                         cp->rsp_head = iocbq;
4036                         cp->rsp_tail = iocbq;
4037                 } else {
4038                         cp->rsp_tail->next = iocbq;
4039                         cp->rsp_tail = iocbq;
4040                 }
4041                 mutex_exit(&cp->rsp_lock);

4043                 /* Delay triggering thread till end of ISR */
4044                 cp->chan_flag |= EMLXS_NEEDS_TRIGGER;
4045         } else {
4046                 emlxs_proc_channel_event(hba, cp, iocbq);
4047         }

4049 } /* emlxs_sli4_process_wqe_cmpl() */
```

_____***unchanged_portion_omitted_***

****
26360 Mon Jul 28 07:44:35 2014
new/usr/src/uts/common/io/fibre-channel/fca/fcoei/fcoei.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
753 static void
754 fcoei_clear_watchdog_jobs(fcoei_soft_state_t *ss)
755 {
756         fcoei_event_t           *ae;
757         fcoe_frame_t            *frm;

759         mutex_enter(&ss->ss_watchdog_mutex);
760         while (!list_is_empty(&ss->ss_event_list)) {
761                 ae = (fcoei_event_t *)list_head(&ss->ss_event_list);
762                 list_remove(&ss->ss_event_list, ae);
763                 switch (ae->ae_type) {
764                 case AE_EVENT_SOL_FRAME:
765                         frm = (fcoe_frame_t *)ae->ae_obj;
766                         frm->frm_eport->eport_release_frame(frm);
767                         break;

769                 case AE_EVENT_UNSOL_FRAME:
770                         frm = (fcoe_frame_t *)ae->ae_obj;
771                         frm->frm_eport->eport_free_netb(frm->frm_netb);
772                         frm->frm_eport->eport_release_frame(frm);
773                         break;

775                 case AE_EVENT_PORT:
776                         atomic_dec_32(&ss->ss_port_event_counter);
776                         atomic_add_32(&ss->ss_port_event_counter, -1);
777                         /* FALLTHROUGH */

779                 case AE_EVENT_RESET:
780                         kmem_free(ae, sizeof (fcoei_event_t));
781                         break;

783                 case AE_EVENT_EXCHANGE:
784                         /* FALLTHROUGH */

786                 default:
787                         break;
788                 }
789         }

791         mod_hash_clear(ss->ss_unsol_rxid_hash);
792         mod_hash_clear(ss->ss_sol_oxid_hash);

794         while (!list_is_empty(&ss->ss_comp_xch_list)) {
795                 (void) list_remove_head(&ss->ss_comp_xch_list);
796         }
797         mutex_exit(&ss->ss_watchdog_mutex);
798 }
```

_____***unchanged_portion_omitted_***

****
34989 Mon Jul 28 07:44:36 2014
new/usr/src/uts/common/io/fibre-channel/fca/fcoei/fcoei_eth.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
1356 /*
1357  * fcoei_process_event_port
1358  *      link/port state changed
1359  *
1360  * Input:
1361  *      ae = link fcoei_event
1362  *
1363  * Returns:
1364  *      N/A
1365  *
1366  * Comments:
1367  *      asynchronous events from FCOE
1368  */
1369 void
1370 fcoei_process_event_port(fcoei_event_t *ae)
1371 {
1372         fcoei_soft_state_t      *ss = (fcoei_soft_state_t *)ae->ae_obj;

1374         if (ss->ss_eport->eport_link_speed == FCOE_PORT_SPEED_1G) {
1375                 ae->ae_specific |= FC_STATE_1GBIT_SPEED;
1376         } else if (ss->ss_eport->eport_link_speed ==
1377             FCOE_PORT_SPEED_10G) {
1378                 ae->ae_specific |= FC_STATE_10GBIT_SPEED;
1379         }

1381         if (ss->ss_flags & SS_FLAG_LV_BOUND) {
1382                 ss->ss_bind_info.port_statec_cb(ss->ss_port,
1383                     (uint32_t)ae->ae_specific);
1384         } else {
1385                 FCOEI_LOG(__FUNCTION__, "ss %p not bound now", ss);
1386         }

1388         atomic_dec_32(&ss->ss_port_event_counter);
1388         atomic_add_32(&ss->ss_port_event_counter, -1);
1389         kmem_free(ae, sizeof (fcoei_event_t));
1390 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   46984 Mon Jul 28 07:44:36 2014
new/usr/src/uts/common/io/fibre-channel/fca/fcoei/fcoei_lv.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_


 822 /*
 823  * fcoei_initiate_ct_req
 824  *      Fill and submit CT request
 825  *
 826  * Input:
 827  *      xch - the exchange that will be initiated
 828  *
 829  * Returns:
 830  *      N/A
 831  *
 832  * Comments:
 833  *      N/A
 834  */
 835 static void
 836 fcoei_initiate_ct_req(fcoei_exchange_t *xch)
 837 {
 838         fc_packet_t     *fpkt    = xch->xch_fpkt;
 839         fc_ct_header_t  *ct      = (fc_ct_header_t *)(void *)fpkt->pkt_cmd;
 840         uint8_t         *bp      = (uint8_t *)fpkt->pkt_cmd;
 841         fcoe_frame_t    *frm;
 842         int              offset;
 843         int              idx;
 844         uint32_t         cmd_len = fpkt->pkt_cmdlen;

 846         /*
 847          * Ensure it's 4-byte aligned
 848          */
 849         cmd_len = P2ROUNDUP(cmd_len, 4);

 851         /*
 852          * Allocate CT request frame
 853          */
 854         frm = xch->xch_ss->ss_eport->eport_alloc_frame(xch->xch_ss->ss_eport,
 855             cmd_len + FCFH_SIZE, NULL);
 856         if (frm == NULL) {
 857                 FCOEI_LOG(__FUNCTION__, "failed to alloc: %p", xch);
 858                 return;
 859         }

 861         bzero(frm->frm_payload, cmd_len);
 862         xch->xch_cnt = xch->xch_ss->ss_sol_cnt;
 863         atomic_inc_32(xch->xch_cnt);
 863         atomic_add_32(xch->xch_cnt, 1);

 865         FFM_R_CTL(fpkt->pkt_cmd_fhdr.r_ctl, frm);
 866         FFM_D_ID(fpkt->pkt_cmd_fhdr.d_id, frm);
 867         FFM_S_ID(fpkt->pkt_cmd_fhdr.s_id, frm);
 868         FFM_TYPE(fpkt->pkt_cmd_fhdr.type, frm);
 869         FFM_F_CTL(fpkt->pkt_cmd_fhdr.f_ctl, frm);
 870         FFM_OXID(xch->xch_oxid, frm);
 871         FFM_RXID(xch->xch_rxid, frm);
 872         fcoei_init_ifm(frm, xch);

 874         /*
 875          * CT header (FC payload)
 876          */
 877         offset = 0;
 878         FCOE_V2B_1(ct->ct_rev, FPLD + offset);
```



```
 880         offset = 1;
 881         FCOE_V2B_3(ct->ct_inid, FPLD + offset);

 883         offset = 4;
 884         FCOE_V2B_1(ct->ct_fcstype, FPLD + offset);

 886         offset = 5;
 887         FCOE_V2B_1(ct->ct_fcssubtype, FPLD + offset);

 889         offset = 6;
 890         FCOE_V2B_1(ct->ct_options, FPLD + offset);

 892         offset = 8;
 893         FCOE_V2B_2(ct->ct_cmdrsp, FPLD + offset);

 895         offset = 10;
 896         FCOE_V2B_2(ct->ct_aiusize, FPLD + offset);

 898         offset = 13;
 899         FCOE_V2B_1(ct->ct_reason, FPLD + offset);

 901         offset = 14;
 902         FCOE_V2B_1(ct->ct_expln, FPLD + offset);

 904         offset = 15;
 905         FCOE_V2B_1(ct->ct_vendor, FPLD + offset);

 907         /*
 908          * CT payload (FC payload)
 909          */
 910         switch (ct->ct_fcstype) {
 911         case FCSTYPE_DIRECTORY:
 912                 switch (ct->ct_cmdrsp) {
 913                 case NS_GA_NXT:
 914                 case NS_GPN_ID:
 915                 case NS_GNN_ID:
 916                 case NS_GCS_ID:
 917                 case NS_GFT_ID:
 918                 case NS_GSPN_ID:
 919                 case NS_GPT_ID:
 920                 case NS_GID_FT:
 921                 case NS_GID_PT:
 922                 case NS_DA_ID:
 923                         offset = 16;
 924                         FCOE_V2B_4(((uint32_t *)(intptr_t)(bp + offset))[0],
 925                             FPLD + offset);
 926                         break;

 928                 case NS_GID_PN:
 929                         offset = 16;
 930                         bcopy(bp + offset, FPLD + offset, 8);
 931                         break;

 933                 case NS_RNN_ID:
 934                 case NS_RPN_ID:
 935                         offset = 16;
 936                         FCOE_V2B_4(((uint32_t *)(intptr_t)(bp + offset))[0],
 937                             FPLD + offset);

 939                         offset = 20;
 940                         bcopy(bp + offset, FPLD + offset, 8);
 941                         break;

 943                 case NS_RSPN_ID:
 944                         offset = 16;
```

```
945                         FCOE_V2B_4(((uint32_t *)(intptr_t)(bp + offset))[0],
946                             FPLD + offset);
947
948                         offset = 20;
949                         bcopy(bp + offset, FPLD + offset, bp[20] + 1);
950                         break;
951
952                 case NS_RSNN_NN:
953                         offset = 16;
954                         bcopy(bp + offset, FPLD + offset, 8);
955
956                         offset = 24;
957                         bcopy(bp + offset, FPLD + offset, bp[24] + 1);
958                         break;
959
960                 case NS_RFT_ID:
961                         offset = 16;
962                         FCOE_V2B_4(((uint32_t *)(intptr_t)(bp + offset))[0],
963                             FPLD + offset);
964
965                         /*
966                          * fp use bcopy to copy fp_fc4_types,
967                          * we need to swap order for each integer
968                          */
969                         offset = 20;
970                         for (idx = 0; idx < 8; idx++) {
971                                 FCOE_V2B_4(
972                                     ((uint32_t *)(intptr_t)(bp + offset))[0],
973                                     FPLD + offset);
974                                 offset += 4;
975                         }
976                         break;
977
978                 case NS_RCS_ID:
979                 case NS_RPT_ID:
980                         offset = 16;
981                         FCOE_V2B_4(((uint32_t *)(intptr_t)(bp + offset))[0],
982                             FPLD + offset);
983
984                         offset = 20;
985                         FCOE_V2B_4(((uint32_t *)(intptr_t)(bp + offset))[0],
986                             FPLD + offset);
987                         break;
988
989                 case NS_RIP_NN:
990                         offset = 16;
991                         bcopy(bp + offset, FPLD + offset, 24);
992                         break;
993
994                 default:
995                         fcoei_complete_xch(xch, frm, FC_PKT_FAILURE,
996                             FC_REASON_CMD_UNSUPPORTED);
997                         break;
998                 }
999                 break; /* FCSTYPE_DIRECTORY */
1000
1001         case FCSTYPE_MGMTSERVICE:
1002                 switch (ct->ct_cmdrsp) {
1003                 case MS_GIEL:
1004                         FCOEI_LOG(__FUNCTION__,
1005                             "MS_GIEL ct_fcstype %x, ct_cmdrsp: %x",
1006                             ct->ct_fcstype, ct->ct_cmdrsp);
1007                         break;
1008
1009                 default:
1010                         fcoei_complete_xch(xch, frm, FC_PKT_FAILURE,
```



```
1011                             FC_REASON_CMD_UNSUPPORTED);
1012                         break;
1013                 }
1014                 break; /* FCSTYPE_MGMTSERVICE */
1015
1016         default:
1017                 fcoei_complete_xch(xch, frm, FC_PKT_FAILURE,
1018                     FC_REASON_CMD_UNSUPPORTED);
1019                 break;
1020         }
1021         xch->xch_ss->ss_eport->eport_tx_frame(frm);
1022 }
1023
1024 /*
1025  * fcoei_initiate_fcp_cmd
1026  *      Submit FCP command
1027  *
1028  * Input:
1029  *      xch - the exchange to be submitted
1030  *
1031  * Returns:
1032  *      N/A
1033  *
1034  * Comments:
1035  *      N/A
1036  */
1037 static void
1038 fcoei_initiate_fcp_cmd(fcoei_exchange_t *xch)
1039 {
1040         fc_packet_t     *fpkt = xch->xch_fpkt;
1041         fcoe_frame_t    *frm;
1042         fcp_cmd_t       *fcp_cmd_iu = (fcp_cmd_t *)(void *)fpkt->pkt_cmd;
1043         int              offset = 0;
1044
1045         ASSERT((fpkt->pkt_cmdlen % 4) == 0);
1046         frm = xch->xch_ss->ss_eport->eport_alloc_frame(xch->xch_ss->ss_eport,
1047             fpkt->pkt_cmdlen + FCFH_SIZE, NULL);
1048         if (!frm) {
1049                 ASSERT(0);
1050         } else {
1051                 fcoei_init_ifm(frm, xch);
1052                 bzero(frm->frm_payload, fpkt->pkt_cmdlen);
1053         }
1054
1055         /*
1056          * This will affect timing check
1057          */
1058         xch->xch_cnt = xch->xch_ss->ss_sol_cnt;
1059         atomic_inc_32(xch->xch_cnt);
1059         atomic_add_32(xch->xch_cnt, 1);
1060
1061         /*
1062          * Set exchange residual bytes
1063          */
1064         xch->xch_resid = (int)fpkt->pkt_datalen;
1065
1066         /*
1067          * Fill FCP command IU
1068          *
1069          * fcp_ent_addr
1070          */
1071         FCOE_V2B_2(fcp_cmd_iu->fcp_ent_addr.ent_addr_0,
1072             frm->frm_payload + offset);
1073         offset += 2;
1074         FCOE_V2B_2(fcp_cmd_iu->fcp_ent_addr.ent_addr_1,
1075             frm->frm_payload + offset);
```

```
1076         offset += 2;
1077         FCOE_V2B_2(fcp_cmd_iu->fcp_ent_addr.ent_addr_2,
1078             frm->frm_payload + offset);
1079         offset += 2;
1080         FCOE_V2B_2(fcp_cmd_iu->fcp_ent_addr.ent_addr_3,
1081             frm->frm_payload + offset);
1082         /*
1083          * fcp_cntl
1084          */
1085         offset = offsetof(fcp_cmd_t, fcp_cntl);
1086         frm->frm_payload[offset] = 0;

1088         offset += 1;
1089         frm->frm_payload[offset] = fcp_cmd_iu->fcp_cntl.cntl_qtype & 0x07;
1090         offset += 1;
1091         frm->frm_payload[offset] =
1092             (fcp_cmd_iu->fcp_cntl.cntl_kill_tsk << 7) |
1093             (fcp_cmd_iu->fcp_cntl.cntl_clr_aca << 6) |
1094             (fcp_cmd_iu->fcp_cntl.cntl_reset_tgt << 5) |
1095             (fcp_cmd_iu->fcp_cntl.cntl_reset_lun << 4) |
1096             (fcp_cmd_iu->fcp_cntl.cntl_clr_tsk << 2) |
1097             (fcp_cmd_iu->fcp_cntl.cntl_abort_tsk << 1);
1098         offset += 1;
1099         frm->frm_payload[offset] =
1100             (fcp_cmd_iu->fcp_cntl.cntl_read_data << 1) |
1101             (fcp_cmd_iu->fcp_cntl.cntl_write_data);
1102         /*
1103          * fcp_cdb
1104          */
1105         offset = offsetof(fcp_cmd_t, fcp_cdb);
1106         bcopy(fcp_cmd_iu->fcp_cdb, frm->frm_payload + offset, FCP_CDB_SIZE);
1107         /*
1108          * fcp_data_len
1109          */
1110         offset += FCP_CDB_SIZE;
1111         FCOE_V2B_4(fcp_cmd_iu->fcp_data_len, frm->frm_payload + offset);

1113         /*
1114          * FC frame header
1115          */
1116         FRM2IFM(frm)->ifm_rctl = fpkt->pkt_cmd_fhdr.r_ctl;

1118         FFM_R_CTL(fpkt->pkt_cmd_fhdr.r_ctl, frm);
1119         FFM_D_ID(fpkt->pkt_cmd_fhdr.d_id, frm);
1120         FFM_S_ID(fpkt->pkt_cmd_fhdr.s_id, frm);
1121         FFM_TYPE(fpkt->pkt_cmd_fhdr.type, frm);
1122         FFM_F_CTL(0x290000, frm);
1123         FFM_OXID(xch->xch_oxid, frm);
1124         FFM_RXID(xch->xch_rxid, frm);

1126         xch->xch_ss->ss_eport->eport_tx_frame(frm);
1127 }

1129 /*
1130  * fcoei_initiate_els_req
1131  *      Initiate ELS request
1132  *
1133  * Input:
1134  *      xch = the exchange that will be initiated
1135  *
1136  * Returns:
1137  *      N/A
1138  *
1139  * Comments:
1140  *      N/A
1141  */
```



```
1142 static void
1143 fcoei_initiate_els_req(fcoei_exchange_t *xch)
1144 {
1145         fc_packet_t     *fpkt = xch->xch_fpkt;
1146         fcoe_frame_t    *frm;
1147         ls_code_t       *els_code;

1149         ASSERT((fpkt->pkt_cmdlen % 4) == 0);
1150         frm = xch->xch_ss->ss_eport->eport_alloc_frame(xch->xch_ss->ss_eport,
1151             fpkt->pkt_cmdlen + FCFH_SIZE, NULL);
1152         if (!frm) {
1153                 ASSERT(0);
1154         } else {
1155                 fcoei_init_ifm(frm, xch);
1156                 bzero(frm->frm_payload, fpkt->pkt_cmdlen);
1157         }

1159         /*
1160          * This will affect timing check
1161          */
1162         xch->xch_cnt = xch->xch_ss->ss_sol_cnt;
1163         atomic_inc_32(xch->xch_cnt);
1163         atomic_add_32(xch->xch_cnt, 1);

1165         els_code = (ls_code_t *)(void *)fpkt->pkt_cmd;
1166         switch (els_code->ls_code) {
1167         case LA_ELS_FLOGI:
1168                 /*
1169                  * For FLOGI, we expect response within E_D_TOV
1170                  */
1171                 xch->xch_start_tick = ddi_get_lbolt();
1172                 xch->xch_end_tick = xch->xch_start_tick +
1173                     FCOE_SEC2TICK(2);
1174                 xch->xch_ss->ss_flags &= ~SS_FLAG_FLOGI_FAILED;
1175                 /* FALLTHROUGH */

1177         case LA_ELS_PLOGI:
1178                 fcoei_fill_els_logi_cmd(fpkt, frm);
1179                 break;

1181         case LA_ELS_PRLI:
1182                 fcoei_fill_els_prli_cmd(fpkt, frm);
1183                 break;

1185         case LA_ELS_SCR:
1186                 fcoei_fill_els_scr_cmd(fpkt, frm);
1187                 break;

1189         case LA_ELS_LINIT:
1190                 fcoei_fill_els_linit_cmd(fpkt, frm);
1191                 break;

1193         case LA_ELS_ADISC:
1194                 fcoei_fill_els_adisc_cmd(fpkt, frm);
1195                 break;

1197         case LA_ELS_LOGO:
1198                 /*
1199                  * For LOGO, we expect response within E_D_TOV
1200                  */
1201                 xch->xch_start_tick = ddi_get_lbolt();
1202                 xch->xch_end_tick = xch->xch_start_tick +
1203                     FCOE_SEC2TICK(2);
1204                 fcoei_fill_els_logo_cmd(fpkt, frm);
1205                 break;
1206         case LA_ELS_RLS:
```

```
1207                fcoei_fill_els_rls_cmd(fpkt, frm);
1208                break;
1209        case LA_ELS_RNID:
1210                fcoei_fill_els_rnid_cmd(fpkt, frm);
1211                break;
1212        default:
1213                fcoei_complete_xch(xch, frm, FC_PKT_FAILURE,
1214                    FC_REASON_CMD_UNSUPPORTED);
1215                return;
1216        }

1218        /*
1219         * set ifm_rtcl
1220         */
1221        FRM2IFM(frm)->ifm_rctl = fpkt->pkt_cmd_fhdr.r_ctl;

1223        /*
1224         * FCPH
1225         */
1226        FFM_R_CTL(fpkt->pkt_cmd_fhdr.r_ctl, frm);
1227        FFM_D_ID(fpkt->pkt_cmd_fhdr.d_id, frm);
1228        FFM_S_ID(fpkt->pkt_cmd_fhdr.s_id, frm);
1229        FFM_TYPE(fpkt->pkt_cmd_fhdr.type, frm);
1230        FFM_F_CTL(0x290000, frm);
1231        FFM_OXID(xch->xch_oxid, frm);
1232        FFM_RXID(xch->xch_rxid, frm);

1234        xch->xch_ss->ss_eport->eport_tx_frame(frm);
1235 }

1237 /*
1238  * fcoei_initiate_els_resp
1239  *      Originate ELS response
1240  *
1241  * Input:
1242  *      xch = the associated exchange
1243  *
1244  * Returns:
1245  *      N/A
1246  *
1247  * Comments:
1248  *      N/A
1249  */
1250 static void
1251 fcoei_initiate_els_resp(fcoei_exchange_t *xch)
1252 {
1253        fc_packet_t     *fpkt = xch->xch_fpkt;
1254        fcoe_frame_t    *frm;

1256        ASSERT((fpkt->pkt_cmdlen % 4) == 0);
1257        frm = xch->xch_ss->ss_eport->eport_alloc_frame(xch->xch_ss->ss_eport,
1258            fpkt->pkt_cmdlen + FCFH_SIZE, NULL);
1259        if (!frm) {
1260                ASSERT(0);
1261        } else {
1262                fcoei_init_ifm(frm, xch);
1263                bzero(frm->frm_payload, fpkt->pkt_cmdlen);
1264        }

1266        /*
1267         * This will affect timing check
1268         */
1269        xch->xch_cnt = xch->xch_ss->ss_unsol_cnt;
1270        atomic_inc_32(xch->xch_cnt);
1270        atomic_add_32(xch->xch_cnt, 1);
```



```
1272        /*
1273         * Set ifm_rctl
1274         */
1275        FRM2IFM(frm)->ifm_rctl = fpkt->pkt_cmd_fhdr.r_ctl;

1277        /*
1278         * FCPH
1279         */
1280        FFM_R_CTL(fpkt->pkt_cmd_fhdr.r_ctl, frm);
1281        FFM_D_ID(fpkt->pkt_cmd_fhdr.d_id, frm);
1282        FFM_S_ID(fpkt->pkt_cmd_fhdr.s_id, frm);
1283        FFM_TYPE(fpkt->pkt_cmd_fhdr.type, frm);
1284        FFM_F_CTL(0x980000, frm);
1285        FFM_OXID(xch->xch_oxid, frm);
1286        FFM_RXID(xch->xch_rxid, frm);

1288        switch (((uint8_t *)&fpkt->pkt_fca_rsvd1)[0]) {
1289        case LA_ELS_FLOGI:
1290                fcoei_fill_els_logi_resp(fpkt, frm);
1291                break;

1293        case LA_ELS_PLOGI:
1294                if (FRM2SS(frm)->ss_eport->eport_flags &
1295                    EPORT_FLAG_IS_DIRECT_P2P) {
1296                        FRM2SS(frm)->ss_p2p_info.fca_d_id = FRM_S_ID(frm);
1297                        FRM2SS(frm)->ss_p2p_info.d_id = FRM_D_ID(frm);
1298                }

1300                fcoei_fill_els_logi_resp(fpkt, frm);
1301                break;

1303        case LA_ELS_PRLI:
1304                fcoei_fill_els_prli_resp(fpkt, frm);
1305                break;

1307        case LA_ELS_ADISC:
1308                fcoei_fill_els_adisc_resp(fpkt, frm);
1309                break;

1311        case LA_ELS_LOGO:
1312                fcoei_fill_els_logo_resp(fpkt, frm);
1313                break;
1314        case LA_ELS_RSCN:
1315                fcoei_fill_els_acc_resp(fpkt, frm);
1316                break;

1318        default:
1319                fcoei_complete_xch(xch, frm, FC_PKT_FAILURE,
1320                    FC_REASON_CMD_UNSUPPORTED);
1321                return;
1322        }

1324        xch->xch_ss->ss_eport->eport_tx_frame(frm);
1325 }
```

_____***unchanged_portion_omitted_***

****
16803 Mon Jul 28 07:44:36 2014
new/usr/src/uts/common/io/fibre-channel/fca/oce/oce_rx.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____unchanged_portion_omitted_

```
 212 /*
 213  * function to free the RQ buffer
 214  *
 215  * rq - pointer to RQ structure
 216  * rqbd - pointer to recieve buffer descriptor
 217  *
 218  * return none
 219  */
 220 static inline void
 221 oce_rqb_free(struct oce_rq *rq, oce_rq_bdesc_t *rqbd)
 222 {
 223         uint32_t free_index;
 224         mutex_enter(&rq->rc_lock);
 225         free_index = rq->rqb_rc_head;
 226         rq->rqb_freelist[free_index] = rqbd;
 227         rq->rqb_rc_head = GET_Q_NEXT(free_index, 1, rq->cfg.nbufs);
 228         mutex_exit(&rq->rc_lock);
 229         atomic_inc_32(&rq->rqb_free);
 229         atomic_add_32(&rq->rqb_free, 1);
 230 } /* oce_rqb_free */
```

_____unchanged_portion_omitted_

```
 571 /*
 572  * function to free mblk databuffer to the RQ pool
 573  *
 574  * arg - pointer to the receive buffer descriptor
 575  *
 576  * return none
 577  */
 578 void
 579 oce_rx_pool_free(char *arg)
 580 {
 581         oce_rq_bdesc_t *rqbd;
 582         struct oce_rq  *rq;

 584         /* During destroy, arg will be NULL */
 585         if (arg == NULL) {
 586                 return;
 587         }

 589         /* retrieve the pointers from arg */
 590         rqbd = (oce_rq_bdesc_t *)(void *)arg;
 591         rq = rqbd->rq;
 592         rqbd->mp = desballoc((uchar_t *)rqbd->rqb->base,
 593             rqbd->rqb->size, 0, &rqbd->fr_rtn);

 595         if (rqbd->mp) {
 596                 rqbd->mp->b_rptr =
 597                     (uchar_t *)rqbd->rqb->base + OCE_RQE_BUF_HEADROOM;
 598         }

 600         oce_rqb_free(rq, rqbd);
 601         (void) atomic_dec_32(&rq->pending);
 601         (void) atomic_add_32(&rq->pending, -1);
 602 } /* rx_pool_free */
```

_____unchanged_portion_omitted_

```
**********************************************************
   71234 Mon Jul 28 07:44:36 2014
new/usr/src/uts/common/io/fssnap.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1423 /*
1424  * fssnap_translate() - helper function for fssnap_strategy()
1425  *
1426  *    performs the actual copy-on-write for write requests, if required.
1427  *    This function does the real work of the file system side of things.
1428  *
1429  *    It first checks the candidate bitmap to quickly determine whether any
1430  *    action is necessary.  If the candidate bitmap indicates the chunk was
1431  *    allocated when the snapshot was created, then it checks to see whether
1432  *    a translation already exists.  If a translation already exists then no
1433  *    action is required.  If the chunk is a candidate for copy-on-write,
1434  *    and a translation does not already exist, then the chunk is read in
1435  *    and a node is added to the translation table.
1436  *
1437  *    Once all of the chunks in the request range have been copied (if they
1438  *    needed to be), then the original request can be satisfied and the old
1439  *    data can be overwritten.
1440  */
1441 static int
1442 fssnap_translate(struct snapshot_id **sidpp, struct buf *wbp)
1443 {
1444         snapshot_id_t   *sidp = *sidpp;
1445         struct buf      *oldbp; /* buffer to store old data in */
1446         struct cow_info *cowp = sidp->sid_cowinfo;
1447         cow_map_t       *cmap = &cowp->cow_map;
1448         cow_map_node_t  *cmn;
1449         chunknumber_t   cowchunk, startchunk, endchunk;
1450         int             error;
1451         int     throttle_write = 0;

1453         /* make sure the snapshot is active */
1454         ASSERT(RW_READ_HELD(&sidp->sid_rwlock));

1456         startchunk = dbtocowchunk(cmap, wbp->b_lblkno);
1457         endchunk   = dbtocowchunk(cmap, wbp->b_lblkno +
1458             ((wbp->b_bcount-1) >> DEV_BSHIFT));

1460         /*
1461          * Do not throttle the writes of the fssnap taskq thread and
1462          * the log roll (trans_roll) thread. Furthermore the writes to
1463          * the on-disk log are also not subject to throttling.
1464          * The fssnap_write_taskq thread's write can block on the throttling
1465          * semaphore which leads to self-deadlock as this same thread
1466          * releases the throttling semaphore after completing the IO.
1467          * If the trans_roll thread's write is throttled then we can deadlock
1468          * because the fssnap_taskq_thread which releases the throttling
1469          * semaphore can block waiting for log space which can only be
1470          * released by the trans_roll thread.
1471          */

1473         throttle_write = !(taskq_member(cowp->cow_taskq, curthread) ||
1474             tsd_get(bypass_snapshot_throttle_key));

1476         /*
1477          * Iterate through all chunks covered by this write and perform the
1478          * copy-aside if necessary.  Once all chunks have been safely
1479          * stowed away, the new data may be written in a single sweep.
1480          *
1481          * For each chunk in the range, the following sequence is performed:
```



```
1482          *      - Is the chunk a candidate for translation?
1483          *              o If not, then no translation is necessary, continue
1484          *      - If it is a candidate, then does it already have a translation?
1485          *              o If so, then no translation is necessary, continue
1486          *      - If it is a candidate, but does not yet have a translation,
1487          *        then read the old data and schedule an asynchronous taskq
1488          *        to write the old data to the backing file.
1489          *
1490          * Once this has been performed over the entire range of chunks, then
1491          * it is safe to overwrite the data that is there.
1492          *
1493          * Note that no lock is required to check the candidate bitmap because
1494          * it never changes once the snapshot is created.  The reader lock is
1495          * taken to check the hastrans bitmap since it may change.  If it
1496          * turns out a copy is required, then the lock is upgraded to a
1497          * writer, and the bitmap is re-checked as it may have changed while
1498          * the lock was released.  Finally, the write lock is held while
1499          * reading the old data to make sure it is not translated out from
1500          * under us.
1501          *
1502          * This locking mechanism should be sufficient to handle multiple
1503          * threads writing to overlapping chunks simultaneously.
1504          */
1505         for (cowchunk = startchunk; cowchunk <= endchunk; cowchunk++) {
1506                 /*
1507                  * If the cowchunk is outside of the range of our
1508                  * candidate maps, then simply break out of the
1509                  * loop and pass the I/O through to bdev_strategy.
1510                  * This would occur if the file system has grown
1511                  * larger since the snapshot was taken.
1512                  */
1513                 if (cowchunk >= (cmap->cmap_bmsize * NBBY))
1514                         break;

1516                 /*
1517                  * If no disk blocks were allocated in this chunk when the
1518                  * snapshot was created then no copy-on-write will be
1519                  * required.  Since this bitmap is read-only no locks are
1520                  * necessary.
1521                  */
1522                 if (isclr(cmap->cmap_candidate, cowchunk)) {
1523                         continue;
1524                 }

1526                 /*
1527                  * If a translation already exists, the data can be written
1528                  * through since the old data has already been saved off.
1529                  */
1530                 if (isset(cmap->cmap_hastrans, cowchunk)) {
1531                         continue;
1532                 }


1535                 /*
1536                  * Throttle translations if there are too many outstanding
1537                  * chunks in memory.  The semaphore is sema_v'd by the taskq.
1538                  *
1539                  * You can't keep the sid_rwlock if you would go to sleep.
1540                  * This will result in deadlock when someone tries to delete
1541                  * the snapshot (wants the sid_rwlock as a writer, but can't
1542                  * get it).
1543                  */
1544                 if (throttle_write) {
1545                         if (sema_tryp(&cmap->cmap_throttle_sem) == 0) {
1546                                 rw_exit(&sidp->sid_rwlock);
1547                                 atomic_inc_32(&cmap->cmap_waiters);
```

```
1547                            atomic_add_32(&cmap->cmap_waiters, 1);
1548                            sema_p(&cmap->cmap_throttle_sem);
1549                            atomic_dec_32(&cmap->cmap_waiters);
1549                            atomic_add_32(&cmap->cmap_waiters, -1);
1550                            rw_enter(&sidp->sid_rwlock, RW_READER);

1552                    /*
1553                     * Now since we released the sid_rwlock the state may
1554                     * have transitioned underneath us. so check that again.
1555                     */
1556                            if (sidp != *sidpp || SID_INACTIVE(sidp)) {
1557                                    sema_v(&cmap->cmap_throttle_sem);
1558                                    return (ENXIO);
1559                            }
1560                    }
1561            }

1563             /*
1564              * Acquire the lock as a writer and check to see if a
1565              * translation has been added in the meantime.
1566              */
1567             rw_enter(&cmap->cmap_rwlock, RW_WRITER);
1568             if (isset(cmap->cmap_hastrans, cowchunk)) {
1569                     if (throttle_write)
1570                             sema_v(&cmap->cmap_throttle_sem);
1571                     rw_exit(&cmap->cmap_rwlock);
1572                     continue; /* go to the next chunk */
1573             }

1575             /*
1576              * read a full chunk of data from the requested offset rounded
1577              * down to the nearest chunk size.
1578              */
1579             oldbp = getrbuf(KM_SLEEP);
1580             oldbp->b_lblkno = cowchunktodb(cmap, cowchunk);
1581             oldbp->b_edev = wbp->b_edev;
1582             oldbp->b_bcount = cmap->cmap_chunksz;
1583             oldbp->b_bufsize = cmap->cmap_chunksz;
1584             oldbp->b_iodone = NULL;
1585             oldbp->b_proc = NULL;
1586             oldbp->b_flags = B_READ;
1587             oldbp->b_un.b_addr = kmem_alloc(cmap->cmap_chunksz, KM_SLEEP);

1589             (void) bdev_strategy(oldbp);
1590             (void) biowait(oldbp);

1592             /*
1593              * It's ok to bail in the middle of translating the range
1594              * because the extra copy-asides will not hurt anything
1595              * (except by using extra space in the backing store).
1596              */
1597             if ((error = geterror(oldbp)) != 0) {
1598                     cmn_err(CE_WARN, "fssnap_translate: error reading "
1599                         "old data for snapshot %d, chunk %llu, disk block "
1600                         "%lld, size %lu, error %d.", sidp->sid_snapnumber,
1601                         cowchunk, oldbp->b_lblkno, oldbp->b_bcount, error);
1602                     kmem_free(oldbp->b_un.b_addr, cmap->cmap_chunksz);
1603                     freerbuf(oldbp);
1604                     rw_exit(&cmap->cmap_rwlock);
1605                     if (throttle_write)
1606                             sema_v(&cmap->cmap_throttle_sem);
1607                     return (error);
1608             }

1610             /*
1611              * add the node to the translation table and save a reference
```



```
1612                      * to pass to the taskq for writing out to the backing file
1613                      */
1614                     cmn = transtbl_add(cmap, cowchunk, oldbp->b_un.b_addr);
1615                     freerbuf(oldbp);

1617                     /*
1618                      * Add a reference to the snapshot id so the lower level
1619                      * processing (ie. the taskq) can get back to the state
1620                      * information.
1621                      */
1622                     cmn->cmn_sid = sidp;
1623                     cmn->release_sem = throttle_write;
1624                     setbit(cmap->cmap_hastrans, cowchunk);

1626                     rw_exit(&cmap->cmap_rwlock);

1628                     /*
1629                      * schedule the asynchronous write to the backing file
1630                      */
1631                     if (cowp->cow_backfile_array != NULL)
1632                             (void) taskq_dispatch(cowp->cow_taskq,
1633                                 fssnap_write_taskq, cmn, TQ_SLEEP);
1634             }

1636             /*
1637              * Write new data in place of the old data.  At this point all of the
1638              * chunks touched by this write have been copied aside and so the new
1639              * data can be written out all at once.
1640              */
1641             (void) bdev_strategy(wbp);

1643             return (0);
1644     }

1646     /*
1647      * fssnap_write_taskq() - write in-memory translations to the backing file
1648      *
1649      *    writes in-memory translations to the backing file asynchronously.  A
1650      *    task is dispatched each time a new translation is created.  The task
1651      *    writes the data to the backing file and removes it from the memory
1652      *    list. The throttling semaphore is released only if the particular
1653      *    translation was throttled in fssnap_translate.
1654      */
1655     static void
1656     fssnap_write_taskq(void *arg)
1657     {
1658             cow_map_node_t  *cmn = (cow_map_node_t *)arg;
1659             snapshot_id_t   *sidp = cmn->cmn_sid;
1660             cow_info_t      *cowp = sidp->sid_cowinfo;
1661             cow_map_t       *cmap = &cowp->cow_map;
1662             int             error;
1663             int             bf_index;
1664             int             release_sem = cmn->release_sem;

1666             /*
1667              * The sid_rwlock does not need to be held here because the taskqs
1668              * are destroyed explicitly by fssnap_delete (with the sid_rwlock
1669              * held as a writer).  taskq_destroy() will flush all of the tasks
1670              * out before fssnap_delete frees up all of the structures.
1671              */

1673             /* if the snapshot was disabled from under us, drop the request. */
1674             rw_enter(&sidp->sid_rwlock, RW_READER);
1675             if (SID_INACTIVE(sidp)) {
1676                     rw_exit(&sidp->sid_rwlock);
1677                     if (release_sem)
```

```
1678                         sema_v(&cmap->cmap_throttle_sem);
1679                 return;
1680         }
1681         rw_exit(&sidp->sid_rwlock);

1683         atomic_inc_64((uint64_t *)&cmap->cmap_nchunks);
1683         atomic_add_64((uint64_t *)&cmap->cmap_nchunks, 1);

1685         if ((cmap->cmap_maxsize != 0) &&
1686             ((cmap->cmap_nchunks * cmap->cmap_chunksz) > cmap->cmap_maxsize)) {
1687                 cmn_err(CE_WARN, "fssnap_write_taskq: snapshot %d (%s) has "
1688                     "reached the maximum backing file size specified (%llu "
1689                     "bytes) and will be deleted.", sidp->sid_snapnumber,
1690                     (char *)cowp->cow_kstat_mntpt->ks_data,
1691                     cmap->cmap_maxsize);
1692                 if (release_sem)
1693                         sema_v(&cmap->cmap_throttle_sem);
1694                 atomic_or_uint(&sidp->sid_flags, SID_DELETE);
1695                 return;
1696         }

1698         /* perform the write */
1699         bf_index = cmn->cmn_chunk / cmap->cmap_chunksperbf;

1701         if (error = vn_rdwr(UIO_WRITE, (cowp->cow_backfile_array)[bf_index],
1702             cmn->cmn_buf, cmap->cmap_chunksz,
1703             (cmn->cmn_chunk % cmap->cmap_chunksperbf) * cmap->cmap_chunksz,
1704             UIO_SYSSPACE, 0, RLIM64_INFINITY, kcred, (ssize_t *)NULL)) {
1705                 cmn_err(CE_WARN, "fssnap_write_taskq: error writing to "
1706                     "backing file.  DELETING SNAPSHOT %d, backing file path "
1707                     "%s, offset %llu bytes, error %d.", sidp->sid_snapnumber,
1708                     (char *)cowp->cow_kstat_bfname->ks_data,
1709                     cmn->cmn_chunk * cmap->cmap_chunksz, error);
1710                 if (release_sem)
1711                         sema_v(&cmap->cmap_throttle_sem);
1712                 atomic_or_uint(&sidp->sid_flags, SID_DELETE);
1713                 return;
1714         }

1716         /*
1717          * now remove the node and buffer from memory
1718          */
1719         rw_enter(&cmap->cmap_rwlock, RW_WRITER);
1720         transtbl_delete(cmap, cmn);
1721         rw_exit(&cmap->cmap_rwlock);

1723         /* Allow more translations */
1724         if (release_sem)
1725                 sema_v(&cmap->cmap_throttle_sem);

1727 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  163883 Mon Jul 28 07:44:37 2014
new/usr/src/uts/common/io/gld.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1694 /*
1695  * gld_wput (q, mp)
1696  * general gld stream write put routine. Receives fastpath data from upper
1697  * modules and processes it immediately.  ioctl and M_PROTO/M_PCPROTO are
1698  * queued for later processing by the service procedure.
1699  */

1701 int
1702 gld_wput(queue_t *q, mblk_t *mp)
1703 {
1704         gld_t  *gld = (gld_t *)(q->q_ptr);
1705         int     rc;
1706         boolean_t multidata = B_TRUE;
1707         uint32_t upri;

1709 #ifdef GLD_DEBUG
1710         if (gld_debug & GLDTRACE)
1711                 cmn_err(CE_NOTE, "gld_wput(%p %p): type %x",
1712                     (void *)q, (void *)mp, DB_TYPE(mp));
1713 #endif
1714         switch (DB_TYPE(mp)) {

1716         case M_DATA:
1717                 /* fast data / raw support */
1718                 /* we must be DL_ATTACHED and DL_BOUND to do this */
1719                 /* Tricky to access memory without taking the mutex */
1720                 if ((gld->gld_flags & (GLD_RAW | GLD_FAST)) == 0 ||
1721                     gld->gld_state != DL_IDLE) {
1722                         merror(q, mp, EPROTO);
1723                         break;
1724                 }
1725                 /*
1726                  * Cleanup MBLK_VTAG in case it is set by other
1727                  * modules. MBLK_VTAG is used to save the vtag information.
1728                  */
1729                 GLD_CLEAR_MBLK_VTAG(mp);
1730                 multidata = B_FALSE;
1731                 /* LINTED: E_CASE_FALLTHRU */
1732         case M_MULTIDATA:
1733                 /* Only call gld_start() directly if nothing queued ahead */
1734                 /* No guarantees about ordering with different threads */
1735                 if (q->q_first)
1736                         goto use_wsrv;

1738                 /*
1739                  * This can happen if wsrv has taken off the last mblk but
1740                  * is still processing it.
1741                  */
1742                 membar_consumer();
1743                 if (gld->gld_in_wsrv)
1744                         goto use_wsrv;

1746                 /*
1747                  * Keep a count of current wput calls to start.
1748                  * Nonzero count delays any attempted DL_UNBIND.
1749                  * See comments above gld_start().
1750                  */
1751                 atomic_inc_32((uint32_t *)&gld->gld_wput_count);
1751                 atomic_add_32((uint32_t *)&gld->gld_wput_count, 1);
```



```
1752                 membar_enter();

1754                 /* Recheck state now wput_count is set to prevent DL_UNBIND */
1755                 /* If this Q is in process of DL_UNBIND, don't call start */
1756                 if (gld->gld_state != DL_IDLE || gld->gld_in_unbind) {
1757                         /* Extremely unlikely */
1758                         atomic_dec_32((uint32_t *)&gld->gld_wput_count);
1758                         atomic_add_32((uint32_t *)&gld->gld_wput_count, -1);
1759                         goto use_wsrv;
1760                 }

1762                 /*
1763                  * Get the priority value. Note that in raw mode, the
1764                  * per-packet priority value kept in b_band is ignored.
1765                  */
1766                 upri = (gld->gld_flags & GLD_RAW) ? gld->gld_upri :
1767                     UPRI(gld, mp->b_band);

1769                 rc = (multidata) ? gld_start_mdt(q, mp, GLD_WPUT) :
1770                     gld_start(q, mp, GLD_WPUT, upri);

1772                 /* Allow DL_UNBIND again */
1773                 membar_exit();
1774                 atomic_dec_32((uint32_t *)&gld->gld_wput_count);
1774                 atomic_add_32((uint32_t *)&gld->gld_wput_count, -1);

1776                 if (rc == GLD_NORESOURCES)
1777                         qenable(q);
1778                 break;  /*  Done with this packet */

1780 use_wsrv:
1781                 /* Q not empty, in DL_DETACH, or start gave NORESOURCES */
1782                 (void) putq(q, mp);
1783                 qenable(q);
1784                 break;

1786         case M_IOCTL:
1787                 /* ioctl relies on wsrv single threading per queue */
1788                 (void) putq(q, mp);
1789                 qenable(q);
1790                 break;

1792         case M_CTL:
1793                 (void) putq(q, mp);
1794                 qenable(q);
1795                 break;

1797         case M_FLUSH:           /* canonical flush handling */
1798                 /* XXX Should these be FLUSHALL? */
1799                 if (*mp->b_rptr & FLUSHW)
1800                         flushq(q, 0);
1801                 if (*mp->b_rptr & FLUSHR) {
1802                         flushq(RD(q), 0);
1803                         *mp->b_rptr &= ~FLUSHW;
1804                         qreply(q, mp);
1805                 } else
1806                         freemsg(mp);
1807                 break;

1809         case M_PROTO:
1810         case M_PCPROTO:
1811                 /* these rely on wsrv single threading per queue */
1812                 (void) putq(q, mp);
1813                 qenable(q);
1814                 break;
```

```
1816         default:
1817 #ifdef GLD_DEBUG
1818                 if (gld_debug & GLDETRACE)
1819                         cmn_err(CE_WARN,
1820                             "gld: Unexpected packet type from queue: 0x%x",
1821                             DB_TYPE(mp));
1822 #endif
1823                 freemsg(mp);
1824         }
1825         return (0);
1826 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  108637 Mon Jul 28 07:44:37 2014
new/usr/src/uts/common/io/hxge/hxge_rxdma.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1002 void
1003 hxge_freeb(p_rx_msg_t rx_msg_p)
1004 {
1005         size_t          size;
1006         uchar_t         *buffer = NULL;
1007         int             ref_cnt;
1008         boolean_t       free_state = B_FALSE;
1009         rx_rbr_ring_t   *ring = rx_msg_p->rx_rbr_p;

1011         HXGE_DEBUG_MSG((NULL, MEM2_CTL, "==> hxge_freeb"));
1012         HXGE_DEBUG_MSG((NULL, MEM2_CTL,
1013             "hxge_freeb:rx_msg_p = $%p (block pending %d)",
1014             rx_msg_p, hxge_mblks_pending));

1016         if (ring == NULL)
1017                 return;

1019         /*
1020          * This is to prevent posting activities while we are recovering
1021          * from fatal errors. This should not be a performance drag since
1022          * ref_cnt != 0 most times.
1023          */
1024         if (ring->rbr_state == RBR_POSTING)
1025                 MUTEX_ENTER(&ring->post_lock);

1027         /*
1028          * First we need to get the free state, then
1029          * atomic decrement the reference count to prevent
1030          * the race condition with the interrupt thread that
1031          * is processing a loaned up buffer block.
1032          */
1033         free_state = rx_msg_p->free;
1034         ref_cnt = atomic_dec_32_nv(&rx_msg_p->ref_cnt);
1034         ref_cnt = atomic_add_32_nv(&rx_msg_p->ref_cnt, -1);
1035         if (!ref_cnt) {
1036                 atomic_dec_32(&hxge_mblks_pending);

1038                 buffer = rx_msg_p->buffer;
1039                 size = rx_msg_p->block_size;

1041                 HXGE_DEBUG_MSG((NULL, MEM2_CTL, "hxge_freeb: "
1042                     "will free: rx_msg_p = $%p (block pending %d)",
1043                     rx_msg_p, hxge_mblks_pending));

1045                 if (!rx_msg_p->use_buf_pool) {
1046                         KMEM_FREE(buffer, size);
1047                 }

1049                 KMEM_FREE(rx_msg_p, sizeof (rx_msg_t));
1050                 /*
1051                  * Decrement the receive buffer ring's reference
1052                  * count, too.
1053                  */
1054                 atomic_dec_32(&ring->rbr_ref_cnt);

1056                 /*
1057                  * Free the receive buffer ring, iff
1058                  * 1. all the receive buffers have been freed
1059                  * 2. and we are in the proper state (that is,
```



```
1060                  *    we are not UNMAPPING).
1061                  */
1062                 if (ring->rbr_ref_cnt == 0 &&
1063                     ring->rbr_state == RBR_UNMAPPED) {
1064                         KMEM_FREE(ring, sizeof (*ring));
1065                         /* post_lock has been destroyed already */
1066                         return;
1067                 }
1068         }

1070         /*
1071          * Repost buffer.
1072          */
1073         if (free_state && (ref_cnt == 1)) {
1074                 HXGE_DEBUG_MSG((NULL, RX_CTL,
1075                     "hxge_freeb: post page $%p:", rx_msg_p));
1076                 if (ring->rbr_state == RBR_POSTING)
1077                         hxge_post_page(rx_msg_p->hxgep, ring, rx_msg_p);
1078         }

1080         if (ring->rbr_state == RBR_POSTING)
1081                 MUTEX_EXIT(&ring->post_lock);

1083         HXGE_DEBUG_MSG((NULL, MEM2_CTL, "<== hxge_freeb"));
1084 }
______unchanged_portion_omitted_
```

****
274020 Mon Jul 28 07:44:37 2014
new/usr/src/uts/common/io/ib/clients/daplt/daplt.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____unchanged_portion_omitted_

```
8270 /*
8271  * IBTF wrappers used for resource accounting
8272  */
8273 static ibt_status_t
8274 daplka_ibt_alloc_rc_channel(daplka_ep_resource_t *ep_rp, ibt_hca_hdl_t hca_hdl,
8275     ibt_chan_alloc_flags_t flags, ibt_rc_chan_alloc_args_t *args,
8276     ibt_channel_hdl_t *chan_hdl_p, ibt_chan_sizes_t *sizes)
8277 {
8278         daplka_hca_t    *hca_p;
8279         uint32_t        max_qps;
8280         boolean_t       acct_enabled;
8281         ibt_status_t    status;

8283         acct_enabled = daplka_accounting_enabled;
8284         hca_p = ep_rp->ep_hca;
8285         max_qps = daplka_max_qp_percent * hca_p->hca_attr.hca_max_chans / 100;

8287         if (acct_enabled) {
8288                 if (daplka_max_qp_percent != 0 &&
8289                     max_qps <= hca_p->hca_qp_count) {
8290                         DERR("ibt_alloc_rc_channel: resource limit exceeded "
8291                             "(limit %d, count %d)\n", max_qps,
8292                             hca_p->hca_qp_count);
8293                         return (IBT_INSUFF_RESOURCE);
8294                 }
8295                 DAPLKA_RS_ACCT_INC(ep_rp, 1);
8296                 atomic_inc_32(&hca_p->hca_qp_count);
8296                 atomic_add_32(&hca_p->hca_qp_count, 1);
8297         }
8298         status = ibt_alloc_rc_channel(hca_hdl, flags, args, chan_hdl_p, sizes);

8300         if (status != IBT_SUCCESS && acct_enabled) {
8301                 DAPLKA_RS_ACCT_DEC(ep_rp, 1);
8302                 atomic_dec_32(&hca_p->hca_qp_count);
8302                 atomic_add_32(&hca_p->hca_qp_count, -1);
8303         }
8304         return (status);
8305 }

8307 static ibt_status_t
8308 daplka_ibt_free_channel(daplka_ep_resource_t *ep_rp, ibt_channel_hdl_t chan_hdl)
8309 {
8310         daplka_hca_t    *hca_p;
8311         ibt_status_t    status;

8313         hca_p = ep_rp->ep_hca;

8315         status = ibt_free_channel(chan_hdl);
8316         if (status != IBT_SUCCESS) {
8317                 return (status);
8318         }
8319         if (DAPLKA_RS_ACCT_CHARGED(ep_rp) > 0) {
8320                 DAPLKA_RS_ACCT_DEC(ep_rp, 1);
8321                 atomic_dec_32(&hca_p->hca_qp_count);
8321                 atomic_add_32(&hca_p->hca_qp_count, -1);
8322         }
8323         return (status);
8324 }
```



```
8326 static ibt_status_t
8327 daplka_ibt_alloc_cq(daplka_evd_resource_t *evd_rp, ibt_hca_hdl_t hca_hdl,
8328     ibt_cq_attr_t *cq_attr, ibt_cq_hdl_t *ibt_cq_p, uint32_t *real_size)
8329 {
8330         daplka_hca_t    *hca_p;
8331         uint32_t        max_cqs;
8332         boolean_t       acct_enabled;
8333         ibt_status_t    status;

8335         acct_enabled = daplka_accounting_enabled;
8336         hca_p = evd_rp->evd_hca;
8337         max_cqs = daplka_max_cq_percent * hca_p->hca_attr.hca_max_cq / 100;

8339         if (acct_enabled) {
8340                 if (daplka_max_cq_percent != 0 &&
8341                     max_cqs <= hca_p->hca_cq_count) {
8342                         DERR("ibt_alloc_cq: resource limit exceeded "
8343                             "(limit %d, count %d)\n", max_cqs,
8344                             hca_p->hca_cq_count);
8345                         return (IBT_INSUFF_RESOURCE);
8346                 }
8347                 DAPLKA_RS_ACCT_INC(evd_rp, 1);
8348                 atomic_inc_32(&hca_p->hca_cq_count);
8348                 atomic_add_32(&hca_p->hca_cq_count, 1);
8349         }
8350         status = ibt_alloc_cq(hca_hdl, cq_attr, ibt_cq_p, real_size);

8352         if (status != IBT_SUCCESS && acct_enabled) {
8353                 DAPLKA_RS_ACCT_DEC(evd_rp, 1);
8354                 atomic_dec_32(&hca_p->hca_cq_count);
8354                 atomic_add_32(&hca_p->hca_cq_count, -1);
8355         }
8356         return (status);
8357 }

8359 static ibt_status_t
8360 daplka_ibt_free_cq(daplka_evd_resource_t *evd_rp, ibt_cq_hdl_t cq_hdl)
8361 {
8362         daplka_hca_t    *hca_p;
8363         ibt_status_t    status;

8365         hca_p = evd_rp->evd_hca;

8367         status = ibt_free_cq(cq_hdl);
8368         if (status != IBT_SUCCESS) {
8369                 return (status);
8370         }
8371         if (DAPLKA_RS_ACCT_CHARGED(evd_rp) > 0) {
8372                 DAPLKA_RS_ACCT_DEC(evd_rp, 1);
8373                 atomic_dec_32(&hca_p->hca_cq_count);
8373                 atomic_add_32(&hca_p->hca_cq_count, -1);
8374         }
8375         return (status);
8376 }

8378 static ibt_status_t
8379 daplka_ibt_alloc_pd(daplka_pd_resource_t *pd_rp, ibt_hca_hdl_t hca_hdl,
8380     ibt_pd_flags_t flags, ibt_pd_hdl_t *pd_hdl_p)
8381 {
8382         daplka_hca_t    *hca_p;
8383         uint32_t        max_pds;
8384         boolean_t       acct_enabled;
8385         ibt_status_t    status;

8387         acct_enabled = daplka_accounting_enabled;
8388         hca_p = pd_rp->pd_hca;
```

```
8389         max_pds = daplka_max_pd_percent * hca_p->hca_attr.hca_max_pd / 100;

8391         if (acct_enabled) {
8392                 if (daplka_max_pd_percent != 0 &&
8393                     max_pds <= hca_p->hca_pd_count) {
8394                         DERR("ibt_alloc_pd: resource limit exceeded "
8395                             "(limit %d, count %d)\n", max_pds,
8396                             hca_p->hca_pd_count);
8397                         return (IBT_INSUFF_RESOURCE);
8398                 }
8399                 DAPLKA_RS_ACCT_INC(pd_rp, 1);
8400                 atomic_inc_32(&hca_p->hca_pd_count);
8400                 atomic_add_32(&hca_p->hca_pd_count, 1);
8401         }
8402         status = ibt_alloc_pd(hca_hdl, flags, pd_hdl_p);

8404         if (status != IBT_SUCCESS && acct_enabled) {
8405                 DAPLKA_RS_ACCT_DEC(pd_rp, 1);
8406                 atomic_dec_32(&hca_p->hca_pd_count);
8406                 atomic_add_32(&hca_p->hca_pd_count, -1);
8407         }
8408         return (status);
8409 }

8411 static ibt_status_t
8412 daplka_ibt_free_pd(daplka_pd_resource_t *pd_rp, ibt_hca_hdl_t hca_hdl,
8413     ibt_pd_hdl_t pd_hdl)
8414 {
8415         daplka_hca_t    *hca_p;
8416         ibt_status_t    status;

8418         hca_p = pd_rp->pd_hca;

8420         status = ibt_free_pd(hca_hdl, pd_hdl);
8421         if (status != IBT_SUCCESS) {
8422                 return (status);
8423         }
8424         if (DAPLKA_RS_ACCT_CHARGED(pd_rp) > 0) {
8425                 DAPLKA_RS_ACCT_DEC(pd_rp, 1);
8426                 atomic_dec_32(&hca_p->hca_pd_count);
8426                 atomic_add_32(&hca_p->hca_pd_count, -1);
8427         }
8428         return (status);
8429 }

8431 static ibt_status_t
8432 daplka_ibt_alloc_mw(daplka_mw_resource_t *mw_rp, ibt_hca_hdl_t hca_hdl,
8433     ibt_pd_hdl_t pd_hdl, ibt_mw_flags_t flags, ibt_mw_hdl_t *mw_hdl_p,
8434     ibt_rkey_t *rkey_p)
8435 {
8436         daplka_hca_t    *hca_p;
8437         uint32_t        max_mws;
8438         boolean_t       acct_enabled;
8439         ibt_status_t    status;

8441         acct_enabled = daplka_accounting_enabled;
8442         hca_p = mw_rp->mw_hca;
8443         max_mws = daplka_max_mw_percent * hca_p->hca_attr.hca_max_mem_win / 100;

8445         if (acct_enabled) {
8446                 if (daplka_max_mw_percent != 0 &&
8447                     max_mws <= hca_p->hca_mw_count) {
8448                         DERR("ibt_alloc_mw: resource limit exceeded "
8449                             "(limit %d, count %d)\n", max_mws,
8450                             hca_p->hca_mw_count);
8451                         return (IBT_INSUFF_RESOURCE);
```



```
8452                 }
8453                 DAPLKA_RS_ACCT_INC(mw_rp, 1);
8454                 atomic_inc_32(&hca_p->hca_mw_count);
8454                 atomic_add_32(&hca_p->hca_mw_count, 1);
8455         }
8456         status = ibt_alloc_mw(hca_hdl, pd_hdl, flags, mw_hdl_p, rkey_p);

8458         if (status != IBT_SUCCESS && acct_enabled) {
8459                 DAPLKA_RS_ACCT_DEC(mw_rp, 1);
8460                 atomic_dec_32(&hca_p->hca_mw_count);
8460                 atomic_add_32(&hca_p->hca_mw_count, -1);
8461         }
8462         return (status);
8463 }

8465 static ibt_status_t
8466 daplka_ibt_free_mw(daplka_mw_resource_t *mw_rp, ibt_hca_hdl_t hca_hdl,
8467     ibt_mw_hdl_t mw_hdl)
8468 {
8469         daplka_hca_t    *hca_p;
8470         ibt_status_t    status;

8472         hca_p = mw_rp->mw_hca;

8474         status = ibt_free_mw(hca_hdl, mw_hdl);
8475         if (status != IBT_SUCCESS) {
8476                 return (status);
8477         }
8478         if (DAPLKA_RS_ACCT_CHARGED(mw_rp) > 0) {
8479                 DAPLKA_RS_ACCT_DEC(mw_rp, 1);
8480                 atomic_dec_32(&hca_p->hca_mw_count);
8480                 atomic_add_32(&hca_p->hca_mw_count, -1);
8481         }
8482         return (status);
8483 }

8485 static ibt_status_t
8486 daplka_ibt_register_mr(daplka_mr_resource_t *mr_rp, ibt_hca_hdl_t hca_hdl,
8487     ibt_pd_hdl_t pd_hdl, ibt_mr_attr_t *mr_attr, ibt_mr_hdl_t *mr_hdl_p,
8488     ibt_mr_desc_t *mr_desc_p)
8489 {
8490         daplka_hca_t    *hca_p;
8491         uint32_t        max_mrs;
8492         boolean_t       acct_enabled;
8493         ibt_status_t    status;

8495         acct_enabled = daplka_accounting_enabled;
8496         hca_p = mr_rp->mr_hca;
8497         max_mrs = daplka_max_mr_percent * hca_p->hca_attr.hca_max_memr / 100;

8499         if (acct_enabled) {
8500                 if (daplka_max_mr_percent != 0 &&
8501                     max_mrs <= hca_p->hca_mr_count) {
8502                         DERR("ibt_register_mr: resource limit exceeded "
8503                             "(limit %d, count %d)\n", max_mrs,
8504                             hca_p->hca_mr_count);
8505                         return (IBT_INSUFF_RESOURCE);
8506                 }
8507                 DAPLKA_RS_ACCT_INC(mr_rp, 1);
8508                 atomic_inc_32(&hca_p->hca_mr_count);
8508                 atomic_add_32(&hca_p->hca_mr_count, 1);
8509         }
8510         status = ibt_register_mr(hca_hdl, pd_hdl, mr_attr, mr_hdl_p, mr_desc_p);

8512         if (status != IBT_SUCCESS && acct_enabled) {
8513                 DAPLKA_RS_ACCT_DEC(mr_rp, 1);
```

```
8514                 atomic_dec_32(&hca_p->hca_mr_count);
8514                 atomic_add_32(&hca_p->hca_mr_count, -1);
8515         }
8516         return (status);
8517 }

8519 static ibt_status_t
8520 daplka_ibt_register_shared_mr(daplka_mr_resource_t *mr_rp,
8521     ibt_hca_hdl_t hca_hdl, ibt_mr_hdl_t mr_hdl, ibt_pd_hdl_t pd_hdl,
8522     ibt_smr_attr_t *smr_attr_p, ibt_mr_hdl_t *mr_hdl_p,
8523     ibt_mr_desc_t *mr_desc_p)
8524 {
8525         daplka_hca_t    *hca_p;
8526         uint32_t        max_mrs;
8527         boolean_t       acct_enabled;
8528         ibt_status_t    status;

8530         acct_enabled = daplka_accounting_enabled;
8531         hca_p = mr_rp->mr_hca;
8532         max_mrs = daplka_max_mr_percent * hca_p->hca_attr.hca_max_memr / 100;

8534         if (acct_enabled) {
8535                 if (daplka_max_mr_percent != 0 &&
8536                     max_mrs <= hca_p->hca_mr_count) {
8537                         DERR("ibt_register_shared_mr: resource limit exceeded "
8538                             "(limit %d, count %d)\n", max_mrs,
8539                             hca_p->hca_mr_count);
8540                         return (IBT_INSUFF_RESOURCE);
8541                 }
8542                 DAPLKA_RS_ACCT_INC(mr_rp, 1);
8543                 atomic_inc_32(&hca_p->hca_mr_count);
8543                 atomic_add_32(&hca_p->hca_mr_count, 1);
8544         }
8545         status = ibt_register_shared_mr(hca_hdl, mr_hdl, pd_hdl,
8546             smr_attr_p, mr_hdl_p, mr_desc_p);

8548         if (status != IBT_SUCCESS && acct_enabled) {
8549                 DAPLKA_RS_ACCT_DEC(mr_rp, 1);
8550                 atomic_dec_32(&hca_p->hca_mr_count);
8550                 atomic_add_32(&hca_p->hca_mr_count, -1);
8551         }
8552         return (status);
8553 }

8555 static ibt_status_t
8556 daplka_ibt_deregister_mr(daplka_mr_resource_t *mr_rp, ibt_hca_hdl_t hca_hdl,
8557     ibt_mr_hdl_t mr_hdl)
8558 {
8559         daplka_hca_t    *hca_p;
8560         ibt_status_t    status;

8562         hca_p = mr_rp->mr_hca;

8564         status = ibt_deregister_mr(hca_hdl, mr_hdl);
8565         if (status != IBT_SUCCESS) {
8566                 return (status);
8567         }
8568         if (DAPLKA_RS_ACCT_CHARGED(mr_rp) > 0) {
8569                 DAPLKA_RS_ACCT_DEC(mr_rp, 1);
8570                 atomic_dec_32(&hca_p->hca_mr_count);
8570                 atomic_add_32(&hca_p->hca_mr_count, -1);
8571         }
8572         return (status);
8573 }

8575 static ibt_status_t
```



```
8576 daplka_ibt_alloc_srq(daplka_srq_resource_t *srq_rp, ibt_hca_hdl_t hca_hdl,
8577     ibt_srq_flags_t flags, ibt_pd_hdl_t pd, ibt_srq_sizes_t *reqsz,
8578     ibt_srq_hdl_t *srq_hdl_p, ibt_srq_sizes_t *realsz)
8579 {
8580         daplka_hca_t    *hca_p;
8581         uint32_t        max_srqs;
8582         boolean_t       acct_enabled;
8583         ibt_status_t    status;

8585         acct_enabled = daplka_accounting_enabled;
8586         hca_p = srq_rp->srq_hca;
8587         max_srqs = daplka_max_srq_percent * hca_p->hca_attr.hca_max_srqs / 100;

8589         if (acct_enabled) {
8590                 if (daplka_max_srq_percent != 0 &&
8591                     max_srqs <= hca_p->hca_srq_count) {
8592                         DERR("ibt_alloc_srq: resource limit exceeded "
8593                             "(limit %d, count %d)\n", max_srqs,
8594                             hca_p->hca_srq_count);
8595                         return (IBT_INSUFF_RESOURCE);
8596                 }
8597                 DAPLKA_RS_ACCT_INC(srq_rp, 1);
8598                 atomic_inc_32(&hca_p->hca_srq_count);
8598                 atomic_add_32(&hca_p->hca_srq_count, 1);
8599         }
8600         status = ibt_alloc_srq(hca_hdl, flags, pd, reqsz, srq_hdl_p, realsz);

8602         if (status != IBT_SUCCESS && acct_enabled) {
8603                 DAPLKA_RS_ACCT_DEC(srq_rp, 1);
8604                 atomic_dec_32(&hca_p->hca_srq_count);
8604                 atomic_add_32(&hca_p->hca_srq_count, -1);
8605         }
8606         return (status);
8607 }

8609 static ibt_status_t
8610 daplka_ibt_free_srq(daplka_srq_resource_t *srq_rp, ibt_srq_hdl_t srq_hdl)
8611 {
8612         daplka_hca_t    *hca_p;
8613         ibt_status_t    status;

8615         hca_p = srq_rp->srq_hca;

8617         D3("ibt_free_srq: %p %p\n", srq_rp, srq_hdl);

8619         status = ibt_free_srq(srq_hdl);
8620         if (status != IBT_SUCCESS) {
8621                 return (status);
8622         }
8623         if (DAPLKA_RS_ACCT_CHARGED(srq_rp) > 0) {
8624                 DAPLKA_RS_ACCT_DEC(srq_rp, 1);
8625                 atomic_dec_32(&hca_p->hca_srq_count);
8625                 atomic_add_32(&hca_p->hca_srq_count, -1);
8626         }
8627         return (status);
8628 }
______unchanged_portion_omitted_

9024 /* ARGSUSED */
9025 static int
9026 daplka_close(dev_t dev, int flag, int otyp, struct cred *cred)
9027 {
9028         daplka_ia_resource_t    *ia_rp;
9029         minor_t                 rnum = getminor(dev);

9031         /*
```

```
9032          * Char only
9033          */
9034         if (otyp != OTYP_CHR) {
9035                 return (EINVAL);
9036         }
9037         D2("daplka_close: closing rnum = %d\n", rnum);
9038         atomic_inc_32(&daplka_pending_close);
9038         atomic_add_32(&daplka_pending_close, 1);

9040         /*
9041          * remove from resource table.
9042          */
9043         ia_rp = (daplka_ia_resource_t *)daplka_resource_remove(rnum);

9045         /*
9046          * remove the initial reference
9047          */
9048         if (ia_rp != NULL) {
9049                 DAPLKA_RS_UNREF(ia_rp);
9050         }
9051         atomic_dec_32(&daplka_pending_close);
9051         atomic_add_32(&daplka_pending_close, -1);
9052         return (DDI_SUCCESS);
9053 }
______unchanged_portion_omitted_

9840 /*
9841  * Generates a non-zero 32 bit hash key used for the timer hash table.
9842  */
9843 static uint32_t
9844 daplka_timer_hkey_gen()
9845 {
9846         uint32_t new_hkey;

9848         do {
9849                 new_hkey = atomic_inc_32_nv(&daplka_timer_hkey);
9849                 new_hkey = atomic_add_32_nv(&daplka_timer_hkey, 1);
9850         } while (new_hkey == 0);

9852         return (new_hkey);
9853 }
______unchanged_portion_omitted_
```

```
**********************************************************
   94448 Mon Jul 28 07:44:38 2014
new/usr/src/uts/common/io/ib/clients/ibd/ibd_cm.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1227 /*
1228  * Post a rwqe to the hardware and add it to the Rx list.
1229  */
1230 static int
1231 ibd_rc_post_srq(ibd_state_t *state, ibd_rwqe_t *rwqe)
1232 {
1233         /*
1234          * Here we should add dl_cnt before post recv, because
1235          * we would have to make sure dl_cnt is updated before
1236          * the corresponding ibd_rc_process_rx() is called.
1237          */
1238         ASSERT(state->rc_srq_rwqe_list.dl_cnt < state->rc_srq_size);
1239         atomic_inc_32(&state->rc_srq_rwqe_list.dl_cnt);
1239         atomic_add_32(&state->rc_srq_rwqe_list.dl_cnt, 1);
1240         if (ibt_post_srq(state->rc_srq_hdl, &rwqe->w_rwr, 1, NULL) !=
1241             IBT_SUCCESS) {
1242                 atomic_dec_32(&state->rc_srq_rwqe_list.dl_cnt);
1243                 DPRINT(40, "ibd_rc_post_srq : ibt_post_srq() failed");
1244                 return (DDI_FAILURE);
1245         }
1246
1247         return (DDI_SUCCESS);
1248 }
1249
1250 /*
1251  * Post a rwqe to the hardware and add it to the Rx list.
1252  */
1253 static int
1254 ibd_rc_post_rwqe(ibd_rc_chan_t *chan, ibd_rwqe_t *rwqe)
1255 {
1256         /*
1257          * Here we should add dl_cnt before post recv, because we would
1258          * have to make sure dl_cnt has already updated before
1259          * corresponding ibd_rc_process_rx() is called.
1260          */
1261         atomic_inc_32(&chan->rx_wqe_list.dl_cnt);
1261         atomic_add_32(&chan->rx_wqe_list.dl_cnt, 1);
1262         if (ibt_post_recv(chan->chan_hdl, &rwqe->w_rwr, 1, NULL) !=
1263             IBT_SUCCESS) {
1264                 atomic_dec_32(&chan->rx_wqe_list.dl_cnt);
1265                 DPRINT(40, "ibd_rc_post_rwqe : failed in ibt_post_recv()");
1266                 return (DDI_FAILURE);
1267         }
1268         return (DDI_SUCCESS);
1269 }
______unchanged_portion_omitted_

1449 /*
1450  * Processing to be done after receipt of a packet; hand off to GLD
1451  * in the format expected by GLD.
1452  */
1453 static void
1454 ibd_rc_process_rx(ibd_rc_chan_t *chan, ibd_rwqe_t *rwqe, ibt_wc_t *wc)
1455 {
1456         ibd_state_t *state = chan->state;
1457         ib_header_info_t *phdr;
1458         ipoib_hdr_t *ipibp;
1459         mblk_t *mp;
1460         mblk_t *mpc;
```



```
1461         int rxcnt;
1462         ip6_t *ip6h;
1463         int len;
1464
1465         /*
1466          * Track number handed to upper layer, and number still
1467          * available to receive packets.
1468          */
1469         if (state->rc_enable_srq) {
1470                 rxcnt = atomic_dec_32_nv(&state->rc_srq_rwqe_list.dl_cnt);
1471         } else {
1472                 rxcnt = atomic_dec_32_nv(&chan->rx_wqe_list.dl_cnt);
1473         }
1474
1475         /*
1476          * It can not be a IBA multicast packet.
1477          */
1478         ASSERT(!wc->wc_flags & IBT_WC_GRH_PRESENT);
1479
1480         /* For the connection reaper routine ibd_rc_conn_timeout_call() */
1481         chan->is_used = B_TRUE;
1482
1483 #ifdef DEBUG
1484         if (rxcnt < state->id_rc_rx_rwqe_thresh) {
1485                 state->rc_rwqe_short++;
1486         }
1487 #endif
1488
1489         /*
1490          * Possibly replenish the Rx pool if needed.
1491          */
1492         if ((rxcnt >= state->id_rc_rx_rwqe_thresh) &&
1493             (wc->wc_bytes_xfer > state->id_rc_rx_copy_thresh)) {
1494                 atomic_add_64(&state->rc_rcv_trans_byte, wc->wc_bytes_xfer);
1495                 atomic_inc_64(&state->rc_rcv_trans_pkt);
1496
1497                 /*
1498                  * Record how many rwqe has been occupied by upper
1499                  * network layer
1500                  */
1501                 if (state->rc_enable_srq) {
1502                         atomic_inc_32(
1503                             &state->rc_srq_rwqe_list.dl_bufs_outstanding);
1502                         atomic_add_32(&state->rc_srq_rwqe_list.
1503                             dl_bufs_outstanding, 1);
1504                 } else {
1505                         atomic_inc_32(&chan->rx_wqe_list.dl_bufs_outstanding);
1505                         atomic_add_32(&chan->rx_wqe_list.
1506                             dl_bufs_outstanding, 1);
1506                 }
1507                 mp = rwqe->rwqe_im_mblk;
1508         } else {
1509                 atomic_add_64(&state->rc_rcv_copy_byte, wc->wc_bytes_xfer);
1510                 atomic_inc_64(&state->rc_rcv_copy_pkt);
1511
1512                 if ((mp = allocb(wc->wc_bytes_xfer + IPOIB_GRH_SIZE,
1513                     BPRI_HI)) == NULL) {        /* no memory */
1514                         DPRINT(40, "ibd_rc_process_rx: allocb() failed");
1515                         state->rc_rcv_alloc_fail++;
1516                         if (state->rc_enable_srq) {
1517                                 if (ibd_rc_post_srq(state, rwqe) ==
1518                                     DDI_FAILURE) {
1519                                         ibd_rc_srq_free_rwqe(state, rwqe);
1520                                 }
1521                         } else {
1522                                 if (ibd_rc_post_rwqe(chan, rwqe) ==
```

```
1523                                    DDI_FAILURE) {
1524                                        ibd_rc_free_rwqe(chan, rwqe);
1525                                }
1526                        }
1527                        return;
1528                }

1530                bcopy(rwqe->rwqe_im_mblk->b_rptr + IPOIB_GRH_SIZE,
1531                    mp->b_wptr + IPOIB_GRH_SIZE, wc->wc_bytes_xfer);

1533                if (state->rc_enable_srq) {
1534                        if (ibd_rc_post_srq(state, rwqe) == DDI_FAILURE) {
1535                                ibd_rc_srq_free_rwqe(state, rwqe);
1536                        }
1537                } else {
1538                        if (ibd_rc_post_rwqe(chan, rwqe) == DDI_FAILURE) {
1539                                ibd_rc_free_rwqe(chan, rwqe);
1540                        }
1541                }
1542        }

1544        ipibp = (ipoib_hdr_t *)((uchar_t *)mp->b_rptr + IPOIB_GRH_SIZE);
1545        if (ntohs(ipibp->ipoib_type) == ETHERTYPE_IPV6) {
1546                ip6h = (ip6_t *)((uchar_t *)ipibp + sizeof (ipoib_hdr_t));
1547                len = ntohs(ip6h->ip6_plen);
1548                if (ip6h->ip6_nxt == IPPROTO_ICMPV6) {
1549                        /* LINTED: E_CONSTANT_CONDITION */
1550                        IBD_PAD_NSNA(ip6h, len, IBD_RECV);
1551                }
1552        }

1554        phdr = (ib_header_info_t *)mp->b_rptr;
1555        phdr->ib_grh.ipoib_vertcflow = 0;
1556        ovbcopy(&state->id_macaddr, &phdr->ib_dst,
1557            sizeof (ipoib_mac_t));
1558        mp->b_wptr = mp->b_rptr + wc->wc_bytes_xfer+ IPOIB_GRH_SIZE;

1560        /*
1561         * Can RC mode in IB guarantee its checksum correctness?
1562         *
1563         *      (void) hcksum_assoc(mp, NULL, NULL, 0, 0, 0, 0,
1564         *          HCK_FULLCKSUM | HCK_FULLCKSUM_OK, 0);
1565         */

1567        /*
1568         * Make sure this is NULL or we're in trouble.
1569         */
1570        if (mp->b_next != NULL) {
1571                ibd_print_warn(state,
1572                    "ibd_rc_process_rx: got duplicate mp from rcq?");
1573                mp->b_next = NULL;
1574        }

1576        /*
1577         * Add this mp to the list of processed mp's to send to
1578         * the nw layer
1579         */
1580        if (state->rc_enable_srq) {
1581                mutex_enter(&state->rc_rx_lock);
1582                if (state->rc_rx_mp) {
1583                        ASSERT(state->rc_rx_mp_tail != NULL);
1584                        state->rc_rx_mp_tail->b_next = mp;
1585                } else {
1586                        ASSERT(state->rc_rx_mp_tail == NULL);
1587                        state->rc_rx_mp = mp;
1588                }
```



```
1590                state->rc_rx_mp_tail = mp;
1591                state->rc_rx_mp_len++;

1593                if (state->rc_rx_mp_len  >= IBD_MAX_RX_MP_LEN) {
1594                        mpc = state->rc_rx_mp;

1596                        state->rc_rx_mp = NULL;
1597                        state->rc_rx_mp_tail = NULL;
1598                        state->rc_rx_mp_len = 0;
1599                        mutex_exit(&state->rc_rx_lock);
1600                        mac_rx(state->id_mh, NULL, mpc);
1601                } else {
1602                        mutex_exit(&state->rc_rx_lock);
1603                }
1604        } else {
1605                mutex_enter(&chan->rx_lock);
1606                if (chan->rx_mp) {
1607                        ASSERT(chan->rx_mp_tail != NULL);
1608                        chan->rx_mp_tail->b_next = mp;
1609                } else {
1610                        ASSERT(chan->rx_mp_tail == NULL);
1611                        chan->rx_mp = mp;
1612                }

1614                chan->rx_mp_tail = mp;
1615                chan->rx_mp_len++;

1617                if (chan->rx_mp_len  >= IBD_MAX_RX_MP_LEN) {
1618                        mpc = chan->rx_mp;

1620                        chan->rx_mp = NULL;
1621                        chan->rx_mp_tail = NULL;
1622                        chan->rx_mp_len = 0;
1623                        mutex_exit(&chan->rx_lock);
1624                        mac_rx(state->id_mh, NULL, mpc);
1625                } else {
1626                        mutex_exit(&chan->rx_lock);
1627                }
1628        }
1629 }

1631 /*
1632  * Callback code invoked from STREAMs when the recv data buffer is free
1633  * for recycling.
1634  */
1635 static void
1636 ibd_rc_freemsg_cb(char *arg)
1637 {
1638        ibd_rwqe_t *rwqe = (ibd_rwqe_t *)arg;
1639        ibd_rc_chan_t *chan = rwqe->w_chan;
1640        ibd_state_t *state = rwqe->w_state;

1642        /*
1643         * If the wqe is being destructed, do not attempt recycling.
1644         */
1645        if (rwqe->w_freeing_wqe == B_TRUE) {
1646                return;
1647        }

1649        ASSERT(!state->rc_enable_srq);
1650        ASSERT(chan->rx_wqe_list.dl_cnt < chan->rcq_size);

1652        rwqe->rwqe_im_mblk = desballoc(rwqe->rwqe_copybuf.ic_bufaddr,
1653            state->rc_mtu + IPOIB_GRH_SIZE, 0, &rwqe->w_freemsg_cb);
1654        if (rwqe->rwqe_im_mblk == NULL) {
```

```
1655                 DPRINT(40, "ibd_rc_freemsg_cb: desballoc() failed");
1656                 ibd_rc_free_rwqe(chan, rwqe);
1657                 return;
1658         }

1660         /*
1661          * Post back to h/w. We could actually have more than
1662          * id_num_rwqe WQEs on the list if there were multiple
1663          * ibd_freemsg_cb() calls outstanding (since the lock is
1664          * not held the entire time). This will start getting
1665          * corrected over subsequent ibd_freemsg_cb() calls.
1666          */
1667         if (ibd_rc_post_rwqe(chan, rwqe) == DDI_FAILURE) {
1668                 ibd_rc_free_rwqe(chan, rwqe);
1669                 return;
1670         }
1671         atomic_dec_32(&chan->rx_wqe_list.dl_bufs_outstanding);
1672         atomic_add_32(&chan->rx_wqe_list.dl_bufs_outstanding, -1);
1672 }
```

_____***unchanged_portion_omitted_***

****
14599 Mon Jul 28 07:44:38 2014
new/usr/src/uts/common/io/ib/clients/rdsv3/cong.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
 254 void
 255 rdsv3_cong_map_updated(struct rdsv3_cong_map *map, uint64_t portmask)
 256 {
 257         RDSV3_DPRINTF4("rdsv3_cong_map_updated",
 258             "waking map %p for %u.%u.%u.%u",
 259             map, NIPQUAD(map->m_addr));

 261         rdsv3_stats_inc(s_cong_update_received);
 262         atomic_inc_32(&rdsv3_cong_generation);
 262         atomic_add_32(&rdsv3_cong_generation, 1);
 263 #if 0
 264 XXX
 265         if (waitqueue_active(&map->m_waitq))
 266 #endif
 267                 rdsv3_wake_up(&map->m_waitq);

 269         if (portmask && !list_is_empty(&rdsv3_cong_monitor)) {
 270                 struct rdsv3_sock *rs;

 272                 rw_enter(&rdsv3_cong_monitor_lock, RW_READER);
 273                 RDSV3_FOR_EACH_LIST_NODE(rs, &rdsv3_cong_monitor,
 274                     rs_cong_list) {
 275                         mutex_enter(&rs->rs_lock);
 276                         rs->rs_cong_notify |= (rs->rs_cong_mask & portmask);
 277                         rs->rs_cong_mask &= ~portmask;
 278                         mutex_exit(&rs->rs_lock);
 279                         if (rs->rs_cong_notify)
 280                                 rdsv3_wake_sk_sleep(rs);
 281                 }
 282                 rw_exit(&rdsv3_cong_monitor_lock);
 283         }

 285         RDSV3_DPRINTF4("rdsv3_cong_map_updated", "Return(map: %p)", map);
 286 }
```

_____***unchanged_portion_omitted_***

****
25391 Mon Jul 28 07:44:38 2014
new/usr/src/uts/common/io/ib/clients/rdsv3/ib_recv.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
109 extern int atomic_add_unless(atomic_t *, uint_t, ulong_t);

111 static int
112 rdsv3_ib_recv_refill_one(struct rdsv3_connection *conn,
113     struct rdsv3_ib_recv_work *recv)
114 {
115         struct rdsv3_ib_connection *ic = conn->c_transport_data;
116         ibt_mi_hdl_t mi_hdl;
117         ibt_iov_attr_t iov_attr;
118         ibt_iov_t iov_arr[1];

120         RDSV3_DPRINTF5("rdsv3_ib_recv_refill_one", "conn: %p, recv: %p",
121             conn, recv);

123         if (!recv->r_ibinc) {
124                 if (!atomic_add_unless(&rdsv3_ib_allocation, 1,
125                     ic->i_max_recv_alloc)) {
126                         rdsv3_ib_stats_inc(s_ib_rx_alloc_limit);
127                         goto out;
128                 }
129                 recv->r_ibinc = kmem_cache_alloc(rdsv3_ib_incoming_slab,
130                     KM_NOSLEEP);
131                 if (recv->r_ibinc == NULL) {
132                         atomic_dec_32(&rdsv3_ib_allocation);
132                         atomic_add_32(&rdsv3_ib_allocation, -1);
133                         goto out;
134                 }
135                 rdsv3_inc_init(&recv->r_ibinc->ii_inc, conn, conn->c_faddr);
136                 recv->r_ibinc->ii_ibdev = ic->rds_ibdev;
137                 recv->r_ibinc->ii_pool = ic->rds_ibdev->inc_pool;
138         }

140         if (!recv->r_frag) {
141                 recv->r_frag = kmem_cache_alloc(ic->rds_ibdev->ib_frag_slab,
142                     KM_NOSLEEP);
143                 if (!recv->r_frag)
144                         goto out;
145         }

147         /* Data sge, structure copy */
148         recv->r_sge[1] = recv->r_frag->f_sge;

150         RDSV3_DPRINTF5("rdsv3_ib_recv_refill_one", "Return: conn: %p, recv: %p",
151             conn, recv);

153         return (0);
154 out:
155         if (recv->r_ibinc) {
156                 kmem_cache_free(rdsv3_ib_incoming_slab, recv->r_ibinc);
157                 atomic_dec_32(&rdsv3_ib_allocation);
157                 atomic_add_32(&rdsv3_ib_allocation, -1);
158                 recv->r_ibinc = NULL;
159         }
160         return (-ENOMEM);
161 }
```

_____***unchanged_portion_omitted_***

****
11301 Mon Jul 28 07:44:38 2014
new/usr/src/uts/common/io/ib/clients/rdsv3/message.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****

_____***unchanged_portion_omitted_***

```
  64 #endif

  66 void
  67 rdsv3_message_addref(struct rdsv3_message *rm)
  68 {
  69         RDSV3_DPRINTF5("rdsv3_message_addref", "addref rm %p ref %d",
  70             rm, atomic_get(&rm->m_refcount));
  71         atomic_inc_32(&rm->m_refcount);
  71         atomic_add_32(&rm->m_refcount, 1);
  72 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   17760 Mon Jul 28 07:44:39 2014
new/usr/src/uts/common/io/ib/clients/rdsv3/rdma.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

  84 static struct rdsv3_mr *
  85 rdsv3_mr_tree_walk(struct avl_tree *root, uint32_t key,
  86         struct rdsv3_mr *insert)
  87 {
  88         struct rdsv3_mr *mr;
  89         avl_index_t where;

  91         mr = avl_find(root, &key, &where);
  92         if ((mr == NULL) && (insert != NULL)) {
  93                 avl_insert(root, (void *)insert, where);
  94                 atomic_inc_32(&insert->r_refcount);
  94                 atomic_add_32(&insert->r_refcount, 1);
  95                 return (NULL);
  96         }

  98         return (mr);
  99 }
______unchanged_portion_omitted_

 166 static int
 167 __rdsv3_rdma_map(struct rdsv3_sock *rs, struct rds_get_mr_args *args,
 168         uint64_t *cookie_ret, struct rdsv3_mr **mr_ret)
 169 {
 170         struct rdsv3_mr *mr = NULL, *found;
 171         void *trans_private;
 172         rds_rdma_cookie_t cookie;
 173         unsigned int nents = 0;
 174         int ret;

 176         if (rs->rs_bound_addr == 0) {
 177                 ret = -ENOTCONN; /* XXX not a great errno */
 178                 goto out;
 179         }

 181         if (!rs->rs_transport->get_mr) {
 182                 ret = -EOPNOTSUPP;
 183                 goto out;
 184         }

 186         mr = kmem_zalloc(sizeof (struct rdsv3_mr), KM_NOSLEEP);
 187         if (!mr) {
 188                 ret = -ENOMEM;
 189                 goto out;
 190         }

 192         mr->r_refcount = 1;
 193         RB_CLEAR_NODE(&mr->r_rb_node);
 194         mr->r_trans = rs->rs_transport;
 195         mr->r_sock = rs;

 197         if (args->flags & RDS_RDMA_USE_ONCE)
 198                 mr->r_use_once = 1;
 199         if (args->flags & RDS_RDMA_INVALIDATE)
 200                 mr->r_invalidate = 1;
 201         if (args->flags & RDS_RDMA_READWRITE)
 202                 mr->r_write = 1;

 204         /*
 205          * Obtain a transport specific MR. If this succeeds, the
```



```
 206          * s/g list is now owned by the MR.
 207          * Note that dma_map() implies that pending writes are
 208          * flushed to RAM, so no dma_sync is needed here.
 209          */
 210         trans_private = rs->rs_transport->get_mr(&args->vec, nents, rs,
 211             &mr->r_key);

 213         if (IS_ERR(trans_private)) {
 214                 ret = PTR_ERR(trans_private);
 215                 goto out;
 216         }

 218         mr->r_trans_private = trans_private;

 220         /*
 221          * The user may pass us an unaligned address, but we can only
 222          * map page aligned regions. So we keep the offset, and build
 223          * a 64bit cookie containing <R_Key, offset> and pass that
 224          * around.
 225          */
 226         cookie = rdsv3_rdma_make_cookie(mr->r_key, args->vec.addr & ~PAGEMASK);
 227         if (cookie_ret)
 228                 *cookie_ret = cookie;

 230         /*
 231          * copy value of cookie to user address at args->cookie_addr
 232          */
 233         if (args->cookie_addr) {
 234                 ret = ddi_copyout((void *)&cookie,
 235                     (void *)((intptr_t)args->cookie_addr),
 236                     sizeof (rds_rdma_cookie_t), 0);
 237                 if (ret != 0) {
 238                         ret = -EFAULT;
 239                         goto out;
 240                 }
 241         }

 243         RDSV3_DPRINTF5("__rdsv3_rdma_map",
 244             "RDS: get_mr mr 0x%p addr 0x%llx key 0x%x",
 245             mr, args->vec.addr, mr->r_key);
 246         /*
 247          * Inserting the new MR into the rbtree bumps its
 248          * reference count.
 249          */
 250         mutex_enter(&rs->rs_rdma_lock);
 251         found = rdsv3_mr_tree_walk(&rs->rs_rdma_keys, mr->r_key, mr);
 252         mutex_exit(&rs->rs_rdma_lock);

 254         ASSERT(!(found && found != mr));

 256         if (mr_ret) {
 257                 atomic_inc_32(&mr->r_refcount);
 257                 atomic_add_32(&mr->r_refcount, 1);
 258                 *mr_ret = mr;
 259         }

 261         ret = 0;
 262 out:
 263         if (mr)
 264                 rdsv3_mr_put(mr);
 265         return (ret);
 266 }
______unchanged_portion_omitted_

 375 /*
 376  * This is called when we receive an extension header that
```

```
 377  * tells us this MR was used. It allows us to implement
 378  * use_once semantics
 379  */
 380 void
 381 rdsv3_rdma_unuse(struct rdsv3_sock *rs, uint32_t r_key, int force)
 382 {
 383         struct rdsv3_mr *mr;
 384         int zot_me = 0;

 386         RDSV3_DPRINTF4("rdsv3_rdma_unuse", "Enter rkey: 0x%x", r_key);

 388         mutex_enter(&rs->rs_rdma_lock);
 389         mr = rdsv3_mr_tree_walk(&rs->rs_rdma_keys, r_key, NULL);
 390         if (!mr) {
 391                 RDSV3_DPRINTF4("rdsv3_rdma_unuse",
 392                     "rdsv3: trying to unuse MR with unknown r_key %u!", r_key);
 393                 mutex_exit(&rs->rs_rdma_lock);
 394                 return;
 395         }

 397         if (mr->r_use_once || force) {
 398                 avl_remove(&rs->rs_rdma_keys, &mr->r_rb_node);
 399                 RB_CLEAR_NODE(&mr->r_rb_node);
 400                 zot_me = 1;
 401         } else {
 402                 atomic_inc_32(&mr->r_refcount);
 402                 atomic_add_32(&mr->r_refcount, 1);
 403         }
 404         mutex_exit(&rs->rs_rdma_lock);

 406         /*
 407          * May have to issue a dma_sync on this memory region.
 408          * Note we could avoid this if the operation was a RDMA READ,
 409          * but at this point we can't tell.
 410          */
 411         if (mr->r_trans->sync_mr)
 412                 mr->r_trans->sync_mr(mr->r_trans_private, DMA_FROM_DEVICE);

 414         /*
 415          * If the MR was marked as invalidate, this will
 416          * trigger an async flush.
 417          */
 418         if (zot_me)
 419                 rdsv3_destroy_mr(mr);
 420         rdsv3_mr_put(mr);
 421         RDSV3_DPRINTF4("rdsv3_rdma_unuse", "Return");
 422 }
______unchanged_portion_omitted_

 609 /*
 610  * The application wants us to pass an RDMA destination (aka MR)
 611  * to the remote
 612  */
 613 int
 614 rdsv3_cmsg_rdma_dest(struct rdsv3_sock *rs, struct rdsv3_message *rm,
 615         struct cmsghdr *cmsg)
 616 {
 617         struct rdsv3_mr *mr;
 618         uint32_t r_key;
 619         int err = 0;

 621         if (cmsg->cmsg_len != CMSG_LEN(sizeof (rds_rdma_cookie_t)) ||
 622             rm->m_rdma_cookie != 0)
 623                 return (-EINVAL);

 625         (void) memcpy(&rm->m_rdma_cookie, CMSG_DATA(cmsg),
```



```
 626             sizeof (rm->m_rdma_cookie));

 628         /*
 629          * We are reusing a previously mapped MR here. Most likely, the
 630          * application has written to the buffer, so we need to explicitly
 631          * flush those writes to RAM. Otherwise the HCA may not see them
 632          * when doing a DMA from that buffer.
 633          */
 634         r_key = rdsv3_rdma_cookie_key(rm->m_rdma_cookie);

 636         mutex_enter(&rs->rs_rdma_lock);
 637         mr = rdsv3_mr_tree_walk(&rs->rs_rdma_keys, r_key, NULL);
 638         if (!mr)
 639                 err = -EINVAL;  /* invalid r_key */
 640         else
 641                 atomic_inc_32(&mr->r_refcount);
 641                 atomic_add_32(&mr->r_refcount, 1);
 642         mutex_exit(&rs->rs_rdma_lock);

 644         if (mr) {
 645                 mr->r_trans->sync_mr(mr->r_trans_private, DMA_TO_DEVICE);
 646                 rm->m_rdma_mr = mr;
 647         }
 648         return (err);
 649 }
______unchanged_portion_omitted_
```

****
18689 Mon Jul 28 07:44:39 2014
new/usr/src/uts/common/io/ib/clients/rdsv3/rds_recv.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
 62 void
 63 rdsv3_inc_addref(struct rdsv3_incoming *inc)
 64 {
 65         RDSV3_DPRINTF4("rdsv3_inc_addref",
 66             "addref inc %p ref %d", inc, atomic_get(&inc->i_refcount));
 67         atomic_inc_32(&inc->i_refcount);
 67         atomic_add_32(&inc->i_refcount, 1);
 68 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   32863 Mon Jul 28 07:44:39 2014
new/usr/src/uts/common/io/ib/clients/rdsv3/send.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 131 /*
 132  * We're making the concious trade-off here to only send one message
 133  * down the connection at a time.
 134  *   Pro:
 135  *      - tx queueing is a simple fifo list
 136  *      - reassembly is optional and easily done by transports per conn
 137  *      - no per flow rx lookup at all, straight to the socket
 138  *      - less per-frag memory and wire overhead
 139  *   Con:
 140  *      - queued acks can be delayed behind large messages
 141  *   Depends:
 142  *      - small message latency is higher behind queued large messages
 143  *      - large message latency isn't starved by intervening small sends
 144  */
 145 int
 146 rdsv3_send_xmit(struct rdsv3_connection *conn)
 147 {
 148         struct rdsv3_message *rm;
 149         unsigned int tmp;
 150         unsigned int send_quota = send_batch_count;
 151         struct rdsv3_scatterlist *sg;
 152         int ret = 0;
 153         int was_empty = 0;
 154         list_t to_be_dropped;

 156 restart:
 157         if (!rdsv3_conn_up(conn))
 158                 goto out;

 160         RDSV3_DPRINTF4("rdsv3_send_xmit", "Enter(conn: %p)", conn);

 162         list_create(&to_be_dropped, sizeof (struct rdsv3_message),
 163             offsetof(struct rdsv3_message, m_conn_item));

 165         /*
 166          * sendmsg calls here after having queued its message on the send
 167          * queue.  We only have one task feeding the connection at a time.  If
 168          * another thread is already feeding the queue then we back off.  This
 169          * avoids blocking the caller and trading per-connection data between
 170          * caches per message.
 171          */
 172         if (!mutex_tryenter(&conn->c_send_lock)) {
 173                 RDSV3_DPRINTF4("rdsv3_send_xmit",
 174                     "Another thread running(conn: %p)", conn);
 175                 rdsv3_stats_inc(s_send_sem_contention);
 176                 ret = -ENOMEM;
 177                 goto out;
 178         }
 179         atomic_inc_32(&conn->c_senders);
 179         atomic_add_32(&conn->c_senders, 1);

 181         if (conn->c_trans->xmit_prepare)
 182                 conn->c_trans->xmit_prepare(conn);

 184         /*
 185          * spin trying to push headers and data down the connection until
 186          * the connection doesn't make forward progress.
 187          */
 188         while (--send_quota) {
```



```
 189                 /*
 190                  * See if need to send a congestion map update if we're
 191                  * between sending messages.  The send_sem protects our sole
 192                  * use of c_map_offset and _bytes.
 193                  * Note this is used only by transports that define a special
 194                  * xmit_cong_map function. For all others, we create allocate
 195                  * a cong_map message and treat it just like any other send.
 196                  */
 197                 if (conn->c_map_bytes) {
 198                         ret = conn->c_trans->xmit_cong_map(conn, conn->c_lcong,
 199                             conn->c_map_offset);
 200                         if (ret <= 0)
 201                                 break;

 203                         conn->c_map_offset += ret;
 204                         conn->c_map_bytes -= ret;
 205                         if (conn->c_map_bytes)
 206                                 continue;
 207                 }

 209                 /*
 210                  * If we're done sending the current message, clear the
 211                  * offset and S/G temporaries.
 212                  */
 213                 rm = conn->c_xmit_rm;
 214                 if (rm != NULL &&
 215                     conn->c_xmit_hdr_off == sizeof (struct rdsv3_header) &&
 216                     conn->c_xmit_sg == rm->m_nents) {
 217                         conn->c_xmit_rm = NULL;
 218                         conn->c_xmit_sg = 0;
 219                         conn->c_xmit_hdr_off = 0;
 220                         conn->c_xmit_data_off = 0;
 221                         conn->c_xmit_rdma_sent = 0;

 223                         /* Release the reference to the previous message. */
 224                         rdsv3_message_put(rm);
 225                         rm = NULL;
 226                 }

 228                 /* If we're asked to send a cong map update, do so. */
 229                 if (rm == NULL && test_and_clear_bit(0, &conn->c_map_queued)) {
 230                         if (conn->c_trans->xmit_cong_map != NULL) {
 231                                 conn->c_map_offset = 0;
 232                                 conn->c_map_bytes =
 233                                     sizeof (struct rdsv3_header) +
 234                                     RDSV3_CONG_MAP_BYTES;
 235                                 continue;
 236                         }

 238                         rm = rdsv3_cong_update_alloc(conn);
 239                         if (IS_ERR(rm)) {
 240                                 ret = PTR_ERR(rm);
 241                                 break;
 242                         }

 244                         conn->c_xmit_rm = rm;
 245                 }

 247                 /*
 248                  * Grab the next message from the send queue, if there is one.
 249                  *
 250                  * c_xmit_rm holds a ref while we're sending this message down
 251                  * the connction.  We can use this ref while holding the
 252                  * send_sem.. rdsv3_send_reset() is serialized with it.
 253                  */
 254                 if (rm == NULL) {
```

```
255                         unsigned int len;

257                         mutex_enter(&conn->c_lock);

259                         if (!list_is_empty(&conn->c_send_queue)) {
260                                 rm = list_remove_head(&conn->c_send_queue);
261                                 rdsv3_message_addref(rm);

263                                 /*
264                                  * Move the message from the send queue to
265                                  * the retransmit
266                                  * list right away.
267                                  */
268                                 list_insert_tail(&conn->c_retrans, rm);
269                         }

271                         mutex_exit(&conn->c_lock);

273                         if (rm == NULL) {
274                                 was_empty = 1;
275                                 break;
276                         }

278                         /*
279                          * Unfortunately, the way Infiniband deals with
280                          * RDMA to a bad MR key is by moving the entire
281                          * queue pair to error state. We cold possibly
282                          * recover from that, but right now we drop the
283                          * connection.
284                          * Therefore, we never retransmit messages with
285                          * RDMA ops.
286                          */
287                         if (rm->m_rdma_op &&
288                             test_bit(RDSV3_MSG_RETRANSMITTED, &rm->m_flags)) {
289                                 mutex_enter(&conn->c_lock);
290                                 if (test_and_clear_bit(RDSV3_MSG_ON_CONN,
291                                     &rm->m_flags))
292                                         list_remove_node(&rm->m_conn_item);
293                                         list_insert_tail(&to_be_dropped, rm);
294                                 mutex_exit(&conn->c_lock);
295                                 rdsv3_message_put(rm);
296                                 continue;
297                         }

299                         /* Require an ACK every once in a while */
300                         len = ntohl(rm->m_inc.i_hdr.h_len);
301                         if (conn->c_unacked_packets == 0 ||
302                             conn->c_unacked_bytes < len) {
303                                 set_bit(RDSV3_MSG_ACK_REQUIRED, &rm->m_flags);

305                                 conn->c_unacked_packets =
306                                     rdsv3_sysctl_max_unacked_packets;
307                                 conn->c_unacked_bytes =
308                                     rdsv3_sysctl_max_unacked_bytes;
309                                 rdsv3_stats_inc(s_send_ack_required);
310                         } else {
311                                 conn->c_unacked_bytes -= len;
312                                 conn->c_unacked_packets--;
313                         }

315                         conn->c_xmit_rm = rm;
316                 }

318                 /*
319                  * Try and send an rdma message.  Let's see if we can
320                  * keep this simple and require that the transport either
```



```
321                  * send the whole rdma or none of it.
322                  */
323                 if (rm->m_rdma_op && !conn->c_xmit_rdma_sent) {
324                         ret = conn->c_trans->xmit_rdma(conn, rm->m_rdma_op);
325                         if (ret)
326                                 break;
327                         conn->c_xmit_rdma_sent = 1;
328                         /*
329                          * The transport owns the mapped memory for now.
330                          * You can't unmap it while it's on the send queue
331                          */
332                         set_bit(RDSV3_MSG_MAPPED, &rm->m_flags);
333                 }

335                 if (conn->c_xmit_hdr_off < sizeof (struct rdsv3_header) ||
336                     conn->c_xmit_sg < rm->m_nents) {
337                         ret = conn->c_trans->xmit(conn, rm,
338                             conn->c_xmit_hdr_off,
339                             conn->c_xmit_sg,
340                             conn->c_xmit_data_off);
341                         if (ret <= 0)
342                                 break;

344                         if (conn->c_xmit_hdr_off <
345                             sizeof (struct rdsv3_header)) {
346                                 tmp = min(ret,
347                                     sizeof (struct rdsv3_header) -
348                                     conn->c_xmit_hdr_off);
349                                 conn->c_xmit_hdr_off += tmp;
350                                 ret -= tmp;
351                         }

353                         sg = &rm->m_sg[conn->c_xmit_sg];
354                         while (ret) {
355                                 tmp = min(ret, rdsv3_sg_len(sg) -
356                                     conn->c_xmit_data_off);
357                                 conn->c_xmit_data_off += tmp;
358                                 ret -= tmp;
359                                 if (conn->c_xmit_data_off == rdsv3_sg_len(sg)) {
360                                         conn->c_xmit_data_off = 0;
361                                         sg++;
362                                         conn->c_xmit_sg++;
363                                         ASSERT(!(ret != 0 &&
364                                             conn->c_xmit_sg == rm->m_nents));
365                                 }
366                         }
367                 }
368         }

370         /* Nuke any messages we decided not to retransmit. */
371         if (!list_is_empty(&to_be_dropped))
372                 rdsv3_send_remove_from_sock(&to_be_dropped, RDS_RDMA_DROPPED);

374         if (conn->c_trans->xmit_complete)
375                 conn->c_trans->xmit_complete(conn);

377         /*
378          * We might be racing with another sender who queued a message but
379          * backed off on noticing that we held the c_send_lock.  If we check
380          * for queued messages after dropping the sem then either we'll
381          * see the queued message or the queuer will get the sem.  If we
382          * notice the queued message then we trigger an immediate retry.
383          *
384          * We need to be careful only to do this when we stopped processing
385          * the send queue because it was empty.  It's the only way we
386          * stop processing the loop when the transport hasn't taken
```

```
387          * responsibility for forward progress.
388          */
389         mutex_exit(&conn->c_send_lock);

391         if (conn->c_map_bytes || (send_quota == 0 && !was_empty)) {
392                 /*
393                  * We exhausted the send quota, but there's work left to
394                  * do. Return and (re-)schedule the send worker.
395                  */
396                 ret = -EAGAIN;
397         }

399         atomic_dec_32(&conn->c_senders);

401         if (ret == 0 && was_empty) {
402                 /*
403                  * A simple bit test would be way faster than taking the
404                  * spin lock
405                  */
406                 mutex_enter(&conn->c_lock);
407                 if (!list_is_empty(&conn->c_send_queue)) {
408                         rdsv3_stats_inc(s_send_sem_queue_raced);
409                         ret = -EAGAIN;
410                 }
411                 mutex_exit(&conn->c_lock);
412         }

414 out:
415         RDSV3_DPRINTF4("rdsv3_send_xmit", "Return(conn: %p, ret: %d)",
416             conn, ret);
417         return (ret);
418 }
```

_____***unchanged_portion_omitted_***

```
553 /*
554  * This is called from the IB send completion when we detect
555  * a RDMA operation that failed with remote access error.
556  * So speed is not an issue here.
557  */
558 struct rdsv3_message *
559 rdsv3_send_get_message(struct rdsv3_connection *conn,
560     struct rdsv3_rdma_op *op)
561 {
562         struct rdsv3_message *rm, *tmp, *found = NULL;

564         RDSV3_DPRINTF4("rdsv3_send_get_message", "Enter(conn: %p)", conn);

566         mutex_enter(&conn->c_lock);

568         RDSV3_FOR_EACH_LIST_NODE_SAFE(rm, tmp, &conn->c_retrans, m_conn_item) {
569                 if (rm->m_rdma_op == op) {
570                         atomic_inc_32(&rm->m_refcount);
570                         atomic_add_32(&rm->m_refcount, 1);
571                         found = rm;
572                         goto out;
573                 }
574         }

576         RDSV3_FOR_EACH_LIST_NODE_SAFE(rm, tmp, &conn->c_send_queue,
577             m_conn_item) {
578                 if (rm->m_rdma_op == op) {
579                         atomic_inc_32(&rm->m_refcount);
579                         atomic_add_32(&rm->m_refcount, 1);
580                         found = rm;
581                         break;
582                 }
```



```
583         }

585 out:
586         mutex_exit(&conn->c_lock);

588         return (found);
589 }
```

_____***unchanged_portion_omitted_***

****
212857 Mon Jul 28 07:44:39 2014
new/usr/src/uts/common/io/mac/mac.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
2247 /*
2248  * Allocate a minor number.
2249  */
2250 minor_t
2251 mac_minor_hold(boolean_t sleep)
2252 {
2253         minor_t minor;

2255         /*
2256          * Grab a value from the arena.
2257          */
2258         atomic_inc_32(&minor_count);
2258         atomic_add_32(&minor_count, 1);

2260         if (sleep)
2261                 minor = (uint_t)id_alloc(minor_ids);
2262         else
2263                 minor = (uint_t)id_alloc_nosleep(minor_ids);

2265         if (minor == 0) {
2266                 atomic_dec_32(&minor_count);
2266                 atomic_add_32(&minor_count, -1);
2267                 return (0);
2268         }

2270         return (minor);
2271 }

2273 /*
2274  * Release a previously allocated minor number.
2275  */
2276 void
2277 mac_minor_rele(minor_t minor)
2278 {
2279         /*
2280          * Return the value to the arena.
2281          */
2282         id_free(minor_ids, minor);
2283         atomic_dec_32(&minor_count);
2283         atomic_add_32(&minor_count, -1);
2284 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   19325 Mon Jul 28 07:44:40 2014
new/usr/src/uts/common/io/mac/mac_bcast.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_
```

```
 257 /*
 258  * Add the specified MAC client to the group corresponding to the specified
 259  * broadcast or multicast address.
 260  * Return 0 on success, or an errno value on failure.
 261  */
 262 int
 263 mac_bcast_add(mac_client_impl_t *mcip, const uint8_t *addr, uint16_t vid,
 264     mac_addrtype_t addrtype)
 265 {
 266         mac_impl_t              *mip = mcip->mci_mip;
 267         mac_bcast_grp_t         *grp = NULL, **last_grp;
 268         size_t                  addr_len = mip->mi_type->mt_addr_length;
 269         int                     rc = 0;
 270         int                     i, index = -1;
 271         mac_mcast_addrs_t       **prev_mi_addr = NULL;
 272         mac_mcast_addrs_t       **prev_mci_addr = NULL;

 274         ASSERT(MAC_PERIM_HELD((mac_handle_t)mip));

 276         ASSERT(addrtype == MAC_ADDRTYPE_MULTICAST ||
 277             addrtype == MAC_ADDRTYPE_BROADCAST);

 279         /*
 280          * Add the MAC client to the list of MAC clients associated
 281          * with the group.
 282          */
 283         if (addrtype == MAC_ADDRTYPE_MULTICAST) {
 284                 mac_mcast_addrs_t       *maddr;

 286                 /*
 287                  * In case of a driver (say aggr), we need this information
 288                  * on a per MAC instance basis.
 289                  */
 290                 prev_mi_addr = &mip->mi_mcast_addrs;
 291                 for (maddr = *prev_mi_addr; maddr != NULL;
 292                     prev_mi_addr = &maddr->mma_next, maddr = maddr->mma_next) {
 293                         if (bcmp(maddr->mma_addr, addr, addr_len) == 0)
 294                                 break;
 295                 }
 296                 if (maddr == NULL) {
 297                         /*
 298                          * For multicast addresses, have the underlying MAC
 299                          * join the corresponding multicast group.
 300                          */
 301                         rc = mip->mi_multicst(mip->mi_driver, B_TRUE, addr);
 302                         if (rc != 0)
 303                                 return (rc);
 304                         maddr = kmem_zalloc(sizeof (mac_mcast_addrs_t),
 305                             KM_SLEEP);
 306                         bcopy(addr, maddr->mma_addr, addr_len);
 307                         *prev_mi_addr = maddr;
 308                 } else {
 309                         prev_mi_addr = NULL;
 310                 }
 311                 maddr->mma_ref++;

 313                 /*
 314                  * We maintain a separate list for each MAC client. Get
 315                  * the entry or add, if it is not present.
```



```
 316                  */
 317                 prev_mci_addr = &mcip->mci_mcast_addrs;
 318                 for (maddr = *prev_mci_addr; maddr != NULL;
 319                     prev_mci_addr = &maddr->mma_next, maddr = maddr->mma_next) {
 320                         if (bcmp(maddr->mma_addr, addr, addr_len) == 0)
 321                                 break;
 322                 }
 323                 if (maddr == NULL) {
 324                         maddr = kmem_zalloc(sizeof (mac_mcast_addrs_t),
 325                             KM_SLEEP);
 326                         bcopy(addr, maddr->mma_addr, addr_len);
 327                         *prev_mci_addr = maddr;
 328                 } else {
 329                         prev_mci_addr = NULL;
 330                 }
 331                 maddr->mma_ref++;
 332         }

 334         /* The list is protected by the perimeter */
 335         last_grp = &mip->mi_bcast_grp;
 336         for (grp = *last_grp; grp != NULL;
 337             last_grp = &grp->mbg_next, grp = grp->mbg_next) {
 338                 if (bcmp(grp->mbg_addr, addr, addr_len) == 0 &&
 339                     grp->mbg_vid == vid)
 340                         break;
 341         }

 343         if (grp == NULL) {
 344                 /*
 345                  * The group does not yet exist, create it.
 346                  */
 347                 flow_desc_t flow_desc;
 348                 char flow_name[MAXFLOWNAMELEN];

 350                 grp = kmem_cache_alloc(mac_bcast_grp_cache, KM_SLEEP);
 351                 bzero(grp, sizeof (mac_bcast_grp_t));
 352                 grp->mbg_next = NULL;
 353                 grp->mbg_mac_impl = mip;

 355                 DTRACE_PROBE1(mac__bcast__add__new__group, mac_bcast_grp_t *,
 356                     grp);

 358                 grp->mbg_addr = kmem_zalloc(addr_len, KM_SLEEP);
 359                 bcopy(addr, grp->mbg_addr, addr_len);
 360                 grp->mbg_addrtype = addrtype;
 361                 grp->mbg_vid = vid;

 363                 /*
 364                  * Add a new flow to the underlying MAC.
 365                  */
 366                 bzero(&flow_desc, sizeof (flow_desc));
 367                 bcopy(addr, &flow_desc.fd_dst_mac, addr_len);
 368                 flow_desc.fd_mac_len = (uint32_t)addr_len;

 370                 flow_desc.fd_mask = FLOW_LINK_DST;
 371                 if (vid != 0) {
 372                         flow_desc.fd_vid = vid;
 373                         flow_desc.fd_mask |= FLOW_LINK_VID;
 374                 }

 376                 grp->mbg_id = atomic_inc_32_nv(&mac_bcast_id);
 376                 grp->mbg_id = atomic_add_32_nv(&mac_bcast_id, 1);
 377                 (void) sprintf(flow_name,
 378                     "mac/%s/mcast%d", mip->mi_name, grp->mbg_id);

 380                 rc = mac_flow_create(&flow_desc, NULL, flow_name,
```

```
381                     grp, FLOW_MCAST, &grp->mbg_flow_ent);
382                 if (rc != 0) {
383                         kmem_free(grp->mbg_addr, addr_len);
384                         kmem_cache_free(mac_bcast_grp_cache, grp);
385                         goto fail;
386                 }
387                 grp->mbg_flow_ent->fe_mbg = grp;
388                 mip->mi_bcast_ngrps++;

390                 /*
391                  * Initial creation reference on the flow. This is released
392                  * in the corresponding delete action i_mac_bcast_delete()
393                  */
394                 FLOW_REFHOLD(grp->mbg_flow_ent);

396                 /*
397                  * When the multicast and broadcast packet is received
398                  * by the underlying NIC, mac_rx_classify() will invoke
399                  * mac_bcast_send() with arg2=NULL, which will cause
400                  * mac_bcast_send() to send a copy of the packet(s)
401                  * to every MAC client opened on top of the underlying MAC.
402                  *
403                  * When the mac_bcast_send() function is invoked from
404                  * the transmit path of a MAC client, it will specify the
405                  * transmitting MAC client as the arg2 value, which will
406                  * allow mac_bcast_send() to skip that MAC client and not
407                  * send it a copy of the packet.
408                  *
409                  * We program the classifier to dispatch matching broadcast
410                  * packets to mac_bcast_send().
411                  */

413                 grp->mbg_flow_ent->fe_cb_fn = mac_bcast_send;
414                 grp->mbg_flow_ent->fe_cb_arg1 = grp;
415                 grp->mbg_flow_ent->fe_cb_arg2 = NULL;

417                 rc = mac_flow_add(mip->mi_flow_tab, grp->mbg_flow_ent);
418                 if (rc != 0) {
419                         FLOW_FINAL_REFRELE(grp->mbg_flow_ent);
420                         goto fail;
421                 }

423                 *last_grp = grp;
424         }

426         ASSERT(grp->mbg_addrtype == addrtype);

428         /*
429          * Add the MAC client to the list of MAC clients associated
430          * with the group.
431          */
432         rw_enter(&mip->mi_rw_lock, RW_WRITER);
433         for (i = 0; i < grp->mbg_nclients_alloc; i++) {
434                 /*
435                  * The MAC client was already added, say when we have
436                  * different unicast addresses with the same vid.
437                  * Just increment the ref and we are done.
438                  */
439                 if (grp->mbg_clients[i].mgb_client == mcip) {
440                         grp->mbg_clients[i].mgb_client_ref++;
441                         rw_exit(&mip->mi_rw_lock);
442                         return (0);
443                 } else if (grp->mbg_clients[i].mgb_client == NULL &&
444                     index == -1) {
445                         index = i;
446                 }
```



```
447         }
448         if (grp->mbg_nclients_alloc == grp->mbg_nclients) {
449                 mac_bcast_grp_mcip_t    *new_clients;
450                 uint_t                  new_size = grp->mbg_nclients+1;

452                 new_clients = kmem_zalloc(new_size *
453                     sizeof (mac_bcast_grp_mcip_t), KM_SLEEP);

455                 if (grp->mbg_nclients > 0) {
456                         ASSERT(grp->mbg_clients != NULL);
457                         bcopy(grp->mbg_clients, new_clients, grp->mbg_nclients *
458                             sizeof (mac_bcast_grp_mcip_t));
459                         kmem_free(grp->mbg_clients, grp->mbg_nclients *
460                             sizeof (mac_bcast_grp_mcip_t));
461                 }

463                 grp->mbg_clients = new_clients;
464                 grp->mbg_nclients_alloc = new_size;
465                 index = new_size - 1;
466         }

468         ASSERT(index != -1);
469         grp->mbg_clients[index].mgb_client = mcip;
470         grp->mbg_clients[index].mgb_client_ref = 1;
471         grp->mbg_nclients++;
472         /*
473          * Since we're adding to the list of MAC clients using that group,
474          * kick the generation count, which will allow mac_bcast_send()
475          * to detect that condition after re-acquiring the lock.
476          */
477         grp->mbg_clients_gen++;
478         rw_exit(&mip->mi_rw_lock);
479         return (0);

481 fail:
482         if (prev_mi_addr != NULL) {
483                 kmem_free(*prev_mi_addr, sizeof (mac_mcast_addrs_t));
484                 *prev_mi_addr = NULL;
485                 (void) mip->mi_multicst(mip->mi_driver, B_FALSE, addr);
486         }
487         if (prev_mci_addr != NULL) {
488                 kmem_free(*prev_mci_addr, sizeof (mac_mcast_addrs_t));
489                 *prev_mci_addr = NULL;
490         }
491         return (rc);
492 }
```

_____***unchanged_portion_omitted_***

****
133681 Mon Jul 28 07:44:40 2014
new/usr/src/uts/common/io/mega_sas/megaraid_sas.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
4565 static void
4566 issue_cmd_xscale(struct megasas_cmd *cmd, struct megasas_instance *instance)
4567 {
4568         atomic_inc_16(&instance->fw_outstanding);
4568         atomic_add_16(&instance->fw_outstanding, 1);

4570         /* Issue the command to the FW */
4571         WR_IB_QPORT((host_to_le32(cmd->frame_phys_addr) >> 3) |
4572             (cmd->frame_count - 1), instance);
4573 }

4575 static void
4576 issue_cmd_ppc(struct megasas_cmd *cmd, struct megasas_instance *instance)
4577 {
4578         atomic_inc_16(&instance->fw_outstanding);
4578         atomic_add_16(&instance->fw_outstanding, 1);

4580         /* Issue the command to the FW */
4581         WR_IB_QPORT((host_to_le32(cmd->frame_phys_addr)) |
4582             (((cmd->frame_count - 1) << 1) | 1), instance);
4583 }
```

_____***unchanged_portion_omitted_***

****
221539 Mon Jul 28 07:44:40 2014
new/usr/src/uts/common/io/mr_sas/mr_sas.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
6733 static void
6734 issue_cmd_ppc(struct mrsas_cmd *cmd, struct mrsas_instance *instance)
6735 {
6736         struct scsi_pkt *pkt;
6737         atomic_inc_16(&instance->fw_outstanding);
6737         atomic_add_16(&instance->fw_outstanding, 1);

6739         pkt = cmd->pkt;
6740         if (pkt) {
6741                 con_log(CL_DLEVEL1, (CE_NOTE, "%llx : issue_cmd_ppc:"
6742                     "ISSUED CMD TO FW : called : cmd:"
6743                     ": %p instance : %p pkt : %p pkt_time : %x\n",
6744                     gethrtime(), (void *)cmd, (void *)instance,
6745                     (void *)pkt, cmd->drv_pkt_time));
6746                 if (instance->adapterresetinprogress) {
6747                         cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
6748                         con_log(CL_ANN1, (CE_NOTE, "Reset the scsi_pkt timer"));
6749                 } else {
6750                         push_pending_mfi_pkt(instance, cmd);
6751                 }

6753         } else {
6754                 con_log(CL_DLEVEL1, (CE_NOTE, "%llx : issue_cmd_ppc:"
6755                     "ISSUED CMD TO FW : called : cmd : %p, instance: %p"
6756                     "(NO PKT)\n", gethrtime(), (void *)cmd, (void *)instance));
6757         }

6759         mutex_enter(&instance->reg_write_mtx);
6760         /* Issue the command to the FW */
6761         WR_IB_PICK_QPORT((cmd->frame_phys_addr) |
6762             (((cmd->frame_count - 1) << 1) | 1), instance);
6763         mutex_exit(&instance->reg_write_mtx);

6765 }
```

_____***unchanged_portion_omitted_***

****
105160 Mon Jul 28 07:44:41 2014
new/usr/src/uts/common/io/mr_sas/mr_sas_tbolt.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
1893 void
1894 tbolt_issue_cmd(struct mrsas_cmd *cmd, struct mrsas_instance *instance)
1895 {
1896         MRSAS_REQUEST_DESCRIPTOR_UNION *req_desc = cmd->request_desc;
1897         atomic_inc_16(&instance->fw_outstanding);
1897         atomic_add_16(&instance->fw_outstanding, 1);

1899         struct scsi_pkt *pkt;

1901         con_log(CL_ANN1,
1902             (CE_NOTE, "tbolt_issue_cmd: cmd->[SMID]=0x%X", cmd->SMID));

1904         con_log(CL_DLEVEL1, (CE_CONT,
1905             " [req desc Words] %" PRIx64 " \n", req_desc->Words));
1906         con_log(CL_DLEVEL1, (CE_CONT,
1907             " [req desc low part] %x \n",
1908             (uint_t)(req_desc->Words & 0xffffffffff)));
1909         con_log(CL_DLEVEL1, (CE_CONT,
1910             " [req desc high part] %x \n", (uint_t)(req_desc->Words >> 32)));
1911         pkt = cmd->pkt;

1913         if (pkt) {
1914                 con_log(CL_ANN1, (CE_CONT, "%llx :TBOLT issue_cmd_ppc:"
1915                     "ISSUED CMD TO FW : called : cmd:"
1916                     ": %p instance : %p pkt : %p pkt_time : %x\n",
1917                     gethrtime(), (void *)cmd, (void *)instance,
1918                     (void *)pkt, cmd->drv_pkt_time));
1919                 if (instance->adapterresetinprogress) {
1920                         cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
1921                         con_log(CL_ANN, (CE_NOTE,
1922                             "TBOLT Reset the scsi_pkt timer"));
1923                 } else {
1924                         push_pending_mfi_pkt(instance, cmd);
1925                 }

1927         } else {
1928                 con_log(CL_ANN1, (CE_CONT, "%llx :TBOLT issue_cmd_ppc:"
1929                     "ISSUED CMD TO FW : called : cmd : %p, instance: %p"
1930                     "(NO PKT)\n", gethrtime(), (void *)cmd, (void *)instance));
1931         }

1933         /* Issue the command to the FW */
1934         mutex_enter(&instance->reg_write_mtx);
1935         WR_IB_LOW_QPORT((uint32_t)(req_desc->Words), instance);
1936         WR_IB_HIGH_QPORT((uint32_t)(req_desc->Words >> 32), instance);
1937         mutex_exit(&instance->reg_write_mtx);
1938 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  159733 Mon Jul 28 07:44:41 2014
new/usr/src/uts/common/io/myri10ge/drv/myri10ge.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

3108 static int
3109 myri10ge_tx_tso_copy(struct myri10ge_slice_state *ss, mblk_t *mp,
3110     mcp_kreq_ether_send_t *req_list, int hdr_size, int pkt_size,
3111     uint16_t mss, uint8_t cksum_offset)
3112 {
3113         myri10ge_tx_ring_t *tx = &ss->tx;
3114         struct myri10ge_priv *mgp = ss->mgp;
3115         mblk_t *bp;
3116         mcp_kreq_ether_send_t *req;
3117         struct myri10ge_tx_copybuf *cp;
3118         caddr_t rptr, ptr;
3119         int mblen, count, cum_len, mss_resid, tx_req, pkt_size_tmp;
3120         int resid, avail, idx, hdr_size_tmp, tx_boundary;
3121         int rdma_count;
3122         uint32_t seglen, len, boundary, low, high_swapped;
3123         uint16_t pseudo_hdr_offset = htons(mss);
3124         uint8_t flags;

3126         tx_boundary = mgp->tx_boundary;
3127         hdr_size_tmp = hdr_size;
3128         resid = tx_boundary;
3129         count = 1;
3130         mutex_enter(&tx->lock);

3132         /* check to see if the slots are really there */
3133         avail = tx->mask - (tx->req - tx->done);
3134         if (unlikely(avail <=  MYRI10GE_MAX_SEND_DESC_TSO)) {
3135                 atomic_inc_32(&tx->stall);
3135                 atomic_add_32(&tx->stall, 1);
3136                 mutex_exit(&tx->lock);
3137                 return (EBUSY);
3138         }

3140         /* copy */
3141         cum_len = -hdr_size;
3142         count = 0;
3143         req = req_list;
3144         idx = tx->mask & tx->req;
3145         cp = &tx->cp[idx];
3146         low = ntohl(cp->dma.low);
3147         ptr = cp->va;
3148         cp->len = 0;
3149         if (mss) {
3150                 int payload = pkt_size - hdr_size;
3151                 uint16_t opackets = (payload / mss) + ((payload % mss) != 0);
3152                 tx->info[idx].ostat.opackets = opackets;
3153                 tx->info[idx].ostat.obytes = (opackets - 1) * hdr_size
3154                     + pkt_size;
3155         }
3156         hdr_size_tmp = hdr_size;
3157         mss_resid = mss;
3158         flags = (MXGEFW_FLAGS_TSO_HDR | MXGEFW_FLAGS_FIRST);
3159         tx_req = tx->req;
3160         for (bp = mp; bp != NULL; bp = bp->b_cont) {
3161                 mblen = MBLKL(bp);
3162                 rptr = (caddr_t)bp->b_rptr;
3163                 len = min(hdr_size_tmp, mblen);
3164                 if (len) {
3165                         bcopy(rptr, ptr, len);
```



```
3166                         rptr += len;
3167                         ptr += len;
3168                         resid -= len;
3169                         mblen -= len;
3170                         hdr_size_tmp -= len;
3171                         cp->len += len;
3172                         if (hdr_size_tmp)
3173                                 continue;
3174                         if (resid < mss) {
3175                                 tx_req++;
3176                                 idx = tx->mask & tx_req;
3177                                 cp = &tx->cp[idx];
3178                                 low = ntohl(cp->dma.low);
3179                                 ptr = cp->va;
3180                                 resid = tx_boundary;
3181                         }
3182                 }
3183                 while (mblen) {
3184                         len = min(mss_resid, mblen);
3185                         bcopy(rptr, ptr, len);
3186                         mss_resid -= len;
3187                         resid -= len;
3188                         mblen -= len;
3189                         rptr += len;
3190                         ptr += len;
3191                         cp->len += len;
3192                         if (mss_resid == 0) {
3193                                 mss_resid = mss;
3194                                 if (resid < mss) {
3195                                         tx_req++;
3196                                         idx = tx->mask & tx_req;
3197                                         cp = &tx->cp[idx];
3198                                         cp->len = 0;
3199                                         low = ntohl(cp->dma.low);
3200                                         ptr = cp->va;
3201                                         resid = tx_boundary;
3202                                 }
3203                         }
3204                 }
3205         }

3207         req = req_list;
3208         pkt_size_tmp = pkt_size;
3209         count = 0;
3210         rdma_count = 0;
3211         tx_req = tx->req;
3212         while (pkt_size_tmp) {
3213                 idx = tx->mask & tx_req;
3214                 cp = &tx->cp[idx];
3215                 high_swapped = cp->dma.high;
3216                 low = ntohl(cp->dma.low);
3217                 len = cp->len;
3218                 if (len == 0) {
3219                         printf("len=0! pkt_size_tmp=%d, pkt_size=%d\n",
3220                             pkt_size_tmp, pkt_size);
3221                         for (bp = mp; bp != NULL; bp = bp->b_cont) {
3222                                 mblen = MBLKL(bp);
3223                                 printf("mblen:%d\n", mblen);
3224                         }
3225                         pkt_size_tmp = pkt_size;
3226                         tx_req = tx->req;
3227                         while (pkt_size_tmp > 0) {
3228                                 idx = tx->mask & tx_req;
3229                                 cp = &tx->cp[idx];
3230                                 printf("cp->len = %d\n", cp->len);
3231                                 pkt_size_tmp -= cp->len;
```

```
3232                                    tx_req++;
3233                            }
3234                            printf("dropped\n");
3235                            MYRI10GE_ATOMIC_SLICE_STAT_INC(xmit_err);
3236                            goto done;
3237                    }
3238                    pkt_size_tmp -= len;
3239                    while (len) {
3240                            while (len) {
3241                                    uint8_t flags_next;
3242                                    int cum_len_next;

3244                                    boundary = (low + mgp->tx_boundary) &
3245                                        ~(mgp->tx_boundary - 1);
3246                                    seglen = boundary - low;
3247                                    if (seglen > len)
3248                                            seglen = len;

3250                                    flags_next = flags & ~MXGEFW_FLAGS_FIRST;
3251                                    cum_len_next = cum_len + seglen;
3252                                    (req-rdma_count)->rdma_count = rdma_count + 1;
3253                                    if (likely(cum_len >= 0)) {
3254                                            /* payload */
3255                                            int next_is_first, chop;

3257                                            chop = (cum_len_next > mss);
3258                                            cum_len_next = cum_len_next % mss;
3259                                            next_is_first = (cum_len_next == 0);
3260                                            flags |= chop *
3261                                                MXGEFW_FLAGS_TSO_CHOP;
3262                                            flags_next |= next_is_first *
3263                                                MXGEFW_FLAGS_FIRST;
3264                                            rdma_count |= -(chop | next_is_first);
3265                                            rdma_count += chop & !next_is_first;
3266                                    } else if (likely(cum_len_next >= 0)) {
3267                                            /* header ends */
3268                                            int small;

3270                                            rdma_count = -1;
3271                                            cum_len_next = 0;
3272                                            seglen = -cum_len;
3273                                            small = (mss <= MXGEFW_SEND_SMALL_SIZE);
3274                                            flags_next = MXGEFW_FLAGS_TSO_PLD |
3275                                                MXGEFW_FLAGS_FIRST |
3276                                                (small * MXGEFW_FLAGS_SMALL);
3277                                    }
3278                                    req->addr_high = high_swapped;
3279                                    req->addr_low = htonl(low);
3280                                    req->pseudo_hdr_offset = pseudo_hdr_offset;
3281                                    req->pad = 0; /* complete solid 16-byte block */
3282                                    req->rdma_count = 1;
3283                                    req->cksum_offset = cksum_offset;
3284                                    req->length = htons(seglen);
3285                                    req->flags = flags | ((cum_len & 1) *
3286                                        MXGEFW_FLAGS_ALIGN_ODD);
3287                                    if (cksum_offset > seglen)
3288                                            cksum_offset -= seglen;
3289                                    else
3290                                            cksum_offset = 0;
3291                                    low += seglen;
3292                                    len -= seglen;
3293                                    cum_len = cum_len_next;
3294                                    req++;
3295                                    req->flags = 0;
3296                                    flags = flags_next;
3297                                    count++;
```



```
3298                                            rdma_count++;
3299                                    }
3300                            }
3301                            tx_req++;
3302                    }
3303                    (req-rdma_count)->rdma_count = (uint8_t)rdma_count;
3304                    do {
3305                            req--;
3306                            req->flags |= MXGEFW_FLAGS_TSO_LAST;
3307                    } while (!(req->flags & (MXGEFW_FLAGS_TSO_CHOP |
3308                        MXGEFW_FLAGS_FIRST)));

3310                    myri10ge_submit_req(tx, req_list, count);
3311 done:
3312                    mutex_exit(&tx->lock);
3313                    freemsg(mp);
3314                    return (DDI_SUCCESS);
3315 }

3317 /*
3318  * Try to send the chain of buffers described by the mp.  We must not
3319  * encapsulate more than eth->tx.req - eth->tx.done, or
3320  * MXGEFW_MAX_SEND_DESC, whichever is more.
3321  */

3323 static int
3324 myri10ge_send(struct myri10ge_slice_state *ss, mblk_t *mp,
3325     mcp_kreq_ether_send_t *req_list, struct myri10ge_tx_buffer_state *tx_info)
3326 {
3327         struct myri10ge_priv *mgp = ss->mgp;
3328         myri10ge_tx_ring_t *tx = &ss->tx;
3329         mcp_kreq_ether_send_t *req;
3330         struct myri10ge_tx_dma_handle *handles, *dma_handle = NULL;
3331         mblk_t  *bp;
3332         ddi_dma_cookie_t cookie;
3333         int err, rv, count, avail, mblen, try_pullup, i, max_segs, maclen,
3334             rdma_count, cum_len, lso_hdr_size;
3335         uint32_t start, stuff, tx_offload_flags;
3336         uint32_t seglen, len, mss, boundary, low, high_swapped;
3337         uint_t ncookies;
3338         uint16_t pseudo_hdr_offset;
3339         uint8_t flags, cksum_offset, odd_flag;
3340         int pkt_size;
3341         int lso_copy = myri10ge_lso_copy;
3342         try_pullup = 1;

3344 again:
3345         /* Setup checksum offloading, if needed */
3346         mac_hcksum_get(mp, &start, &stuff, NULL, NULL, &tx_offload_flags);
3347         myri10ge_lso_info_get(mp, &mss, &tx_offload_flags);
3348         if (tx_offload_flags & HW_LSO) {
3349                 max_segs = MYRI10GE_MAX_SEND_DESC_TSO;
3350                 if ((tx_offload_flags & HCK_PARTIALCKSUM) == 0) {
3351                         MYRI10GE_ATOMIC_SLICE_STAT_INC(xmit_lsobadflags);
3352                         freemsg(mp);
3353                         return (DDI_SUCCESS);
3354                 }
3355         } else {
3356                 max_segs = MXGEFW_MAX_SEND_DESC;
3357                 mss = 0;
3358         }
3359         req = req_list;
3360         cksum_offset = 0;
3361         pseudo_hdr_offset = 0;

3363         /* leave an extra slot keep the ring from wrapping */
```

```
3364         avail = tx->mask - (tx->req - tx->done);

3366         /*
3367          * If we have > MXGEFW_MAX_SEND_DESC, then any over-length
3368          * message will need to be pulled up in order to fit.
3369          * Otherwise, we are low on transmit descriptors, it is
3370          * probably better to stall and try again rather than pullup a
3371          * message to fit.
3372          */

3374         if (avail < max_segs) {
3375                 err = EBUSY;
3376                 atomic_inc_32(&tx->stall_early);
3376                 atomic_add_32(&tx->stall_early, 1);
3377                 goto stall;
3378         }

3380         /* find out how long the frame is and how many segments it is */
3381         count = 0;
3382         odd_flag = 0;
3383         pkt_size = 0;
3384         flags = (MXGEFW_FLAGS_NO_TSO | MXGEFW_FLAGS_FIRST);
3385         for (bp = mp; bp != NULL; bp = bp->b_cont) {
3386                 dblk_t *dbp;
3387                 mblen = MBLKL(bp);
3388                 if (mblen == 0) {
3389                         /*
3390                          * we can't simply skip over 0-length mblks
3391                          * because the hardware can't deal with them,
3392                          * and we could leak them.
3393                          */
3394                         MYRI10GE_ATOMIC_SLICE_STAT_INC(xmit_zero_len);
3395                         err = EIO;
3396                         goto pullup;
3397                 }
3398                 /*
3399                  * There's no advantage to copying most gesballoc
3400                  * attached blocks, so disable lso copy in that case
3401                  */
3402                 if (mss && lso_copy == 1 && ((dbp = bp->b_datap) != NULL)) {
3403                         if ((void *)dbp->db_lastfree != myri10ge_db_lastfree) {
3404                                 lso_copy = 0;
3405                         }
3406                 }
3407                 pkt_size += mblen;
3408                 count++;
3409         }

3411         /* Try to pull up excessivly long chains */
3412         if (count >= max_segs) {
3413                 err = myri10ge_pullup(ss, mp);
3414                 if (likely(err == DDI_SUCCESS)) {
3415                         count = 1;
3416                 } else {
3417                         if (count <  MYRI10GE_MAX_SEND_DESC_TSO) {
3418                                 /*
3419                                  * just let the h/w send it, it will be
3420                                  * inefficient, but us better than dropping
3421                                  */
3422                                 max_segs = MYRI10GE_MAX_SEND_DESC_TSO;
3423                         } else {
3424                                 /* drop it */
3425                                 MYRI10GE_ATOMIC_SLICE_STAT_INC(xmit_err);
3426                                 freemsg(mp);
3427                                 return (0);
3428                         }
```



```
3429                 }
3430         }

3432         cum_len = 0;
3433         maclen = myri10ge_ether_parse_header(mp);

3435         if (tx_offload_flags & HCK_PARTIALCKSUM) {

3437                 cksum_offset = start + maclen;
3438                 pseudo_hdr_offset = htons(stuff + maclen);
3439                 odd_flag = MXGEFW_FLAGS_ALIGN_ODD;
3440                 flags |= MXGEFW_FLAGS_CKSUM;
3441         }

3443         lso_hdr_size = 0; /* -Wunitinialized */
3444         if (mss) { /* LSO */
3445                 /* this removes any CKSUM flag from before */
3446                 flags = (MXGEFW_FLAGS_TSO_HDR | MXGEFW_FLAGS_FIRST);
3447                 /*
3448                  * parse the headers and set cum_len to a negative
3449                  * value to reflect the offset of the TCP payload
3450                  */
3451                 lso_hdr_size =  myri10ge_lso_parse_header(mp, maclen);
3452                 cum_len = -lso_hdr_size;
3453                 if ((mss < mgp->tx_boundary) && lso_copy) {
3454                         err = myri10ge_tx_tso_copy(ss, mp, req_list,
3455                             lso_hdr_size, pkt_size, mss, cksum_offset);
3456                         return (err);
3457                 }

3459                 /*
3460                  * for TSO, pseudo_hdr_offset holds mss.  The firmware
3461                  * figures out where to put the checksum by parsing
3462                  * the header.
3463                  */

3465                 pseudo_hdr_offset = htons(mss);
3466         } else if (pkt_size <= MXGEFW_SEND_SMALL_SIZE) {
3467                 flags |= MXGEFW_FLAGS_SMALL;
3468                 if (pkt_size < myri10ge_tx_copylen) {
3469                         req->cksum_offset = cksum_offset;
3470                         req->pseudo_hdr_offset = pseudo_hdr_offset;
3471                         req->flags = flags;
3472                         err = myri10ge_tx_copy(ss, mp, req);
3473                         return (err);
3474                 }
3475                 cum_len = 0;
3476         }

3478         /* pull one DMA handle for each bp from our freelist */
3479         handles = NULL;
3480         err = myri10ge_alloc_tx_handles(ss, count, &handles);
3481         if (err != DDI_SUCCESS) {
3482                 err = DDI_FAILURE;
3483                 goto stall;
3484         }
3485         count = 0;
3486         rdma_count = 0;
3487         for (bp = mp; bp != NULL; bp = bp->b_cont) {
3488                 mblen = MBLKL(bp);
3489                 dma_handle = handles;
3490                 handles = handles->next;

3492                 rv = ddi_dma_addr_bind_handle(dma_handle->h, NULL,
3493                     (caddr_t)bp->b_rptr, mblen,
3494                     DDI_DMA_WRITE | DDI_DMA_STREAMING, DDI_DMA_SLEEP, NULL,
```

```
3495                    &cookie, &ncookies);
3496                if (unlikely(rv != DDI_DMA_MAPPED)) {
3497                        err = EIO;
3498                        try_pullup = 0;
3499                        dma_handle->next = handles;
3500                        handles = dma_handle;
3501                        goto abort_with_handles;
3502                }

3504                /* reserve the slot */
3505                tx_info[count].m = bp;
3506                tx_info[count].handle = dma_handle;

3508                for (; ; ) {
3509                        low = MYRI10GE_LOWPART_TO_U32(cookie.dmac_laddress);
3510                        high_swapped =
3511                            htonl(MYRI10GE_HIGHPART_TO_U32(
3512                            cookie.dmac_laddress));
3513                        len = (uint32_t)cookie.dmac_size;
3514                        while (len) {
3515                                uint8_t flags_next;
3516                                int cum_len_next;

3518                                boundary = (low + mgp->tx_boundary) &
3519                                    ~(mgp->tx_boundary - 1);
3520                                seglen = boundary - low;
3521                                if (seglen > len)
3522                                        seglen = len;

3524                                flags_next = flags & ~MXGEFW_FLAGS_FIRST;
3525                                cum_len_next = cum_len + seglen;
3526                                if (mss) {
3527                                        (req-rdma_count)->rdma_count =
3528                                            rdma_count + 1;
3529                                        if (likely(cum_len >= 0)) {
3530                                                /* payload */
3531                                                int next_is_first, chop;

3533                                                chop = (cum_len_next > mss);
3534                                                cum_len_next =
3535                                                    cum_len_next % mss;
3536                                                next_is_first =
3537                                                    (cum_len_next == 0);
3538                                                flags |= chop *
3539                                                    MXGEFW_FLAGS_TSO_CHOP;
3540                                                flags_next |= next_is_first *
3541                                                    MXGEFW_FLAGS_FIRST;
3542                                                rdma_count |=
3543                                                    -(chop | next_is_first);
3544                                                rdma_count +=
3545                                                    chop & !next_is_first;
3546                                        } else if (likely(cum_len_next >= 0)) {
3547                                                /* header ends */
3548                                                int small;

3550                                                rdma_count = -1;
3551                                                cum_len_next = 0;
3552                                                seglen = -cum_len;
3553                                                small = (mss <=
3554                                                    MXGEFW_SEND_SMALL_SIZE);
3555                                                flags_next =
3556                                                    MXGEFW_FLAGS_TSO_PLD
3557                                                    | MXGEFW_FLAGS_FIRST
3558                                                    | (small *
3559                                                    MXGEFW_FLAGS_SMALL);
3560                                        }
```



```
3561                                }
3562                                req->addr_high = high_swapped;
3563                                req->addr_low = htonl(low);
3564                                req->pseudo_hdr_offset = pseudo_hdr_offset;
3565                                req->pad = 0; /* complete solid 16-byte block */
3566                                req->rdma_count = 1;
3567                                req->cksum_offset = cksum_offset;
3568                                req->length = htons(seglen);
3569                                req->flags = flags | ((cum_len & 1) * odd_flag);
3570                                if (cksum_offset > seglen)
3571                                        cksum_offset -= seglen;
3572                                else
3573                                        cksum_offset = 0;
3574                                low += seglen;
3575                                len -= seglen;
3576                                cum_len = cum_len_next;
3577                                count++;
3578                                rdma_count++;
3579                                /*  make sure all the segments will fit */
3580                                if (unlikely(count >= max_segs)) {
3581                                        MYRI10GE_ATOMIC_SLICE_STAT_INC(
3582                                            xmit_lowbuf);
3583                                        /* may try a pullup */
3584                                        err = EBUSY;
3585                                        if (try_pullup)
3586                                                try_pullup = 2;
3587                                        goto abort_with_handles;
3588                                }
3589                                req++;
3590                                req->flags = 0;
3591                                flags = flags_next;
3592                                tx_info[count].m = 0;
3593                        }
3594                        ncookies--;
3595                        if (ncookies == 0)
3596                                break;
3597                        ddi_dma_nextcookie(dma_handle->h, &cookie);
3598                }
3599        }
3600        (req-rdma_count)->rdma_count = (uint8_t)rdma_count;

3602        if (mss) {
3603                do {
3604                        req--;
3605                        req->flags |= MXGEFW_FLAGS_TSO_LAST;
3606                } while (!(req->flags & (MXGEFW_FLAGS_TSO_CHOP |
3607                    MXGEFW_FLAGS_FIRST)));
3608        }

3610        /* calculate tx stats */
3611        if (mss) {
3612                uint16_t opackets;
3613                int payload;

3615                payload = pkt_size - lso_hdr_size;
3616                opackets = (payload / mss) + ((payload % mss) != 0);
3617                tx_info[0].stat.un.all = 0;
3618                tx_info[0].ostat.opackets = opackets;
3619                tx_info[0].ostat.obytes = (opackets - 1) * lso_hdr_size
3620                    + pkt_size;
3621        } else {
3622                myri10ge_tx_stat(&tx_info[0].stat,
3623                    (struct ether_header *)(void *)mp->b_rptr, 1, pkt_size);
3624        }
3625        mutex_enter(&tx->lock);
```

```
3627         /* check to see if the slots are really there */
3628         avail = tx->mask - (tx->req - tx->done);
3629         if (unlikely(avail <= count)) {
3630                 mutex_exit(&tx->lock);
3631                 err = 0;
3632                 goto late_stall;
3633         }

3635         myri10ge_send_locked(tx, req_list, tx_info, count);
3636         mutex_exit(&tx->lock);
3637         return (DDI_SUCCESS);

3639 late_stall:
3640         try_pullup = 0;
3641         atomic_inc_32(&tx->stall_late);
3641         atomic_add_32(&tx->stall_late, 1);

3643 abort_with_handles:
3644         /* unbind and free handles from previous mblks */
3645         for (i = 0; i < count; i++) {
3646                 bp = tx_info[i].m;
3647                 tx_info[i].m = 0;
3648                 if (bp) {
3649                         dma_handle = tx_info[i].handle;
3650                         (void) ddi_dma_unbind_handle(dma_handle->h);
3651                         dma_handle->next = handles;
3652                         handles = dma_handle;
3653                         tx_info[i].handle = NULL;
3654                         tx_info[i].m = NULL;
3655                 }
3656         }
3657         myri10ge_free_tx_handle_slist(tx, handles);
3658 pullup:
3659         if (try_pullup) {
3660                 err = myri10ge_pullup(ss, mp);
3661                 if (err != DDI_SUCCESS && try_pullup == 2) {
3662                         /* drop */
3663                         MYRI10GE_ATOMIC_SLICE_STAT_INC(xmit_err);
3664                         freemsg(mp);
3665                         return (0);
3666                 }
3667                 try_pullup = 0;
3668                 goto again;
3669         }

3671 stall:
3672         if (err != 0) {
3673                 if (err == EBUSY) {
3674                         atomic_inc_32(&tx->stall);
3674                         atomic_add_32(&tx->stall, 1);
3675                 } else {
3676                         MYRI10GE_ATOMIC_SLICE_STAT_INC(xmit_err);
3677                 }
3678         }
3679         return (err);
3680 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   13297 Mon Jul 28 07:44:41 2014
new/usr/src/uts/common/io/myri10ge/drv/myri10ge_var.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_


 285 #define MYRI10GE_NIC_STAT_INC(field)                                    \
 286 (((struct myri10ge_nic_stat *)mgp->ksp_stat->ks_data)->field.value.ul)++
 287 #define MYRI10GE_SLICE_STAT_INC(field)                                  \
 288 (((struct myri10ge_slice_stat *)ss->ksp_stat->ks_data)->field.value.ul)++
 289 #define MYRI10GE_SLICE_STAT_ADD(field, val)                             \
 290 (((struct myri10ge_slice_stat *)ss->ksp_stat->ks_data)->field.value.ul) += val
 291 #define MYRI10GE_SLICE_STAT_DEC(field)                                  \
 292 (((struct myri10ge_slice_stat *)ss->ksp_stat->ks_data)->field.value.ul)--
 293 #define MYRI10GE_ATOMIC_SLICE_STAT_INC(field)                           \
 294 atomic_inc_ulong(&(((struct myri10ge_slice_stat *)                      \
 295         ss->ksp_stat->ks_data)->field.value.ul))
 294 atomic_add_long(&(((struct myri10ge_slice_stat *)                       \
 295             ss->ksp_stat->ks_data)->field.value.ul), 1)
 296 #define MYRI10GE_ATOMIC_SLICE_STAT_DEC(field)                           \
 297 atomic_dec_ulong(&(((struct myri10ge_slice_stat *)                      \
 298         ss->ksp_stat->ks_data)->field.value.ul))
 297 atomic_add_long(&(((struct myri10ge_slice_stat *)                       \
 298             ss->ksp_stat->ks_data)->field.value.ul), -1)
 299 #define MYRI10GE_SLICE_STAT(field)                                      \
 300 (((struct myri10ge_slice_stat *)ss->ksp_stat->ks_data)->field.value.ul)


 303 struct myri10ge_tx_copybuf
 304 {
 305         caddr_t va;
 306         int len;
 307         struct myri10ge_dma_stuff dma;
 308 };
______unchanged_portion_omitted_
```

```
**********************************************************
   14823 Mon Jul 28 07:44:41 2014
new/usr/src/uts/common/io/neti_impl.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 122 net_handle_t
 123 net_protocol_lookup(netid_t netid, const char *protocol)
 124 {
 125         neti_stack_t *nts;
 126         net_handle_t nd;

 128         ASSERT(protocol != NULL);

 130         nts = net_getnetistackbyid(netid);
 131         if (nts == NULL)
 132                 return (NULL);

 134         mutex_enter(&nts->nts_lock);
 135         nd = net_find(protocol, nts);
 136         if (nd != NULL)
 137                 atomic_inc_32((uint_t *)&nd->netd_refcnt);
 137                 atomic_add_32((uint_t *)&nd->netd_refcnt, 1);
 138         mutex_exit(&nts->nts_lock);
 139         return (nd);
 140 }

 142 /*
 143  * Note: the man page specifies "returns -1 if the value passed in is unknown
 144  * to this framework".  We are not doing a lookup in this function, just a
 145  * simply add to the netd_refcnt of the net_handle_t passed in, so -1 is never a
 146  * return value.
 147  */
 148 int
 149 net_protocol_release(net_handle_t info)
 150 {

 152         ASSERT(info->netd_refcnt > 0);
 153         /*
 154          * Is this safe? No hold on nts_lock? Consider that if the caller
 155          * of net_protocol_release() is going to free this structure then
 156          * it is now the only owner (refcnt==1) and it will have been
 157          * removed from the nts_netd_head list on the neti_stack_t from a
 158          * call to net_protocol_unregister already, so it is thus an orphan.
 159          */
 160         if (atomic_dec_32_nv((uint_t *)&info->netd_refcnt) == 0) {
 160         if (atomic_add_32_nv((uint_t *)&info->netd_refcnt, -1) == 0) {
 161                 ASSERT(info->netd_hooks == NULL);
 162                 ASSERT(info->netd_stack == NULL);
 163                 kmem_free(info, sizeof (struct net_data));
 164         }

 166         return (0);
 167 }

 169 net_handle_t
 170 net_protocol_walk(netid_t netid, net_handle_t info)
 171 {
 172         struct net_data *n = NULL;
 173         boolean_t found = B_FALSE;
 174         neti_stack_t *nts;

 176         nts = net_getnetistackbyid(netid);
 177         ASSERT(nts != NULL);
```



```
 179         if (info == NULL)
 180                 found = B_TRUE;

 182         mutex_enter(&nts->nts_lock);
 183         LIST_FOREACH(n, &nts->nts_netd_head, netd_list) {
 184                 if (found) {
 185                         /*
 186                          * We are only interested in finding protocols that
 187                          * are not in some sort of shutdown state.  There is
 188                          * no need to check for netd_stack==NULL because
 189                          * that implies it is no longer on this list.
 190                          */
 191                         if (n->netd_condemned == 0)
 192                                 continue;
 193                         break;
 194                 }

 196                 if (n == info)
 197                         found = B_TRUE;
 198         }

 200         if (info != NULL)
 201                 (void) net_protocol_release(info);

 203         if (n != NULL)
 204                 atomic_inc_32((uint_t *)&n->netd_refcnt);
 204                 atomic_add_32((uint_t *)&n->netd_refcnt, 1);

 206         mutex_exit(&nts->nts_lock);

 208         return (n);
 209 }
______unchanged_portion_omitted_
```

```
**********************************************************
  133972 Mon Jul 28 07:44:42 2014
new/usr/src/uts/common/io/nxge/nxge_rxdma.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1665 void
1666 nxge_freeb(p_rx_msg_t rx_msg_p)
1667 {
1668         size_t size;
1669         uchar_t *buffer = NULL;
1670         int ref_cnt;
1671         boolean_t free_state = B_FALSE;

1673         rx_rbr_ring_t *ring = rx_msg_p->rx_rbr_p;

1675         NXGE_DEBUG_MSG((NULL, MEM2_CTL, "==> nxge_freeb"));
1676         NXGE_DEBUG_MSG((NULL, MEM2_CTL,
1677             "nxge_freeb:rx_msg_p = $%p (block pending %d)",
1678             rx_msg_p, nxge_mblks_pending));

1680         /*
1681          * First we need to get the free state, then
1682          * atomic decrement the reference count to prevent
1683          * the race condition with the interrupt thread that
1684          * is processing a loaned up buffer block.
1685          */
1686         free_state = rx_msg_p->free;
1687         ref_cnt = atomic_dec_32_nv(&rx_msg_p->ref_cnt);
1687         ref_cnt = atomic_add_32_nv(&rx_msg_p->ref_cnt, -1);
1688         if (!ref_cnt) {
1689                 atomic_dec_32(&nxge_mblks_pending);
1690                 buffer = rx_msg_p->buffer;
1691                 size = rx_msg_p->block_size;
1692                 NXGE_DEBUG_MSG((NULL, MEM2_CTL, "nxge_freeb: "
1693                     "will free: rx_msg_p = $%p (block pending %d)",
1694                     rx_msg_p, nxge_mblks_pending));

1696                 if (!rx_msg_p->use_buf_pool) {
1697                         KMEM_FREE(buffer, size);
1698                 }

1700                 KMEM_FREE(rx_msg_p, sizeof (rx_msg_t));

1702                 if (ring) {
1703                         /*
1704                          * Decrement the receive buffer ring's reference
1705                          * count, too.
1706                          */
1707                         atomic_dec_32(&ring->rbr_ref_cnt);

1709                         /*
1710                          * Free the receive buffer ring, if
1711                          * 1. all the receive buffers have been freed
1712                          * 2. and we are in the proper state (that is,
1713                          *    we are not UNMAPPING).
1714                          */
1715                         if (ring->rbr_ref_cnt == 0 &&
1716                             ring->rbr_state == RBR_UNMAPPED) {
1717                                 /*
1718                                  * Free receive data buffers,
1719                                  * buffer index information
1720                                  * (rxring_info) and
1721                                  * the message block ring.
1722                                  */
```



```
1723                                 NXGE_DEBUG_MSG((NULL, RX_CTL,
1724                                     "nxge_freeb:rx_msg_p = $%p "
1725                                     "(block pending %d) free buffers",
1726                                     rx_msg_p, nxge_mblks_pending));
1727                                 nxge_rxdma_databuf_free(ring);
1728                                 if (ring->ring_info) {
1729                                         KMEM_FREE(ring->ring_info,
1730                                             sizeof (rxring_info_t));
1731                                 }

1733                                 if (ring->rx_msg_ring) {
1734                                         KMEM_FREE(ring->rx_msg_ring,
1735                                             ring->tnblocks *
1736                                             sizeof (p_rx_msg_t));
1737                                 }
1738                                 KMEM_FREE(ring, sizeof (*ring));
1739                         }
1740                 }
1741                 return;
1742         }

1744         /*
1745          * Repost buffer.
1746          */
1747         if (free_state && (ref_cnt == 1) && ring) {
1748                 NXGE_DEBUG_MSG((NULL, RX_CTL,
1749                     "nxge_freeb: post page $%p:", rx_msg_p));
1750                 if (ring->rbr_state == RBR_POSTING)
1751                         nxge_post_page(rx_msg_p->nxgep, ring, rx_msg_p);
1752         }

1754         NXGE_DEBUG_MSG((NULL, MEM2_CTL, "<== nxge_freeb"));
1755 }
______unchanged_portion_omitted_
```

****
93051 Mon Jul 28 07:44:42 2014
new/usr/src/uts/common/io/pciex/pcie_fault.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
2621 #define PCIE_EREPORT    DDI_IO_CLASS "." PCI_ERROR_SUBCLASS "." PCIEX_FABRIC
2622 static int
2623 pf_ereport_setup(dev_info_t *dip, uint64_t ena, nvlist_t **ereport,
2624     nvlist_t **detector, errorq_elem_t **eqep)
2625 {
2626         struct i_ddi_fmhdl *fmhdl = DEVI(dip)->devi_fmhdl;
2627         char device_path[MAXPATHLEN];
2628         nv_alloc_t *nva;

2630         *eqep = errorq_reserve(fmhdl->fh_errorq);
2631         if (*eqep == NULL) {
2632                 atomic_inc_64(&fmhdl->fh_kstat.fek_erpt_dropped.value.ui64);
2632                 atomic_add_64(&fmhdl->fh_kstat.fek_erpt_dropped.value.ui64, 1);
2633                 return (DDI_FAILURE);
2634         }

2636         *ereport = errorq_elem_nvl(fmhdl->fh_errorq, *eqep);
2637         nva = errorq_elem_nva(fmhdl->fh_errorq, *eqep);

2639         ASSERT(*ereport);
2640         ASSERT(nva);

2642         /*
2643          * Use the dev_path/devid for this device instance.
2644          */
2645         *detector = fm_nvlist_create(nva);
2646         if (dip == ddi_root_node()) {
2647                 device_path[0] = '/';
2648                 device_path[1] = '\0';
2649         } else {
2650                 (void) ddi_pathname(dip, device_path);
2651         }

2653         fm_fmri_dev_set(*detector, FM_DEV_SCHEME_VERSION, NULL,
2654             device_path, NULL, NULL);

2656         if (ena == 0)
2657                 ena = fm_ena_generate(0, FM_ENA_FMT1);

2659         fm_ereport_set(*ereport, 0, PCIE_EREPORT, ena, *detector, NULL);

2661         return (DDI_SUCCESS);
2662 }
```

_____***unchanged_portion_omitted_***

****
260594 Mon Jul 28 07:44:42 2014
new/usr/src/uts/common/io/rsm/rsm.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****

_____unchanged_portion_omitted_

```
4019 static void
4020 rsm_intr_event(rsmipc_request_t *msg)
4021 {
4022         rsmseg_t        *seg;
4023         rsmresource_t   *p;
4024         rsm_node_id_t   src_node;
4025         DBG_DEFINE(category,
4026             RSM_KERNEL_AGENT | RSM_FUNC_ALL | RSM_INTR_CALLBACK);

4028         DBG_PRINTF((category, RSM_DEBUG_VERBOSE, "rsm_intr_event enter\n"));

4030         src_node = msg->rsmipc_hdr.rsmipc_src;

4032         if ((seg = msg->rsmipc_segment_cookie) != NULL) {
4033                 /* This is for an import segment */
4034                 uint_t hashval = rsmhash(msg->rsmipc_key);

4036                 rw_enter(&rsm_import_segs.rsmhash_rw, RW_READER);

4038                 p = (rsmresource_t *)rsmhash_getbkt(&rsm_import_segs, hashval);

4040                 for (; p; p = p->rsmrc_next) {
4041                         if ((p->rsmrc_key == msg->rsmipc_key) &&
4042                             (p->rsmrc_node == src_node)) {
4043                                 seg = (rsmseg_t *)p;
4044                                 rsmseglock_acquire(seg);

4046                                 atomic_inc_32(&seg->s_pollevent);
4046                                 atomic_add_32(&seg->s_pollevent, 1);

4048                                 if (seg->s_pollflag & RSM_SEGMENT_POLL)
4049                                         pollwakeup(&seg->s_poll, POLLRDNORM);

4051                                 rsmseglock_release(seg);
4052                         }
4053                 }

4055                 rw_exit(&rsm_import_segs.rsmhash_rw);
4056         } else {
4057                 /* This is for an export segment */
4058                 seg = rsmexport_lookup(msg->rsmipc_key);
4059                 if (!seg) {
4060                         DBG_PRINTF((category, RSM_DEBUG_VERBOSE,
4061                             "rsm_intr_event done: exp seg not found\n"));
4062                         return;
4063                 }

4065                 ASSERT(rsmseglock_held(seg));

4067                 atomic_inc_32(&seg->s_pollevent);
4067                 atomic_add_32(&seg->s_pollevent, 1);

4069                 /*
4070                  * We must hold the segment lock here, or else the segment
4071                  * can be freed while pollwakeup is using it. This implies
4072                  * that we MUST NOT grab the segment lock during rsm_chpoll,
4073                  * as outlined in the chpoll(2) man page.
4074                  */
4075                 if (seg->s_pollflag & RSM_SEGMENT_POLL)
```



```
4076                         pollwakeup(&seg->s_poll, POLLRDNORM);

4078                 rsmseglock_release(seg);
4079         }

4081         DBG_PRINTF((category, RSM_DEBUG_VERBOSE, "rsm_intr_event done\n"));
4082 }
```

_____unchanged_portion_omitted_

```
5289 static int
5290 rsmipc_send(rsm_node_id_t dest, rsmipc_request_t *req, rsmipc_reply_t *reply)
5291 {
5292         int             e = 0;
5293         int             credit_check = 0;
5294         int             retry_cnt = 0;
5295         int             min_retry_cnt = 10;
5296         rsm_send_t      is;
5297         rsmipc_slot_t   *rslot;
5298         adapter_t       *adapter;
5299         path_t          *path;
5300         sendq_token_t   *sendq_token;
5301         sendq_token_t   *used_sendq_token = NULL;
5302         rsm_send_q_handle_t     ipc_handle;
5303         DBG_DEFINE(category,
5304             RSM_KERNEL_AGENT | RSM_FUNC_ALL | RSM_INTR_CALLBACK);

5306         DBG_PRINTF((category, RSM_DEBUG_VERBOSE, "rsmipc_send enter:dest=%d",
5307             dest));

5309         /*
5310          * Check if this is a local case
5311          */
5312         if (dest == my_nodeid) {
5313                 switch (req->rsmipc_hdr.rsmipc_type) {
5314                 case RSMIPC_MSG_SEGCONNECT:
5315                         reply->rsmipc_status = (short)rsmsegacl_validate(
5316                             req, dest, reply);
5317                         break;
5318                 case RSMIPC_MSG_BELL:
5319                         req->rsmipc_hdr.rsmipc_src = dest;
5320                         rsm_intr_event(req);
5321                         break;
5322                 case RSMIPC_MSG_IMPORTING:
5323                         importer_list_add(dest, req->rsmipc_key,
5324                             req->rsmipc_adapter_hwaddr,
5325                             req->rsmipc_segment_cookie);
5326                         break;
5327                 case RSMIPC_MSG_NOTIMPORTING:
5328                         importer_list_rm(dest, req->rsmipc_key,
5329                             req->rsmipc_segment_cookie);
5330                         break;
5331                 case RSMIPC_MSG_REPUBLISH:
5332                         importer_update(dest, req->rsmipc_key,
5333                             req->rsmipc_perm);
5334                         break;
5335                 case RSMIPC_MSG_SUSPEND:
5336                         importer_suspend(dest);
5337                         break;
5338                 case RSMIPC_MSG_SUSPEND_DONE:
5339                         rsm_suspend_complete(dest, 0);
5340                         break;
5341                 case RSMIPC_MSG_RESUME:
5342                         importer_resume(dest);
5343                         break;
5344                 default:
5345                         ASSERT(0);
```

```
5346                 }
5347                 DBG_PRINTF((category, RSM_DEBUG_VERBOSE,
5348                     "rsmipc_send done\n"));
5349                 return (0);
5350         }

5352         if (dest >= MAX_NODES) {
5353                 DBG_PRINTF((category, RSM_ERR,
5354                     "rsm: rsmipc_send bad node number %x\n", dest));
5355                 return (RSMERR_REMOTE_NODE_UNREACHABLE);
5356         }

5358         /*
5359          * Oh boy! we are going remote.
5360          */

5362         /*
5363          * identify if we need to have credits to send this message
5364          * - only selected requests are flow controlled
5365          */
5366         if (req != NULL) {
5367                 DBG_PRINTF((category, RSM_DEBUG_VERBOSE,
5368                     "rsmipc_send:request type=%d\n",
5369                     req->rsmipc_hdr.rsmipc_type));

5371                 switch (req->rsmipc_hdr.rsmipc_type) {
5372                 case RSMIPC_MSG_SEGCONNECT:
5373                 case RSMIPC_MSG_DISCONNECT:
5374                 case RSMIPC_MSG_IMPORTING:
5375                 case RSMIPC_MSG_SUSPEND:
5376                 case RSMIPC_MSG_SUSPEND_DONE:
5377                 case RSMIPC_MSG_RESUME:
5378                         credit_check = 1;
5379                         break;
5380                 default:
5381                         credit_check = 0;
5382                 }
5383         }

5385 again:
5386         if (retry_cnt++ == min_retry_cnt) {
5387                 /* backoff before further retries for 10ms */
5388                 delay(drv_usectohz(10000));
5389                 retry_cnt = 0; /* reset retry_cnt */
5390         }
5391         sendq_token = rsmka_get_sendq_token(dest, used_sendq_token);
5392         if (sendq_token == NULL) {
5393                 DBG_PRINTF((category, RSM_ERR,
5394                     "rsm: rsmipc_send no device to reach node %d\n", dest));
5395                 return (RSMERR_REMOTE_NODE_UNREACHABLE);
5396         }

5398         if ((sendq_token == used_sendq_token) &&
5399             ((e == RSMERR_CONN_ABORTED) || (e == RSMERR_TIMEOUT) ||
5400             (e == RSMERR_COMM_ERR_MAYBE_DELIVERED))) {
5401                 rele_sendq_token(sendq_token);
5402                 DBG_PRINTF((category, RSM_DEBUG, "rsmipc_send done=%d\n", e));
5403                 return (RSMERR_CONN_ABORTED);
5404         } else
5405                 used_sendq_token = sendq_token;

5407 /* lint -save -e413 */
5408         path = SQ_TOKEN_TO_PATH(sendq_token);
5409         adapter = path->local_adapter;
5410 /* lint -restore */
5411         ipc_handle = sendq_token->rsmpi_sendq_handle;
```



```
5413         DBG_PRINTF((category, RSM_DEBUG_VERBOSE,
5414             "rsmipc_send: path=%lx sendq_hdl=%lx\n", path, ipc_handle));

5416         if (reply == NULL) {
5417                 /* Send request without ack */
5418                 /*
5419                  * Set the rsmipc_version number in the msghdr for KA
5420                  * communication versioning
5421                  */
5422                 req->rsmipc_hdr.rsmipc_version = RSM_VERSION;
5423                 req->rsmipc_hdr.rsmipc_src = my_nodeid;
5424                 /*
5425                  * remote endpoints incn should match the value in our
5426                  * path's remote_incn field. No need to grab any lock
5427                  * since we have refcnted the path in rsmka_get_sendq_token
5428                  */
5429                 req->rsmipc_hdr.rsmipc_incn = path->remote_incn;

5431                 is.is_data = (void *)req;
5432                 is.is_size = sizeof (*req);
5433                 is.is_flags = RSM_INTR_SEND_DELIVER | RSM_INTR_SEND_SLEEP;
5434                 is.is_wait = 0;

5436                 if (credit_check) {
5437                         mutex_enter(&path->mutex);
5438                         /*
5439                          * wait till we recv credits or path goes down. If path
5440                          * goes down rsm_send will fail and we handle the error
5441                          * then
5442                          */
5443                         while ((sendq_token->msgbuf_avail == 0) &&
5444                             (path->state == RSMKA_PATH_ACTIVE)) {
5445                                 e = cv_wait_sig(&sendq_token->sendq_cv,
5446                                     &path->mutex);
5447                                 if (e == 0) {
5448                                         mutex_exit(&path->mutex);
5449                                         no_reply_cnt++;
5450                                         rele_sendq_token(sendq_token);
5451                                         DBG_PRINTF((category, RSM_DEBUG,
5452                                             "rsmipc_send done: "
5453                                             "cv_wait INTERRUPTED"));
5454                                         return (RSMERR_INTERRUPTED);
5455                                 }
5456                         }

5458                         /*
5459                          * path is not active retry on another path.
5460                          */
5461                         if (path->state != RSMKA_PATH_ACTIVE) {
5462                                 mutex_exit(&path->mutex);
5463                                 rele_sendq_token(sendq_token);
5464                                 e = RSMERR_CONN_ABORTED;
5465                                 DBG_PRINTF((category, RSM_ERR,
5466                                     "rsm: rsmipc_send: path !ACTIVE"));
5467                                 goto again;
5468                         }

5470                         ASSERT(sendq_token->msgbuf_avail > 0);

5472                         /*
5473                          * reserve a msgbuf
5474                          */
5475                         sendq_token->msgbuf_avail--;

5477                         mutex_exit(&path->mutex);
```

```
5479                        e = adapter->rsmpi_ops->rsm_send(ipc_handle, &is,
5480                            NULL);

5482                        if (e != RSM_SUCCESS) {
5483                                mutex_enter(&path->mutex);
5484                                /*
5485                                 * release the reserved msgbuf since
5486                                 * the send failed
5487                                 */
5488                                sendq_token->msgbuf_avail++;
5489                                cv_broadcast(&sendq_token->sendq_cv);
5490                                mutex_exit(&path->mutex);
5491                        }
5492                } else
5493                        e = adapter->rsmpi_ops->rsm_send(ipc_handle, &is,
5494                            NULL);

5496                no_reply_cnt++;
5497                rele_sendq_token(sendq_token);
5498                if (e != RSM_SUCCESS) {
5499                        DBG_PRINTF((category, RSM_ERR,
5500                            "rsm: rsmipc_send no reply send"
5501                            " err = %d no reply count = %d\n",
5502                            e, no_reply_cnt));
5503                        ASSERT(e != RSMERR_QUEUE_FENCE_UP &&
5504                            e != RSMERR_BAD_BARRIER_HNDL);
5505                        atomic_inc_64(&rsm_ipcsend_errcnt);
5505                        atomic_add_64(&rsm_ipcsend_errcnt, 1);
5506                        goto again;
5507                } else {
5508                        DBG_PRINTF((category, RSM_DEBUG_VERBOSE,
5509                            "rsmipc_send done\n"));
5510                        return (e);
5511                }

5513        }

5515        if (req == NULL) {
5516                /* Send reply - No flow control is done for reply */
5517                /*
5518                 * Set the version in the msg header for KA communication
5519                 * versioning
5520                 */
5521                reply->rsmipc_hdr.rsmipc_version = RSM_VERSION;
5522                reply->rsmipc_hdr.rsmipc_src = my_nodeid;
5523                /* incn number is not used for reply msgs currently */
5524                reply->rsmipc_hdr.rsmipc_incn = path->remote_incn;

5526                is.is_data = (void *)reply;
5527                is.is_size = sizeof (*reply);
5528                is.is_flags = RSM_INTR_SEND_DELIVER | RSM_INTR_SEND_SLEEP;
5529                is.is_wait = 0;
5530                e = adapter->rsmpi_ops->rsm_send(ipc_handle, &is, NULL);
5531                rele_sendq_token(sendq_token);
5532                if (e != RSM_SUCCESS) {
5533                        DBG_PRINTF((category, RSM_ERR,
5534                            "rsm: rsmipc_send reply send"
5535                            " err = %d\n", e));
5536                        atomic_inc_64(&rsm_ipcsend_errcnt);
5536                        atomic_add_64(&rsm_ipcsend_errcnt, 1);
5537                        goto again;
5538                } else {
5539                        DBG_PRINTF((category, RSM_DEBUG_VERBOSE,
5540                            "rsmipc_send done\n"));
5541                        return (e);
```



```
5542                }
5543        }

5545        /* Reply needed */
5546        rslot = rsmipc_alloc(); /* allocate a new ipc slot */

5548        mutex_enter(&rslot->rsmipc_lock);

5550        rslot->rsmipc_data = (void *)reply;
5551        RSMIPC_SET(rslot, RSMIPC_PENDING);

5553        while (RSMIPC_GET(rslot, RSMIPC_PENDING)) {
5554                /*
5555                 * Set the rsmipc_version number in the msghdr for KA
5556                 * communication versioning
5557                 */
5558                req->rsmipc_hdr.rsmipc_version = RSM_VERSION;
5559                req->rsmipc_hdr.rsmipc_src = my_nodeid;
5560                req->rsmipc_hdr.rsmipc_cookie = rslot->rsmipc_cookie;
5561                /*
5562                 * remote endpoints incn should match the value in our
5563                 * path's remote_incn field. No need to grab any lock
5564                 * since we have refcnted the path in rsmka_get_sendq_token
5565                 */
5566                req->rsmipc_hdr.rsmipc_incn = path->remote_incn;

5568                is.is_data = (void *)req;
5569                is.is_size = sizeof (*req);
5570                is.is_flags = RSM_INTR_SEND_DELIVER | RSM_INTR_SEND_SLEEP;
5571                is.is_wait = 0;
5572                if (credit_check) {

5574                        mutex_enter(&path->mutex);
5575                        /*
5576                         * wait till we recv credits or path goes down. If path
5577                         * goes down rsm_send will fail and we handle the error
5578                         * then.
5579                         */
5580                        while ((sendq_token->msgbuf_avail == 0) &&
5581                            (path->state == RSMKA_PATH_ACTIVE)) {
5582                                e = cv_wait_sig(&sendq_token->sendq_cv,
5583                                    &path->mutex);
5584                                if (e == 0) {
5585                                        mutex_exit(&path->mutex);
5586                                        RSMIPC_CLEAR(rslot, RSMIPC_PENDING);
5587                                        rsmipc_free(rslot);
5588                                        rele_sendq_token(sendq_token);
5589                                        DBG_PRINTF((category, RSM_DEBUG,
5590                                            "rsmipc_send done: "
5591                                            "cv_wait INTERRUPTED"));
5592                                        return (RSMERR_INTERRUPTED);
5593                                }
5594                        }

5596                        /*
5597                         * path is not active retry on another path.
5598                         */
5599                        if (path->state != RSMKA_PATH_ACTIVE) {
5600                                mutex_exit(&path->mutex);
5601                                RSMIPC_CLEAR(rslot, RSMIPC_PENDING);
5602                                rsmipc_free(rslot);
5603                                rele_sendq_token(sendq_token);
5604                                e = RSMERR_CONN_ABORTED;
5605                                DBG_PRINTF((category, RSM_ERR,
5606                                    "rsm: rsmipc_send: path !ACTIVE"));
5607                                goto again;
```

```
5608                        }

5610                        ASSERT(sendq_token->msgbuf_avail > 0);

5612                        /*
5613                         * reserve a msgbuf
5614                         */
5615                        sendq_token->msgbuf_avail--;

5617                        mutex_exit(&path->mutex);

5619                        e = adapter->rsmpi_ops->rsm_send(ipc_handle, &is,
5620                            NULL);

5622                        if (e != RSM_SUCCESS) {
5623                                mutex_enter(&path->mutex);
5624                                /*
5625                                 * release the reserved msgbuf since
5626                                 * the send failed
5627                                 */
5628                                sendq_token->msgbuf_avail++;
5629                                cv_broadcast(&sendq_token->sendq_cv);
5630                                mutex_exit(&path->mutex);
5631                        }
5632                } else
5633                        e = adapter->rsmpi_ops->rsm_send(ipc_handle, &is,
5634                            NULL);

5636                if (e != RSM_SUCCESS) {
5637                        DBG_PRINTF((category, RSM_ERR,
5638                            "rsm: rsmipc_send rsmpi send err = %d\n", e));
5639                        RSMIPC_CLEAR(rslot, RSMIPC_PENDING);
5640                        rsmipc_free(rslot);
5641                        rele_sendq_token(sendq_token);
5642                        atomic_inc_64(&rsm_ipcsend_errcnt);
5642                        atomic_add_64(&rsm_ipcsend_errcnt, 1);
5643                        goto again;
5644                }

5646                /* wait for a reply signal, a SIGINT, or 5 sec. timeout */
5647                e = cv_reltimedwait_sig(&rslot->rsmipc_cv, &rslot->rsmipc_lock,
5648                    drv_usectohz(5000000), TR_CLOCK_TICK);
5649                if (e < 0) {
5650                        /* timed out - retry */
5651                        e = RSMERR_TIMEOUT;
5652                } else if (e == 0) {
5653                        /* signalled - return error */
5654                        e = RSMERR_INTERRUPTED;
5655                        break;
5656                } else {
5657                        e = RSM_SUCCESS;
5658                }
5659        }

5661        RSMIPC_CLEAR(rslot, RSMIPC_PENDING);
5662        rsmipc_free(rslot);
5663        rele_sendq_token(sendq_token);

5665        DBG_PRINTF((category, RSM_DEBUG_VERBOSE, "rsmipc_send done=%d\n", e));
5666        return (e);
5667 }
```

*unchanged_portion_omitted_*

```
5866 /*
5867  * This function takes path and sends a message using the sendq
5868  * corresponding to it. The RSMIPC_MSG_SQREADY, RSMIPC_MSG_SQREADY_ACK
```



```
5869  * and RSMIPC_MSG_CREDIT are sent using this function.
5870  */
5871 int
5872 rsmipc_send_controlmsg(path_t *path, int msgtype)
5873 {
5874         int                     e;
5875         int                     retry_cnt = 0;
5876         int                     min_retry_cnt = 10;
5877         adapter_t               *adapter;
5878         rsm_send_t              is;
5879         rsm_send_q_handle_t     ipc_handle;
5880         rsmipc_controlmsg_t     msg;
5881         DBG_DEFINE(category, RSM_KERNEL_AGENT | RSM_FUNC_ALL | RSM_FLOWCONTROL);

5883         DBG_PRINTF((category, RSM_DEBUG_VERBOSE,
5884             "rsmipc_send_controlmsg enter\n"));

5886         ASSERT(MUTEX_HELD(&path->mutex));

5888         adapter = path->local_adapter;

5890         DBG_PRINTF((category, RSM_DEBUG, "rsmipc_send_controlmsg:path=%lx "
5891             "msgtype=%d %lx:%llx->%lx:%llx procmsg=%d\n", path, msgtype,
5892             my_nodeid, adapter->hwaddr, path->remote_node,
5893             path->remote_hwaddr, path->procmsg_cnt));

5895         if (path->state != RSMKA_PATH_ACTIVE) {
5896                 DBG_PRINTF((category, RSM_DEBUG_VERBOSE,
5897                     "rsmipc_send_controlmsg done: ! RSMKA_PATH_ACTIVE"));
5898                 return (1);
5899         }

5901         ipc_handle = path->sendq_token.rsmpi_sendq_handle;

5903         msg.rsmipc_hdr.rsmipc_version = RSM_VERSION;
5904         msg.rsmipc_hdr.rsmipc_src = my_nodeid;
5905         msg.rsmipc_hdr.rsmipc_type = msgtype;
5906         msg.rsmipc_hdr.rsmipc_incn = path->remote_incn;

5908         if (msgtype == RSMIPC_MSG_CREDIT)
5909                 msg.rsmipc_credits = path->procmsg_cnt;

5911         msg.rsmipc_local_incn = path->local_incn;

5913         msg.rsmipc_adapter_hwaddr = adapter->hwaddr;
5914         /* incr the sendq, path refcnt */
5915         PATH_HOLD_NOLOCK(path);
5916         SENDQ_TOKEN_HOLD(path);

5918         do {
5919                 /* drop the path lock before doing the rsm_send */
5920                 mutex_exit(&path->mutex);

5922                 is.is_data = (void *)&msg;
5923                 is.is_size = sizeof (msg);
5924                 is.is_flags = RSM_INTR_SEND_DELIVER | RSM_INTR_SEND_SLEEP;
5925                 is.is_wait = 0;

5927                 e = adapter->rsmpi_ops->rsm_send(ipc_handle, &is, NULL);

5929                 ASSERT(e != RSMERR_QUEUE_FENCE_UP &&
5930                     e != RSMERR_BAD_BARRIER_HNDL);

5932                 mutex_enter(&path->mutex);

5934                 if (e == RSM_SUCCESS) {
```

```
5935                            break;
5936                    }
5937                    /* error counter for statistics */
5938                    atomic_inc_64(&rsm_ctrlmsg_errcnt);
5938                    atomic_add_64(&rsm_ctrlmsg_errcnt, 1);

5940                    DBG_PRINTF((category, RSM_ERR,
5941                        "rsmipc_send_controlmsg:rsm_send error=%d", e));

5943                    if (++retry_cnt == min_retry_cnt) { /* backoff before retry */
5944                            (void) cv_reltimedwait(&path->sendq_token.sendq_cv,
5945                                &path->mutex, drv_usectohz(10000), TR_CLOCK_TICK);
5946                            retry_cnt = 0;
5947                    }
5948            } while (path->state == RSMKA_PATH_ACTIVE);

5950            /* decrement the sendq,path refcnt that we incr before rsm_send */
5951            SENDQ_TOKEN_RELE(path);
5952            PATH_RELE_NOLOCK(path);

5954            DBG_PRINTF((category, RSM_DEBUG_VERBOSE,
5955                "rsmipc_send_controlmsg done=%d", e));
5956            return (e);
5957 }
______unchanged_portion_omitted_


6085 static int
6086 rsm_connect(rsmseg_t *seg, rsm_ioctlmsg_t *msg, cred_t *cred,
6087     intptr_t dataptr, int mode)
6088 {
6089            int e;
6090            int                     recheck_state = 0;
6091            void                    *shared_cookie;
6092            rsmipc_request_t        request;
6093            rsmipc_reply_t          reply;
6094            rsm_permission_t        access;
6095            adapter_t               *adapter;
6096            rsm_addr_t              addr = 0;
6097            rsm_import_share_t      *sharedp;
6098            DBG_DEFINE(category, RSM_KERNEL_AGENT | RSM_IMPORT);

6100            DBG_PRINTF((category, RSM_DEBUG_VERBOSE, "rsm_connect enter\n"));

6102            adapter = rsm_getadapter(msg, mode);
6103            if (adapter == NULL) {
6104                    DBG_PRINTF((category, RSM_ERR,
6105                        "rsm_connect done:ENODEV adapter=NULL\n"));
6106                    return (RSMERR_CTLR_NOT_PRESENT);
6107            }

6109            if ((adapter == &loopback_adapter) && (msg->nodeid != my_nodeid)) {
6110                    rsmka_release_adapter(adapter);
6111                    DBG_PRINTF((category, RSM_ERR,
6112                        "rsm_connect done:ENODEV loopback\n"));
6113                    return (RSMERR_CTLR_NOT_PRESENT);
6114            }


6117            ASSERT(seg->s_hdr.rsmrc_type == RSM_RESOURCE_IMPORT_SEGMENT);
6118            ASSERT(seg->s_state == RSM_STATE_NEW);

6120            /*
6121             * Translate perm to access
6122             */
6123            if (msg->perm & ~RSM_PERM_RDWR) {
```



```
6124                    rsmka_release_adapter(adapter);
6125                    DBG_PRINTF((category, RSM_ERR,
6126                        "rsm_connect done:EINVAL invalid perms\n"));
6127                    return (RSMERR_BAD_PERMS);
6128            }
6129            access = 0;
6130            if (msg->perm & RSM_PERM_READ)
6131                    access |= RSM_ACCESS_READ;
6132            if (msg->perm & RSM_PERM_WRITE)
6133                    access |= RSM_ACCESS_WRITE;

6135            seg->s_node = msg->nodeid;

6137            /*
6138             * Adding to the import list locks the segment; release the segment
6139             * lock so we can get the reply for the send.
6140             */
6141            e = rsmimport_add(seg, msg->key);
6142            if (e) {
6143                    rsmka_release_adapter(adapter);
6144                    DBG_PRINTF((category, RSM_ERR,
6145                        "rsm_connect done:rsmimport_add failed %d\n", e));
6146                    return (e);
6147            }
6148            seg->s_state = RSM_STATE_CONNECTING;

6150            /*
6151             * Set the s_adapter field here so as to have a valid comparison of
6152             * the adapter and the s_adapter value during rsmshare_get. For
6153             * any error, set s_adapter to NULL before doing a release_adapter
6154             */
6155            seg->s_adapter = adapter;

6157            rsmseglock_release(seg);

6159            /*
6160             * get the pointer to the shared data structure; the
6161             * shared data is locked and refcount has been incremented
6162             */
6163            sharedp = rsmshare_get(msg->key, msg->nodeid, adapter, seg);

6165            ASSERT(rsmsharelock_held(seg));

6167            do {
6168                    /* flag indicates whether we need to recheck the state */
6169                    recheck_state = 0;
6170                    DBG_PRINTF((category, RSM_DEBUG_VERBOSE,
6171                        "rsm_connect:RSMSI_STATE=%d\n", sharedp->rsmsi_state));
6172                    switch (sharedp->rsmsi_state) {
6173                    case RSMSI_STATE_NEW:
6174                            sharedp->rsmsi_state = RSMSI_STATE_CONNECTING;
6175                            break;
6176                    case RSMSI_STATE_CONNECTING:
6177                            /* FALLTHRU */
6178                    case RSMSI_STATE_CONN_QUIESCE:
6179                            /* FALLTHRU */
6180                    case RSMSI_STATE_MAP_QUIESCE:
6181                            /* wait for the state to change */
6182                            while ((sharedp->rsmsi_state ==
6183                                RSMSI_STATE_CONNECTING) ||
6184                                (sharedp->rsmsi_state ==
6185                                RSMSI_STATE_CONN_QUIESCE) ||
6186                                (sharedp->rsmsi_state ==
6187                                RSMSI_STATE_MAP_QUIESCE)) {
6188                                    if (cv_wait_sig(&sharedp->rsmsi_cv,
6189                                        &sharedp->rsmsi_lock) == 0) {
```

```
6190                                    /* signalled - clean up and return */
6191                                    rsmsharelock_release(seg);
6192                                    rsmimport_rm(seg);
6193                                    seg->s_adapter = NULL;
6194                                    rsmka_release_adapter(adapter);
6195                                    seg->s_state = RSM_STATE_NEW;
6196                                    DBG_PRINTF((category, RSM_ERR,
6197                                        "rsm_connect done: INTERRUPTED\n"));
6198                                    return (RSMERR_INTERRUPTED);
6199                            }
6200                    }
6201                    /*
6202                     * the state changed, loop back and check what it is
6203                     */
6204                    recheck_state = 1;
6205                    break;
6206            case RSMSI_STATE_ABORT_CONNECT:
6207                    /* exit the loop and clean up further down */
6208                    break;
6209            case RSMSI_STATE_CONNECTED:
6210                    /* already connected, good - fall through */
6211            case RSMSI_STATE_MAPPED:
6212                    /* already mapped, wow - fall through */
6213                    /* access validation etc is done further down */
6214                    break;
6215            case RSMSI_STATE_DISCONNECTED:
6216                    /* disconnected - so reconnect now */
6217                    sharedp->rsmsi_state = RSMSI_STATE_CONNECTING;
6218                    break;
6219            default:
6220                    ASSERT(0); /* Invalid State */
6221            }
6222    } while (recheck_state);

6224    if (sharedp->rsmsi_state == RSMSI_STATE_CONNECTING) {
6225            /* we are the first to connect */
6226            rsmsharelock_release(seg);

6228            if (msg->nodeid != my_nodeid) {
6229                    addr = get_remote_hwaddr(adapter, msg->nodeid);

6231                    if ((int64_t)addr < 0) {
6232                            rsmsharelock_acquire(seg);
6233                            rsmsharecv_signal(seg, RSMSI_STATE_CONNECTING,
6234                                RSMSI_STATE_NEW);
6235                            rsmsharelock_release(seg);
6236                            rsmimport_rm(seg);
6237                            seg->s_adapter = NULL;
6238                            rsmka_release_adapter(adapter);
6239                            seg->s_state = RSM_STATE_NEW;
6240                            DBG_PRINTF((category, RSM_ERR,
6241                                "rsm_connect done: hwaddr<0\n"));
6242                            return (RSMERR_INTERNAL_ERROR);
6243                    }
6244            } else {
6245                    addr = adapter->hwaddr;
6246            }

6248            /*
6249             * send request to node [src, dest, key, msgid] and get back
6250             * [status, msgid, cookie]
6251             */
6252            request.rsmipc_key = msg->key;
6253            /*
6254             * we need the s_mode of the exporter so pass
6255             * RSM_ACCESS_TRUSTED
```



```
6256             */
6257            request.rsmipc_perm = RSM_ACCESS_TRUSTED;
6258            request.rsmipc_hdr.rsmipc_type = RSMIPC_MSG_SEGCONNECT;
6259            request.rsmipc_adapter_hwaddr = addr;
6260            request.rsmipc_segment_cookie = sharedp;

6262            e = (int)rsmipc_send(msg->nodeid, &request, &reply);
6263            if (e) {
6264                    rsmsharelock_acquire(seg);
6265                    rsmsharecv_signal(seg, RSMSI_STATE_CONNECTING,
6266                        RSMSI_STATE_NEW);
6267                    rsmsharelock_release(seg);
6268                    rsmimport_rm(seg);
6269                    seg->s_adapter = NULL;
6270                    rsmka_release_adapter(adapter);
6271                    seg->s_state = RSM_STATE_NEW;
6272                    DBG_PRINTF((category, RSM_ERR,
6273                        "rsm_connect done:rsmipc_send failed %d\n", e));
6274                    return (e);
6275            }

6277            if (reply.rsmipc_status != RSM_SUCCESS) {
6278                    rsmsharelock_acquire(seg);
6279                    rsmsharecv_signal(seg, RSMSI_STATE_CONNECTING,
6280                        RSMSI_STATE_NEW);
6281                    rsmsharelock_release(seg);
6282                    rsmimport_rm(seg);
6283                    seg->s_adapter = NULL;
6284                    rsmka_release_adapter(adapter);
6285                    seg->s_state = RSM_STATE_NEW;
6286                    DBG_PRINTF((category, RSM_ERR,
6287                        "rsm_connect done:rsmipc_send reply err %d\n",
6288                        reply.rsmipc_status));
6289                    return (reply.rsmipc_status);
6290            }

6292            rsmsharelock_acquire(seg);
6293            /* store the information recvd into the shared data struct */
6294            sharedp->rsmsi_mode = reply.rsmipc_mode;
6295            sharedp->rsmsi_uid = reply.rsmipc_uid;
6296            sharedp->rsmsi_gid = reply.rsmipc_gid;
6297            sharedp->rsmsi_seglen = reply.rsmipc_seglen;
6298            sharedp->rsmsi_cookie = sharedp;
6299    }

6301    rsmsharelock_release(seg);

6303    /*
6304     * Get the segment lock and check for a force disconnect
6305     * from the export side which would have changed the state
6306     * back to RSM_STATE_NEW. Once the segment lock is acquired a
6307     * force disconnect will be held off until the connection
6308     * has completed.
6309     */
6310    rsmseglock_acquire(seg);
6311    rsmsharelock_acquire(seg);
6312    ASSERT(seg->s_state == RSM_STATE_CONNECTING ||
6313        seg->s_state == RSM_STATE_ABORT_CONNECT);

6315    shared_cookie = sharedp->rsmsi_cookie;

6317    if ((seg->s_state == RSM_STATE_ABORT_CONNECT) ||
6318        (sharedp->rsmsi_state == RSMSI_STATE_ABORT_CONNECT)) {
6319            seg->s_state = RSM_STATE_NEW;
6320            seg->s_adapter = NULL;
6321            rsmsharelock_release(seg);
```

```
6322                rsmseglock_release(seg);
6323                rsmimport_rm(seg);
6324                rsmka_release_adapter(adapter);

6326                rsmsharelock_acquire(seg);
6327                if (!(sharedp->rsmsi_flags & RSMSI_FLAGS_ABORTDONE)) {
6328                        /*
6329                         * set a flag indicating abort handling has been
6330                         * done
6331                         */
6332                        sharedp->rsmsi_flags |= RSMSI_FLAGS_ABORTDONE;
6333                        rsmsharelock_release(seg);
6334                        /* send a message to exporter - only once */
6335                        (void) rsm_send_notimporting(msg->nodeid,
6336                            msg->key, shared_cookie);
6337                        rsmsharelock_acquire(seg);
6338                        /*
6339                         * wake up any waiting importers and inform that
6340                         * connection has been aborted
6341                         */
6342                        cv_broadcast(&sharedp->rsmsi_cv);
6343                }
6344                rsmsharelock_release(seg);

6346                DBG_PRINTF((category, RSM_ERR,
6347                    "rsm_connect done: RSM_STATE_ABORT_CONNECT\n"));
6348                return (RSMERR_INTERRUPTED);
6349        }


6352        /*
6353         * We need to verify that this process has access
6354         */
6355        e = rsm_access(sharedp->rsmsi_uid, sharedp->rsmsi_gid,
6356            access & sharedp->rsmsi_mode,
6357            (int)(msg->perm & RSM_PERM_RDWR), cred);
6358        if (e) {
6359                rsmsharelock_release(seg);
6360                seg->s_state = RSM_STATE_NEW;
6361                seg->s_adapter = NULL;
6362                rsmseglock_release(seg);
6363                rsmimport_rm(seg);
6364                rsmka_release_adapter(adapter);
6365                /*
6366                 * No need to lock segment it has been removed
6367                 * from the hash table
6368                 */
6369                rsmsharelock_acquire(seg);
6370                if (sharedp->rsmsi_state == RSMSI_STATE_CONNECTING) {
6371                        rsmsharelock_release(seg);
6372                        /* this is the first importer */

6374                        (void) rsm_send_notimporting(msg->nodeid, msg->key,
6375                            shared_cookie);
6376                        rsmsharelock_acquire(seg);
6377                        sharedp->rsmsi_state = RSMSI_STATE_NEW;
6378                        cv_broadcast(&sharedp->rsmsi_cv);
6379                }
6380                rsmsharelock_release(seg);

6382                DBG_PRINTF((category, RSM_ERR,
6383                    "rsm_connect done: ipcaccess failed\n"));
6384                return (RSMERR_PERM_DENIED);
6385        }

6387        /* update state and cookie */
```



```
6388        seg->s_segid = sharedp->rsmsi_segid;
6389        seg->s_len = sharedp->rsmsi_seglen;
6390        seg->s_mode = access & sharedp->rsmsi_mode;
6391        seg->s_pid = ddi_get_pid();
6392        seg->s_mapinfo = NULL;

6394        if (seg->s_node != my_nodeid) {
6395                if (sharedp->rsmsi_state == RSMSI_STATE_CONNECTING) {
6396                        e = adapter->rsmpi_ops->rsm_connect(
6397                            adapter->rsmpi_handle,
6398                            addr, seg->s_segid, &sharedp->rsmsi_handle);

6400                        if (e != RSM_SUCCESS) {
6401                                seg->s_state = RSM_STATE_NEW;
6402                                seg->s_adapter = NULL;
6403                                rsmsharelock_release(seg);
6404                                rsmseglock_release(seg);
6405                                rsmimport_rm(seg);
6406                                rsmka_release_adapter(adapter);
6407                                /*
6408                                 *  inform the exporter to delete this importer
6409                                 */
6410                                (void) rsm_send_notimporting(msg->nodeid,
6411                                    msg->key, shared_cookie);

6413                                /*
6414                                 * Now inform any waiting importers to
6415                                 * retry connect. This needs to be done
6416                                 * after sending notimporting so that
6417                                 * the notimporting is sent before a waiting
6418                                 * importer sends a segconnect while retrying
6419                                 *
6420                                 * No need to lock segment it has been removed
6421                                 * from the hash table
6422                                 */

6424                                rsmsharelock_acquire(seg);
6425                                sharedp->rsmsi_state = RSMSI_STATE_NEW;
6426                                cv_broadcast(&sharedp->rsmsi_cv);
6427                                rsmsharelock_release(seg);

6429                                DBG_PRINTF((category, RSM_ERR,
6430                                    "rsm_connect error %d\n", e));
6431                                if (e == RSMERR_SEG_NOT_PUBLISHED_TO_RSM_ADDR)
6432                                        return (
6433                                            RSMERR_SEG_NOT_PUBLISHED_TO_NODE);
6434                                else if ((e == RSMERR_RSM_ADDR_UNREACHABLE) ||
6435                                    (e == RSMERR_UNKNOWN_RSM_ADDR))
6436                                        return (RSMERR_REMOTE_NODE_UNREACHABLE);
6437                                else
6438                                        return (e);
6439                        }

6441                }
6442                seg->s_handle.in = sharedp->rsmsi_handle;

6444        }

6446        seg->s_state = RSM_STATE_CONNECT;


6449        seg->s_flags &= ~RSM_IMPORT_DUMMY;      /* clear dummy flag */
6450        if (bar_va) {
6451                /* increment generation number on barrier page */
6452                atomic_inc_16(bar_va + seg->s_hdr.rsmrc_num);
6452                atomic_add_16(bar_va + seg->s_hdr.rsmrc_num, 1);
```

```
6453                 /* return user off into barrier page where status will be */
6454                 msg->off = (int)seg->s_hdr.rsmrc_num;
6455                 msg->gnum = bar_va[msg->off];   /* gnum race */
6456         } else {
6457                 msg->off = 0;
6458                 msg->gnum = 0;  /* gnum race */
6459         }

6461         msg->len = (int)sharedp->rsmsi_seglen;
6462         msg->rnum = seg->s_minor;
6463         rsmsharecv_signal(seg, RSMSI_STATE_CONNECTING, RSMSI_STATE_CONNECTED);
6464         rsmsharelock_release(seg);
6465         rsmseglock_release(seg);

6467         /* Return back to user the segment size & perm in case it's needed */

6469 #ifdef _MULTI_DATAMODEL
6470         if ((mode & DATAMODEL_MASK) == DATAMODEL_ILP32) {
6471                 rsm_ioctlmsg32_t msg32;

6473                 if (msg->len > UINT_MAX)
6474                         msg32.len = RSM_MAXSZ_PAGE_ALIGNED;
6475                 else
6476                         msg32.len = msg->len;
6477                 msg32.off = msg->off;
6478                 msg32.perm = msg->perm;
6479                 msg32.gnum = msg->gnum;
6480                 msg32.rnum = msg->rnum;

6482                 DBG_PRINTF((category, RSM_DEBUG_VERBOSE,
6483                     "rsm_connect done\n"));

6485                 if (ddi_copyout((caddr_t)&msg32, (caddr_t)dataptr,
6486                     sizeof (msg32), mode))
6487                         return (RSMERR_BAD_ADDR);
6488                 else
6489                         return (RSM_SUCCESS);
6490         }
6491 #endif
6492         DBG_PRINTF((category, RSM_DEBUG_VERBOSE, "rsm_connect done\n"));

6494         if (ddi_copyout((caddr_t)msg, (caddr_t)dataptr, sizeof (*msg),
6495             mode))
6496                 return (RSMERR_BAD_ADDR);
6497         else
6498                 return (RSM_SUCCESS);
6499 }
```

unchanged_portion_omitted_

```
6595 /*
6596  * cookie returned here if not null indicates that it is
6597  * the last importer and it can be used in the RSMIPC_NOT_IMPORTING
6598  * message.
6599  */
6600 static int
6601 rsm_closeconnection(rsmseg_t *seg, void **cookie)
6602 {
6603         int                     e;
6604         adapter_t               *adapter;
6605         rsm_import_share_t      *sharedp;
6606         DBG_DEFINE(category, RSM_KERNEL_AGENT | RSM_IMPORT);

6608         DBG_PRINTF((category, RSM_DEBUG_VERBOSE,
6609             "rsm_closeconnection enter\n"));

6611         *cookie = (void *)NULL;
```



```
6613         ASSERT(seg->s_hdr.rsmrc_type == RSM_RESOURCE_IMPORT_SEGMENT);

6615         /* assert seg is locked */
6616         ASSERT(rsmseglock_held(seg));

6618         if (seg->s_state == RSM_STATE_DISCONNECT) {
6619                 DBG_PRINTF((category, RSM_DEBUG_VERBOSE,
6620                     "rsm_closeconnection done: already disconnected\n"));
6621                 return (RSM_SUCCESS);
6622         }

6624         /* wait for all putv/getv ops to get done */
6625         while (seg->s_rdmacnt > 0) {
6626                 cv_wait(&seg->s_cv, &seg->s_lock);
6627         }

6629         (void) rsm_unmap(seg);

6631         ASSERT(seg->s_state == RSM_STATE_CONNECT ||
6632             seg->s_state == RSM_STATE_CONN_QUIESCE);

6634         adapter = seg->s_adapter;
6635         sharedp = seg->s_share;

6637         ASSERT(sharedp != NULL);

6639         rsmsharelock_acquire(seg);

6641         /*
6642          * Disconnect on adapter
6643          *
6644          * The current algorithm is stateless, I don't have to contact
6645          * server when I go away. He only gives me permissions. Of course,
6646          * the adapters will talk to terminate the connect.
6647          *
6648          * disconnect is needed only if we are CONNECTED not in CONN_QUIESCE
6649          */
6650         if ((sharedp->rsmsi_state == RSMSI_STATE_CONNECTED) &&
6651             (sharedp->rsmsi_node != my_nodeid)) {

6653                 if (sharedp->rsmsi_refcnt == 1) {
6654                         /* this is the last importer */
6655                         ASSERT(sharedp->rsmsi_mapcnt == 0);

6657                         e = adapter->rsmpi_ops->
6658                             rsm_disconnect(sharedp->rsmsi_handle);
6659                         if (e != RSM_SUCCESS) {
6660                                 DBG_PRINTF((category, RSM_DEBUG,
6661                                     "rsm:disconnect failed seg=%x:err=%d\n",
6662                                     seg->s_key, e));
6663                         }
6664                 }
6665         }

6667         seg->s_handle.in = NULL;

6669         sharedp->rsmsi_refcnt--;

6671         if (sharedp->rsmsi_refcnt == 0) {
6672                 *cookie = (void *)sharedp->rsmsi_cookie;
6673                 sharedp->rsmsi_state = RSMSI_STATE_DISCONNECTED;
6674                 sharedp->rsmsi_handle = NULL;
6675                 rsmsharelock_release(seg);

6677                 /* clean up the shared data structure */
```

```
6678                mutex_destroy(&sharedp->rsmsi_lock);
6679                cv_destroy(&sharedp->rsmsi_cv);
6680                kmem_free((void *)(sharedp), sizeof (rsm_import_share_t));

6682        } else {
6683                rsmsharelock_release(seg);
6684        }

6686        /* increment generation number on barrier page */
6687        if (bar_va) {
6688                atomic_inc_16(bar_va + seg->s_hdr.rsmrc_num);
6688                atomic_add_16(bar_va + seg->s_hdr.rsmrc_num, 1);
6689        }

6691        /*
6692         * The following needs to be done after any
6693         * rsmsharelock calls which use seg->s_share.
6694         */
6695        seg->s_share = NULL;

6697        seg->s_state = RSM_STATE_DISCONNECT;
6698        /* signal anyone waiting in the CONN_QUIESCE state */
6699        cv_broadcast(&seg->s_cv);

6701        DBG_PRINTF((category, RSM_DEBUG_VERBOSE,
6702            "rsm_closeconnection done\n"));

6704        return (RSM_SUCCESS);
6705 }
```

_____unchanged_portion_omitted_

```
7257 static int
7258 rsm_consumeevent_ioctl(caddr_t arg, int mode)
7259 {
7260        int     rc;
7261        int     i;
7262        minor_t rnum;
7263        rsm_consume_event_msg_t msg = {0};
7264        rsmseg_t                *seg;
7265        rsm_poll_event_t        *event_list;
7266        rsm_poll_event_t        events[RSM_MAX_POLLFDS];
7267        DBG_DEFINE(category, RSM_KERNEL_AGENT | RSM_IOCTL);

7269        event_list = events;

7271        if ((rc = rsm_consumeevent_copyin(arg, &msg, &event_list, mode)) !=
7272            RSM_SUCCESS) {
7273                return (rc);
7274        }

7276        for (i = 0; i < msg.numents; i++) {
7277                rnum = event_list[i].rnum;
7278                event_list[i].revent = 0;
7279                /* get the segment structure */
7280                seg = (rsmseg_t *)rsmresource_lookup(rnum, RSM_LOCK);
7281                if (seg) {
7282                        DBG_PRINTF((category, RSM_DEBUG_VERBOSE,
7283                            "consumeevent_ioctl: rnum(%d) seg(%p)\n", rnum,
7284                            seg));
7285                        if (seg->s_pollevent) {
7286                                /* consume the event */
7287                                atomic_dec_32(&seg->s_pollevent);
7287                                atomic_add_32(&seg->s_pollevent, -1);
7288                                event_list[i].revent = POLLRDNORM;
7289                        }
7290                        rsmseglock_release(seg);
```



```
7291                }
7292        }

7294        if ((rc = rsm_consumeevent_copyout(&msg, event_list, mode)) !=
7295            RSM_SUCCESS) {
7296                return (rc);
7297        }

7299        return (RSM_SUCCESS);
7300 }
```

_____unchanged_portion_omitted_

```
**********************************************************
    7049 Mon Jul 28 07:44:43 2014
new/usr/src/uts/common/io/str_conf.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/types.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/conf.h>
#include <sys/stream.h>
#include <sys/strsubr.h>
#include <sys/modctl.h>
#include <sys/modhash.h>
#include <sys/atomic.h>

#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/t_lock.h>

/*
 * This module provides the framework that manage STREAMS modules.
 * fmodsw_alloc() is called from modconf.c as a result of a module calling
 * mod_install() and fmodsw_free() is called as the result of the module
 * calling mod_remove().
 * fmodsw_find() will find the fmodsw_impl_t structure relating to a named
 * module. There is no equivalent of driver major numbers for modules; the
 * the database of fmodsw_impl_t structures is purely keyed by name and
 * is hence a hash table to keep lookup cost to a minimum.
 */

/*
 * fmodsw_hash is the hash table that will be used to map module names to
 * their fmodsw_impl_t structures. The hash function requires that the value is
 * a power of 2 so this definition specifies the log of the hash table size.
 */
#define FMODSW_LOG_HASHSZ       8

/*
```



```
 * Hash table and associated reader-writer lock
 *
 * NOTE: Because the lock is global data, it is initialized to zero and hence
 *       a call to rw_init() is not required. Similarly all the pointers in
 *       the hash table will be implicitly initialized to NULL.
 */
#define FMODSW_HASHSZ           (1 << FMODSW_LOG_HASHSZ)

static fmodsw_impl_t    *fmodsw_hash[FMODSW_HASHSZ];
static krwlock_t        fmodsw_lock;

/*
 * Debug code:
 *
 * This is not conditionally compiled since it may be useful to third
 * parties when developing new modules.
 */

#define BUFSZ   512

#define FMODSW_INIT             0x00000001
#define FMODSW_REGISTER         0x00000002
#define FMODSW_UNREGISTER       0x00000004
#define FMODSW_FIND             0x00000008

uint32_t        fmodsw_debug_flags = 0x00000000;

static void fmodsw_dprintf(uint_t flag, const char *fmt, ...) __KPRINTFLIKE(2);

/* PRINTFLIKE2 */
static void
i_fmodsw_dprintf(uint_t flag, const char *fmt, ...)
{
        va_list alist;
        char    buf[BUFSZ + 1];
        char    *ptr;

        if (fmodsw_debug_flags & flag) {
                va_start(alist, fmt);
                ptr = buf;
                (void) sprintf(ptr, "strmod debug: ");
                ptr += strlen(buf);
                (void) vsnprintf(ptr, buf + BUFSZ - ptr, fmt, alist);
                printf(buf);
                va_end(alist);
        }
}
```

_____unchanged_portion_omitted_

```
fmodsw_impl_t *
fmodsw_find(const char *name, fmodsw_flags_t flags)
{
        fmodsw_impl_t   *fp;
        int             id;

try_again:
        rw_enter(&fmodsw_lock, RW_READER);
        if (i_fmodsw_hash_find(name, &fp) == 0) {
                if (flags & FMODSW_HOLD) {
                        atomic_inc_32(&(fp->f_ref));    /* lock must be held */
                        atomic_add_32(&(fp->f_ref), 1); /* lock must be held */
                        ASSERT(fp->f_ref > 0);
                }

                rw_exit(&fmodsw_lock);
                return (fp);
```

```
291         }
292         rw_exit(&fmodsw_lock);

294         if (flags & FMODSW_LOAD) {
295                 if ((id = modload("strmod", (char *)name)) != -1) {
296                         i_fmodsw_dprintf(FMODSW_FIND,
297                             "module '%s' loaded: id = %d\n", name, id);
298                         goto try_again;
299                 }
300         }

302         return (NULL);
303 }

305 void
306 fmodsw_rele(fmodsw_impl_t *fp)
307 {
308         ASSERT(fp->f_ref > 0);
309         atomic_dec_32(&(fp->f_ref));
311         atomic_add_32(&(fp->f_ref), -1);
310 }
```

_____unchanged_portion_omitted_

```
**********************************************************
  161669 Mon Jul 28 07:44:43 2014
new/usr/src/uts/common/io/tl.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1135 /*
1136  * Endpoint reference management.
1137  */
1138 static void
1139 tl_refhold(tl_endpt_t *tep)
1140 {
1141         atomic_inc_32(&tep->te_refcnt);
1141         atomic_add_32(&tep->te_refcnt, 1);
1142 }

1144 static void
1145 tl_refrele(tl_endpt_t *tep)
1146 {
1147         ASSERT(tep->te_refcnt != 0);

1149         if (atomic_dec_32_nv(&tep->te_refcnt) == 0)
1149         if (atomic_add_32_nv(&tep->te_refcnt, -1) == 0)
1150                 tl_free(tep);
1151 }
______unchanged_portion_omitted_

1264 static void
1265 tl_serializer_refhold(tl_serializer_t *s)
1266 {
1267         atomic_inc_32(&s->ts_refcnt);
1267         atomic_add_32(&s->ts_refcnt, 1);
1268 }

1270 static void
1271 tl_serializer_refrele(tl_serializer_t *s)
1272 {
1273         if (atomic_dec_32_nv(&s->ts_refcnt) == 0) {
1273         if (atomic_add_32_nv(&s->ts_refcnt, -1) == 0) {
1274                 serializer_destroy(s->ts_serializer);
1275                 kmem_free(s, sizeof (tl_serializer_t));
1276         }
1277 }
______unchanged_portion_omitted_

5348 /*
5349  * Generate a free addr and return it in struct pointed by ap
5350  * but allocating space for address buffer.
5351  * The generated address will be at least 4 bytes long and, if req->ta_alen
5352  * exceeds 4 bytes, be req->ta_alen bytes long.
5353  *
5354  * If address is found it will be inserted in the hash.
5355  *
5356  * If req->ta_alen is larger than the default alen (4 bytes) the last
5357  * alen-4 bytes will always be the same as in req.
5358  *
5359  * Return 0 for failure.
5360  * Return non-zero for success.
5361  */
5362 static boolean_t
5363 tl_get_any_addr(tl_endpt_t *tep, tl_addr_t *req)
5364 {
5365         t_scalar_t      alen;
5366         uint32_t        loopcnt;        /* Limit loop to 2^32 */
```



```
5368         ASSERT(tep->te_hash_hndl != NULL);
5369         ASSERT(! IS_SOCKET(tep));

5371         if (tep->te_hash_hndl == NULL)
5372                 return (B_FALSE);

5374         /*
5375          * check if default addr is in use
5376          * if it is - bump it and try again
5377          */
5378         if (req == NULL) {
5379                 alen = sizeof (uint32_t);
5380         } else {
5381                 alen = max(req->ta_alen, sizeof (uint32_t));
5382                 ASSERT(tep->te_zoneid == req->ta_zoneid);
5383         }

5385         if (tep->te_alen < alen) {
5386                 void *abuf = kmem_zalloc((size_t)alen, KM_NOSLEEP);

5388                 /*
5389                  * Not enough space in tep->ta_ap to hold the address,
5390                  * allocate a bigger space.
5391                  */
5392                 if (abuf == NULL)
5393                         return (B_FALSE);

5395                 if (tep->te_alen > 0)
5396                         kmem_free(tep->te_abuf, tep->te_alen);

5398                 tep->te_alen = alen;
5399                 tep->te_abuf = abuf;
5400         }

5402         /* Copy in the address in req */
5403         if (req != NULL) {
5404                 ASSERT(alen >= req->ta_alen);
5405                 bcopy(req->ta_abuf, tep->te_abuf, (size_t)req->ta_alen);
5406         }

5408         /*
5409          * First try minor number then try default addresses.
5410          */
5411         bcopy(&tep->te_minor, tep->te_abuf, sizeof (uint32_t));

5413         for (loopcnt = 0; loopcnt < UINT32_MAX; loopcnt++) {
5414                 if (mod_hash_insert_reserve(tep->te_addrhash,
5415                     (mod_hash_key_t)&tep->te_ap, (mod_hash_val_t)tep,
5416                     tep->te_hash_hndl) == 0) {
5417                         /*
5418                          * found free address
5419                          */
5420                         tep->te_flag |= TL_ADDRHASHED;
5421                         tep->te_hash_hndl = NULL;

5423                         return (B_TRUE); /* successful return */
5424                 }
5425                 /*
5426                  * Use default address.
5427                  */
5428                 bcopy(&tep->te_defaddr, tep->te_abuf, sizeof (uint32_t));
5429                 atomic_inc_32(&tep->te_defaddr);
5429                 atomic_add_32(&tep->te_defaddr, 1);
5430         }

5432         /*
```

```
5433          * Failed to find anything.
5434          */
5435         (void) (STRLOG(TL_ID, -1, 1, SL_ERROR,
5436             "tl_get_any_addr:looped 2^32 times"));
5437         return (B_FALSE);
5438 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   45335 Mon Jul 28 07:44:43 2014
new/usr/src/uts/common/io/usb/usba/usbai_pipe_mgmt.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_


 423 /*
 424  * pipe management
 425  *      utility functions to init and destroy a pipehandle
 426  */
 427 static int
 428 usba_init_pipe_handle(dev_info_t *dip,
 429         usba_device_t           *usba_device,
 430         usb_ep_descr_t          *ep,
 431         usb_pipe_policy_t       *pipe_policy,
 432         usba_ph_impl_t          *ph_impl)
 433 {
 434         int instance = ddi_get_instance(dip);
 435         unsigned int def_instance = instance;
 436         static unsigned int anon_instance = 0;
 437         char tq_name[TASKQ_NAMELEN];

 439         usba_pipe_handle_data_t *ph_data = ph_impl->usba_ph_data;
 440         ddi_iblock_cookie_t     iblock_cookie =
 441             usba_hcdi_get_hcdi(usba_device->usb_root_hub_dip)->
 442             hcdi_iblock_cookie;

 444         USB_DPRINTF_L4(DPRINT_MASK_USBAI, usbai_log_handle,
 445             "usba_init_pipe_handle: "
 446             "usba_device=0x%p ep=0x%x", (void *)usba_device,
 447             ep->bEndpointAddress);
 448         mutex_init(&ph_data->p_mutex, NULL, MUTEX_DRIVER, iblock_cookie);

 450         /* just to keep warlock happy, there is no contention yet */
 451         mutex_enter(&ph_data->p_mutex);
 452         mutex_enter(&usba_device->usb_mutex);

 454         ASSERT(pipe_policy->pp_max_async_reqs);

 456         if (instance != -1) {
 457                 (void) snprintf(tq_name, sizeof (tq_name),
 458                     "USB_%s_%x_pipehndl_tq_%d",
 459                     ddi_driver_name(dip), ep->bEndpointAddress, instance);
 460         } else {
 461                 def_instance = atomic_inc_32_nv(&anon_instance);
 461                 def_instance = atomic_add_32_nv(&anon_instance, 1);

 463                 (void) snprintf(tq_name, sizeof (tq_name),
 464                     "USB_%s_%x_pipehndl_tq_%d_",
 465                     ddi_driver_name(dip), ep->bEndpointAddress, def_instance);
 466         }

 468         ph_data->p_taskq = taskq_create(tq_name,
 469             pipe_policy->pp_max_async_reqs + 1,
 470             ((ep->bmAttributes & USB_EP_ATTR_MASK) ==
 471             USB_EP_ATTR_ISOCH) ?
 472             (maxclsyspri - 5) : minclsyspri,
 473             2 * (pipe_policy->pp_max_async_reqs + 1),
 474             8 * (pipe_policy->pp_max_async_reqs + 1),
 475             TASKQ_PREPOPULATE);

 477         /*
 478          * Create a shared taskq.
 479          */
```



```
 480         if (ph_data->p_spec_flag & USBA_PH_FLAG_TQ_SHARE) {
 481                 int iface = usb_get_if_number(dip);
 482                 if (iface < 0) {
 483                         /* we own the device, use first entry */
 484                         iface = 0;
 485                 }

 487                 if (instance != -1) {
 488                         (void) snprintf(tq_name, sizeof (tq_name),
 489                             "USB_%s_%x_shared_tq_%d",
 490                             ddi_driver_name(dip), ep->bEndpointAddress,
 491                             instance);
 492                 } else {
 493                         (void) snprintf(tq_name, sizeof (tq_name),
 494                             "USB_%s_%x_shared_tq_%d_",
 495                             ddi_driver_name(dip), ep->bEndpointAddress,
 496                             def_instance);
 497                 }

 499                 if (usba_device->usb_shared_taskq_ref_count[iface] == 0) {
 500                         usba_device->usb_shared_taskq[iface] =
 501                             taskq_create(tq_name,
 502                             1,                          /* Number threads. */
 503                             maxclsyspri - 5,            /* Priority */
 504                             1,                          /* minalloc */
 505                             USBA_N_ENDPOINTS + 4,       /* maxalloc */
 506                             TASKQ_PREPOPULATE);
 507                         ASSERT(usba_device->usb_shared_taskq[iface] != NULL);
 508                 }
 509                 usba_device->usb_shared_taskq_ref_count[iface]++;
 510         }

 512         ph_data->p_dip          = dip;
 513         ph_data->p_usba_device  = usba_device;
 514         ph_data->p_ep           = *ep;
 515         ph_data->p_ph_impl      = ph_impl;
 516         if ((ep->bmAttributes & USB_EP_ATTR_MASK) ==
 517             USB_EP_ATTR_ISOCH) {
 518                 ph_data->p_spec_flag |= USBA_PH_FLAG_USE_SOFT_INTR;
 519         }

 521         /* fix up the MaxPacketSize if it is the default endpoint descr */
 522         if ((ep == &usba_default_ep_descr) && usba_device) {
 523                 uint16_t        maxpktsize;

 525                 maxpktsize = usba_device->usb_dev_descr->bMaxPacketSize0;
 526                 if (usba_device->usb_is_wireless) {
 527                         /*
 528                          * according to wusb 1.0 spec 4.8.1, the host must
 529                          * assume a wMaxPacketSize of 512 for the default
 530                          * control pipe of a wusb device
 531                          */
 532                         maxpktsize = 0x200;
 533                 }
 534                 USB_DPRINTF_L3(DPRINT_MASK_USBAI, usbai_log_handle,
 535                     "adjusting max packet size from %d to %d",
 536                     ph_data->p_ep.wMaxPacketSize, maxpktsize);

 538                 ph_data->p_ep.wMaxPacketSize = maxpktsize;
 539         }

 541         /* now update usba_ph_impl structure */
 542         mutex_enter(&ph_impl->usba_ph_mutex);
 543         ph_impl->usba_ph_dip = dip;
 544         ph_impl->usba_ph_ep = ph_data->p_ep;
 545         ph_impl->usba_ph_policy = ph_data->p_policy = *pipe_policy;
```

```
 546         mutex_exit(&ph_impl->usba_ph_mutex);

 548         usba_init_list(&ph_data->p_queue, (usb_opaque_t)ph_data, iblock_cookie);
 549         usba_init_list(&ph_data->p_cb_queue, (usb_opaque_t)ph_data,
 550             iblock_cookie);
 551         mutex_exit(&usba_device->usb_mutex);
 552         mutex_exit(&ph_data->p_mutex);

 554         return (USB_SUCCESS);
 555 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   69689 Mon Jul 28 07:44:43 2014
new/usr/src/uts/common/io/xge/drv/xgell.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 941 mblk_t *
 942 xgell_ring_tx(void *arg, mblk_t *mp)
 943 {
 944         xgell_tx_ring_t *ring = (xgell_tx_ring_t *)arg;
 945         mblk_t *bp;
 946         xgelldev_t *lldev = ring->lldev;
 947         xge_hal_device_t *hldev = lldev->devh;
 948         xge_hal_status_e status;
 949         xge_hal_dtr_h dtr;
 950         xgell_txd_priv_t *txd_priv;
 951         uint32_t hckflags;
 952         uint32_t lsoflags;
 953         uint32_t mss;
 954         int handle_cnt, frag_cnt, ret, i, copied;
 955         boolean_t used_copy;
 956         uint64_t sent_bytes;

 958 _begin:
 959         handle_cnt = frag_cnt = 0;
 960         sent_bytes = 0;

 962         if (!lldev->is_initialized || lldev->in_reset)
 963                 return (mp);

 965         /*
 966          * If the free Tx dtrs count reaches the lower threshold,
 967          * inform the gld to stop sending more packets till the free
 968          * dtrs count exceeds higher threshold. Driver informs the
 969          * gld through gld_sched call, when the free dtrs count exceeds
 970          * the higher threshold.
 971          */
 972         if (xge_hal_channel_dtr_count(ring->channelh)
 973             <= XGELL_TX_LEVEL_LOW) {
 974                 xge_debug_ll(XGE_TRACE, "%s%d: queue %d: err on xmit,"
 975                     "free descriptors count at low threshold %d",
 976                     XGELL_IFNAME, lldev->instance,
 977                     ((xge_hal_channel_t *)ring->channelh)->post_qid,
 978                     XGELL_TX_LEVEL_LOW);
 979                 goto _exit;
 980         }

 982         status = xge_hal_fifo_dtr_reserve(ring->channelh, &dtr);
 983         if (status != XGE_HAL_OK) {
 984                 switch (status) {
 985                 case XGE_HAL_INF_CHANNEL_IS_NOT_READY:
 986                         xge_debug_ll(XGE_ERR,
 987                             "%s%d: channel %d is not ready.", XGELL_IFNAME,
 988                             lldev->instance,
 989                             ((xge_hal_channel_t *)
 990                             ring->channelh)->post_qid);
 991                         goto _exit;
 992                 case XGE_HAL_INF_OUT_OF_DESCRIPTORS:
 993                         xge_debug_ll(XGE_TRACE, "%s%d: queue %d: error in xmit,"
 994                             " out of descriptors.", XGELL_IFNAME,
 995                             lldev->instance,
 996                             ((xge_hal_channel_t *)
 997                             ring->channelh)->post_qid);
 998                         goto _exit;
 999                 default:
```



```
1000                         return (mp);
1001                 }
1002         }

1004         txd_priv = xge_hal_fifo_dtr_private(dtr);
1005         txd_priv->mblk = mp;

1007         /*
1008          * VLAN tag should be passed down along with MAC header, so h/w needn't
1009          * do insertion.
1010          *
1011          * For NIC driver that has to strip and re-insert VLAN tag, the example
1012          * is the other implementation for xge. The driver can simple bcopy()
1013          * ether_vlan_header to overwrite VLAN tag and let h/w insert the tag
1014          * automatically, since it's impossible that GLD sends down mp(s) with
1015          * splited ether_vlan_header.
1016          *
1017          * struct ether_vlan_header *evhp;
1018          * uint16_t tci;
1019          *
1020          * evhp = (struct ether_vlan_header *)mp->b_rptr;
1021          * if (evhp->ether_tpid == htons(VLAN_TPID)) {
1022          *      tci = ntohs(evhp->ether_tci);
1023          *      (void) bcopy(mp->b_rptr, mp->b_rptr + VLAN_TAGSZ,
1024          *          2 * ETHERADDRL);
1025          *      mp->b_rptr += VLAN_TAGSZ;
1026          *
1027          *      xge_hal_fifo_dtr_vlan_set(dtr, tci);
1028          * }
1029          */

1031         copied = 0;
1032         used_copy = B_FALSE;
1033         for (bp = mp; bp != NULL; bp = bp->b_cont) {
1034                 int mblen;
1035                 uint_t ncookies;
1036                 ddi_dma_cookie_t dma_cookie;
1037                 ddi_dma_handle_t dma_handle;

1039                 /* skip zero-length message blocks */
1040                 mblen = MBLKL(bp);
1041                 if (mblen == 0) {
1042                         continue;
1043                 }

1045                 sent_bytes += mblen;

1047                 /*
1048                  * Check the message length to decide to DMA or bcopy() data
1049                  * to tx descriptor(s).
1050                  */
1051                 if (mblen < lldev->config.tx_dma_lowat &&
1052                     (copied + mblen) < lldev->tx_copied_max) {
1053                         xge_hal_status_e rc;
1054                         rc = xge_hal_fifo_dtr_buffer_append(ring->channelh,
1055                             dtr, bp->b_rptr, mblen);
1056                         if (rc == XGE_HAL_OK) {
1057                                 used_copy = B_TRUE;
1058                                 copied += mblen;
1059                                 continue;
1060                         } else if (used_copy) {
1061                                 xge_hal_fifo_dtr_buffer_finalize(
1062                                     ring->channelh, dtr, frag_cnt++);
1063                                 used_copy = B_FALSE;
1064                         }
1065                 } else if (used_copy) {
```

```
1066                    xge_hal_fifo_dtr_buffer_finalize(ring->channelh,
1067                        dtr, frag_cnt++);
1068                    used_copy = B_FALSE;
1069            }

1071            ret = ddi_dma_alloc_handle(lldev->dev_info, &tx_dma_attr,
1072                DDI_DMA_DONTWAIT, 0, &dma_handle);
1073            if (ret != DDI_SUCCESS) {
1074                    xge_debug_ll(XGE_ERR,
1075                        "%s%d: can not allocate dma handle", XGELL_IFNAME,
1076                        lldev->instance);
1077                    goto _exit_cleanup;
1078            }

1080            ret = ddi_dma_addr_bind_handle(dma_handle, NULL,
1081                (caddr_t)bp->b_rptr, mblen,
1082                DDI_DMA_WRITE | DDI_DMA_STREAMING, DDI_DMA_DONTWAIT, 0,
1083                &dma_cookie, &ncookies);

1085            switch (ret) {
1086            case DDI_DMA_MAPPED:
1087                    /* everything's fine */
1088                    break;

1090            case DDI_DMA_NORESOURCES:
1091                    xge_debug_ll(XGE_ERR,
1092                        "%s%d: can not bind dma address",
1093                        XGELL_IFNAME, lldev->instance);
1094                    ddi_dma_free_handle(&dma_handle);
1095                    goto _exit_cleanup;

1097            case DDI_DMA_NOMAPPING:
1098            case DDI_DMA_INUSE:
1099            case DDI_DMA_TOOBIG:
1100            default:
1101                    /* drop packet, don't retry */
1102                    xge_debug_ll(XGE_ERR,
1103                        "%s%d: can not map message buffer",
1104                        XGELL_IFNAME, lldev->instance);
1105                    ddi_dma_free_handle(&dma_handle);
1106                    goto _exit_cleanup;
1107            }

1109            if (ncookies + frag_cnt > hldev->config.fifo.max_frags) {
1110                    xge_debug_ll(XGE_ERR, "%s%d: too many fragments, "
1111                        "requested c:%d+f:%d", XGELL_IFNAME,
1112                        lldev->instance, ncookies, frag_cnt);
1113                    (void) ddi_dma_unbind_handle(dma_handle);
1114                    ddi_dma_free_handle(&dma_handle);
1115                    goto _exit_cleanup;
1116            }

1118            /* setup the descriptors for this data buffer */
1119            while (ncookies) {
1120                    xge_hal_fifo_dtr_buffer_set(ring->channelh, dtr,
1121                        frag_cnt++, dma_cookie.dmac_laddress,
1122                        dma_cookie.dmac_size);
1123                    if (--ncookies) {
1124                            ddi_dma_nextcookie(dma_handle, &dma_cookie);
1125                    }

1127            }

1129            txd_priv->dma_handles[handle_cnt++] = dma_handle;

1131            if (bp->b_cont &&
```



```
1132                        (frag_cnt + XGE_HAL_DEFAULT_FIFO_FRAGS_THRESHOLD >=
1133                        hldev->config.fifo.max_frags)) {
1134                            mblk_t *nmp;

1136                            xge_debug_ll(XGE_TRACE,
1137                                "too many FRAGs [%d], pull up them", frag_cnt);

1139                            if ((nmp = msgpullup(bp->b_cont, -1)) == NULL) {
1140                                    /* Drop packet, don't retry */
1141                                    xge_debug_ll(XGE_ERR,
1142                                        "%s%d: can not pullup message buffer",
1143                                        XGELL_IFNAME, lldev->instance);
1144                                    goto _exit_cleanup;
1145                            }
1146                            freemsg(bp->b_cont);
1147                            bp->b_cont = nmp;
1148                    }
1149            }

1151            /* finalize unfinished copies */
1152            if (used_copy) {
1153                    xge_hal_fifo_dtr_buffer_finalize(ring->channelh, dtr,
1154                        frag_cnt++);
1155            }

1157            txd_priv->handle_cnt = handle_cnt;

1159            /*
1160             * If LSO is required, just call xge_hal_fifo_dtr_mss_set(dtr, mss) to
1161             * do all necessary work.
1162             */
1163            mac_lso_get(mp, &mss, &lsoflags);

1165            if (lsoflags & HW_LSO) {
1166                    xge_assert((mss != 0) && (mss <= XGE_HAL_DEFAULT_MTU));
1167                    xge_hal_fifo_dtr_mss_set(dtr, mss);
1168            }

1170            mac_hcksum_get(mp, NULL, NULL, NULL, NULL, &hckflags);
1171            if (hckflags & HCK_IPV4_HDRCKSUM) {
1172                    xge_hal_fifo_dtr_cksum_set_bits(dtr,
1173                        XGE_HAL_TXD_TX_CKO_IPV4_EN);
1174            }
1175            if (hckflags & HCK_FULLCKSUM) {
1176                    xge_hal_fifo_dtr_cksum_set_bits(dtr, XGE_HAL_TXD_TX_CKO_TCP_EN |
1177                        XGE_HAL_TXD_TX_CKO_UDP_EN);
1178            }

1180            xge_hal_fifo_dtr_post(ring->channelh, dtr);

1182            /* Update per-ring tx statistics */
1183            atomic_inc_64(&ring->tx_pkts);
1183            atomic_add_64(&ring->tx_pkts, 1);
1184            atomic_add_64(&ring->tx_bytes, sent_bytes);

1186            return (NULL);

1188    _exit_cleanup:
1189            /*
1190             * Could not successfully transmit but have changed the message,
1191             * so just free it and return NULL
1192             */
1193            for (i = 0; i < handle_cnt; i++) {
1194                    (void) ddi_dma_unbind_handle(txd_priv->dma_handles[i]);
1195                    ddi_dma_free_handle(&txd_priv->dma_handles[i]);
1196                    txd_priv->dma_handles[i] = 0;
```

```
1197         }

1199         xge_hal_fifo_dtr_free(ring->channelh, dtr);

1201         freemsg(mp);
1202         return (NULL);

1204 _exit:
1205         ring->need_resched = B_TRUE;
1206         return (mp);
1207 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    3932 Mon Jul 28 07:44:44 2014
new/usr/src/uts/common/ipp/dlcosmk/dlcosmk.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/types.h>
#include <sys/stream.h>
#include <sys/dlpi.h>
#include <sys/strsun.h>
#include <netinet/in.h>
#include <netinet/ip6.h>
#include <inet/common.h>
#include <inet/ip.h>
#include <inet/ip6.h>
#include <inet/ip_if.h>
#include <ipp/dlcosmk/dlcosmk_impl.h>

/* Module to mark the 802.1d user priority field for a given packet */

/* Debug level */
int dlcosmk_debug = 0;

/*
 * Given a packet, this module marks the mblk with the appropriate b_band or
 * dl_max value so that the VLAN driver marks the outgoing frame with the
 * configured 802.1D user_priority value. For non-VLAN devices or for inbound
 * packets, this module does not do anything (i.e. the packet is processed by
 * the next action in the list, if present).
 * This module does not free any mblks or packets in case or errors.
 */

int
dlcosmk_process(mblk_t **mpp, dlcosmk_data_t *dlcosmk_data, uint32_t ill_index,
    ip_proc_t proc)
{
        ill_t *ill = NULL;
        mblk_t *mp;

        ASSERT((mpp != NULL) && (*mpp != NULL));
        mp = *mpp;
```



```
        /*
         * The action module will receive an M_DATA or an M_CTL followed
         * by an M_DATA. In the latter case skip the M_CTL.
         */
        if (mp->b_datap->db_type != M_DATA) {
                if ((mp->b_cont == NULL) ||
                    (mp->b_cont->b_datap->db_type != M_DATA)) {
                        atomic_inc_64(&dlcosmk_data->epackets);
                        atomic_add_64(&dlcosmk_data->epackets, 1);
                        dlcosmk0dbg(("dlcosmk_process: no data\n"));
                        return (EINVAL);
                }
        }

        /* Update global stats */
        atomic_inc_64(&dlcosmk_data->npackets);
        atomic_add_64(&dlcosmk_data->npackets, 1);

        /*
         * This should only be called for outgoing packets. For inbound, just
         * send it along.
         */
        if ((proc == IPP_LOCAL_IN) || (proc == IPP_FWD_IN)) {
                dlcosmk2dbg(("dlcosmk_process:cannot mark incoming packets\n"));
                atomic_inc_64(&dlcosmk_data->ipackets);
                atomic_add_64(&dlcosmk_data->ipackets, 1);
                return (0);
        }

        if ((ill_index == 0) ||
            ((ill = ill_lookup_on_ifindex_global_instance(ill_index,
            B_FALSE)) == NULL)) {
                dlcosmk2dbg(("dlcosmk_process:invalid ill index %u\n",
                    ill_index));
                atomic_inc_64(&dlcosmk_data->ipackets);
                atomic_add_64(&dlcosmk_data->ipackets, 1);
                return (0);
        }

        /*
         * Check if the interface supports CoS marking. If not send it to the
         * next action in the chain
         */
        if (!(ill->ill_flags & ILLF_COS_ENABLED)) {
                dlcosmk2dbg(("dlcosmk_process:ill %u does not support CoS\n",
                    ill_index));
                atomic_inc_64(&dlcosmk_data->ipackets);
                atomic_add_64(&dlcosmk_data->ipackets, 1);
                ill_refrele(ill);
                return (0);
        }
        ill_refrele(ill);


        /*
         * Mark the b_band for fastpath messages or dl_priority.dl_max for
         * DL_UNITDATA_REQ messages. For, others just pass it along.
         */
        switch (DB_TYPE(mp)) {
                case M_PROTO:
                case M_PCPROTO:
                        {       /* DL_UNITDATA */
                                dl_unitdata_req_t *dlur;
                                dlur = (dl_unitdata_req_t *)mp->b_rptr;

                                /* DL_UNITDATA message?? */
```

```
123                                 if (dlur->dl_primitive == DL_UNITDATA_REQ) {
124                                         dlur->dl_priority.dl_max =
125                                             dlcosmk_data->dl_max;
126                                 } else {
127                                         atomic_inc_64(&dlcosmk_data->ipackets);
127                                         atomic_add_64(&dlcosmk_data->ipackets,
128                                             1);
128                                 }
129                                 break;
130                         }
131                 case M_DATA:
132                         /* fastpath message */
133                         mp->b_band = dlcosmk_data->b_band;
134                         break;
135                 default:
136                         atomic_inc_64(&dlcosmk_data->ipackets);
137                         atomic_add_64(&dlcosmk_data->ipackets, 1);
137                         break;
138         }

140         return (0);
141 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
    5061 Mon Jul 28 07:44:44 2014
new/usr/src/uts/common/ipp/dscpmk/dscpmk.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2005 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/types.h>
#include <sys/atomic.h>
#include <sys/pattr.h>
#include <netinet/in.h>
#include <netinet/ip6.h>
#include <inet/common.h>
#include <inet/ip.h>
#include <inet/ip6.h>
#include <ipp/dscpmk/dscpmk_impl.h>

/* Module to mark the ToS/DS field for a given packet */

/* Debug level */
int dscpmk_debug = 0;

/*
 * Given a packet, this routine marks the ToS or DSCP for IPv4 and IPv6 resp.
 * using the configured dscp_map.
 * Note that this module does not change the ECN bits.
 */
int
dscpmk_process(mblk_t **mpp, dscpmk_data_t *dscpmk_data, ip_proc_t proc)
{
        ipha_t *ipha;
        ip6_t *ip6_hr;
        boolean_t is_v4;
        uint8_t dscp, new_dscp;
        mblk_t *mp;

        ASSERT((mpp != NULL) && (*mpp != NULL));
        mp = *mpp;

        /*
```



```
         * The action module will receive an M_DATA or an M_CTL followed
         * by an M_DATA. In the latter case skip the M_CTL.
         */
        if (mp->b_datap->db_type != M_DATA) {
                if ((mp->b_cont != NULL) &&
                    (mp->b_cont->b_datap->db_type == M_DATA)) {
                        mp = mp->b_cont;
                } else {
                        dscpmk0dbg(("dscpmk_process: no data\n"));
                        atomic_inc_64(&dscpmk_data->epackets);
                        atomic_add_64(&dscpmk_data->epackets, 1);
                        return (EINVAL);
                }
        }

        /* Pull-up needed? */
        if ((mp->b_wptr - mp->b_rptr) < IP_SIMPLE_HDR_LENGTH) {
                if (!pullupmsg(mp, IP_SIMPLE_HDR_LENGTH)) {
                        dscpmk0dbg(("dscpmk_process: pullup failed\n"));
                        atomic_inc_64(&dscpmk_data->epackets);
                        atomic_add_64(&dscpmk_data->epackets, 1);
                        return (EINVAL);
                }
        }
        ipha = (ipha_t *)mp->b_rptr;

        /* Update global stats */
        atomic_inc_64(&dscpmk_data->npackets);
        atomic_add_64(&dscpmk_data->npackets, 1);

        /*
         * This should only be called for outgoing packets. For inbound packets
         * proceed with the next action.
         */
        if ((proc == IPP_LOCAL_IN) || (proc == IPP_FWD_IN)) {
                dscpmk2dbg(("dscpmk_process: cannot mark incoming packets\n"));
                atomic_inc_64(&dscpmk_data->ipackets);
                atomic_add_64(&dscpmk_data->ipackets, 1);
                return (0);
        }

        /* Figure out the ToS or the Traffic Class from the message */
        if (IPH_HDR_VERSION(ipha) == IPV4_VERSION) {
                dscp = ipha->ipha_type_of_service;
                is_v4 = B_TRUE;
        } else {
                ip6_hdr = (ip6_t *)mp->b_rptr;
                dscp = __IPV6_TCLASS_FROM_FLOW(ip6_hdr->ip6_vcf);
                is_v4 = B_FALSE;
        }

        /*
         * Select the new dscp from the dscp_map after ignoring the
         * ECN/CU from dscp (hence dscp >> 2). new_dscp will be the
         * 6-bit DSCP value.
         */
        new_dscp = dscpmk_data->dscp_map[dscp >> 2];

        /* Update stats for this new_dscp */
        atomic_inc_64(&dscpmk_data->dscp_stats[new_dscp].npackets);
        atomic_add_64(&dscpmk_data->dscp_stats[new_dscp].npackets, 1);

        /*
         * if new_dscp is same as the original, update stats and
         * return.
         */
```

```
121         if (new_dscp == (dscp >> 2)) {
122                 atomic_inc_64(&dscpmk_data->unchanged);
124                 atomic_add_64(&dscpmk_data->unchanged, 1);
123                 return (0);
124         }

126         /* Get back the ECN/CU value from the original dscp */
127         new_dscp = (new_dscp << 2) | (dscp & 0x3);

129         atomic_inc_64(&dscpmk_data->changed);
131         atomic_add_64(&dscpmk_data->changed, 1);
130         /*
131          * IPv4 : ToS structure -- RFC 791
132          *
133          *       0   1   2   3   4   5   6   7
134          *     +---+---+---+---+---+---+---+---+
135          *     | IP Precd  | D | T | R | 0 | 0 |
136          *     |           |   |   |   |   |   |
137          *     +---+---+---+---+---+---+---+---+
138          *
139          * For Backward Compatability the diff serv DSCP will be mapped
140          * to the 3-bits Precedence field. DTR is not supported. Thus,
141          * the following Class Seletor CodePoints are reserved from this
142          * purpose : xxx000; where x is 0 or 1 (note the last 2 bits are
143          * 00) -- see RFC 2474.
144          */

146         if (is_v4) {
147                 ipha->ipha_type_of_service = new_dscp;
148                 /*
149                  * If the hardware supports checksumming, we don't need
150                  * to do anything.
151                  */
152                 if (!(mp->b_datap->db_struioun.cksum.flags &
153                     HCK_IPV4_HDRCKSUM)) {
154                         ipha->ipha_hdr_checksum = 0;
155                         ipha->ipha_hdr_checksum = ip_csum_hdr(ipha);
156                 }
157         } else {

159         /*
160          * IPv6 : DSCP field structure is as given -- RFC 2474
161          *
162          *       0   1   2   3   4   5   6   7
163          *     +---+---+---+---+---+---+---+---+
164          *     |       DSCP            | CU    |
165          *     |                       |       |
166          *     +---+---+---+---+---+---+---+---+
167          *
168          * CU -- Currently Unused
169          *
170          * the 32 bit vcf consists of version (4 bits), Traffic class (8 bits)
171          * and flow id (20 bits). Need to take care of Big/Little-Endianess.
172          */
173 #ifdef _BIG_ENDIAN
174                 ip6_hdr->ip6_vcf = (ip6_hdr->ip6_vcf & TCLASS_MASK) |
175                     (new_dscp << 20);
176 #else
177                 ip6_hdr->ip6_vcf = (ip6_hdr->ip6_vcf & TCLASS_MASK) |
178                     ((new_dscp >> 4) | ((new_dscp << 12) & 0xF000));
179 #endif
180         }

182         return (0);
183 }
```

unchanged_portion_omitted_

```
**********************************************************
   25931 Mon Jul 28 07:44:44 2014
new/usr/src/uts/common/ipp/flowacct/flowacct.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 426 /*
 427  * Add the flow to the table, if not already present. If the flow is
 428  * present in the table, add the item. Also, update the flow stats.
 429  * Additionally, re-adjust the timout list as well.
 430  */
 431 static int
 432 flowacct_update_flows_tbl(header_t *header, flowacct_data_t *flowacct_data)
 433 {
 434         int index;
 435         list_head_t *fhead;
 436         list_head_t *thead;
 437         list_head_t *ihead;
 438         boolean_t added_flow = B_FALSE;
 439         timespec_t  now;
 440         flow_item_t *item;
 441         flow_t *flow;

 443         index = FLOWACCT_FLOW_HASH(header);
 444         fhead = &flowacct_data->flows_tbl[index];

 446         /* The timeout list */
 447         thead = &flowacct_data->flows_tbl[FLOW_TBL_COUNT];

 449         mutex_enter(&fhead->lock);
 450         flow = flowacct_flow_present(header, index, flowacct_data);
 451         if (flow == NULL) {
 452                 flow = (flow_t *)kmem_zalloc(FLOWACCT_FLOW_SZ, KM_NOSLEEP);
 453                 if (flow == NULL) {
 454                         mutex_exit(&fhead->lock);
 455                         flowacct0dbg(("flowacct_update_flows_tbl: mem alloc "\
 456                             "error"));
 457                         return (-1);
 458                 }
 459                 flow->hdr = flowacct_add_obj(fhead, fhead->tail, (void *)flow);
 460                 if (flow->hdr == NULL) {
 461                         mutex_exit(&fhead->lock);
 462                         kmem_free(flow, FLOWACCT_FLOW_SZ);
 463                         flowacct0dbg(("flowacct_update_flows_tbl: mem alloc "\
 464                             "error"));
 465                         return (-1);
 466                 }

 468                 flow->type = FLOWACCT_FLOW;
 469                 flow->isv4 = header->isv4;
 470                 bcopy(header->saddr.s6_addr32, flow->saddr.s6_addr32,
 471                     sizeof (header->saddr.s6_addr32));
 472                 bcopy(header->daddr.s6_addr32, flow->daddr.s6_addr32,
 473                     sizeof (header->daddr.s6_addr32));
 474                 flow->proto = header->proto;
 475                 flow->sport = header->sport;
 476                 flow->dport = header->dport;
 477                 flow->back_ptr = fhead;
 478                 added_flow = B_TRUE;
 479         } else {
 480                 /*
 481                  * We need to make sure that this 'flow' is not deleted
 482                  * either by a scheduled timeout or an explict call
 483                  * to flowacct_timer() below.
 484                  */
```



```
 485                 flow->inuse = B_TRUE;
 486         }

 488         ihead = &flow->items;
 489         item = flowacct_item_present(flow, header->dsfield, header->projid,
 490             header->uid);
 491         if (item == NULL) {
 492                 boolean_t just_once = B_TRUE;
 493                 /*
 494                  * For all practical purposes, we limit the no. of entries in
 495                  * the flow table - i.e. the max_limt that a user specifies is
 496                  * the maximum no. of flow items in the table.
 497                  */
 498         try_again:
 499                 atomic_inc_32(&flowacct_data->nflows);
 499                 atomic_add_32(&flowacct_data->nflows, 1);
 500                 if (flowacct_data->nflows > flowacct_data->max_limit) {
 501                         atomic_dec_32(&flowacct_data->nflows);
 501                         atomic_add_32(&flowacct_data->nflows, -1);

 503                         /* Try timing out once */
 504                         if (just_once) {
 505                                 /*
 506                                  * Need to release the lock, as this entry
 507                                  * could contain a flow that can be timed
 508                                  * out.
 509                                  */
 510                                 mutex_exit(&fhead->lock);
 511                                 flowacct_timer(FLOWACCT_JUST_ONE,
 512                                     flowacct_data);
 513                                 mutex_enter(&fhead->lock);
 514                                 /* Lets check again */
 515                                 just_once = B_FALSE;
 516                                 goto try_again;
 517                         } else {
 518                                 flow->inuse = B_FALSE;
 519                                 /* Need to remove the flow, if one was added */
 520                                 if (added_flow) {
 521                                         flowacct_del_obj(fhead, flow->hdr,
 522                                             FLOWACCT_DEL_OBJ);
 523                                 }
 524                                 mutex_exit(&fhead->lock);
 525                                 flowacct1dbg(("flowacct_update_flows_tbl: "\
 526                                     "maximum active flows exceeded\n"));
 527                                 return (-1);
 528                         }
 529                 }
 530                 item = (flow_item_t *)kmem_zalloc(FLOWACCT_ITEM_SZ, KM_NOSLEEP);
 531                 if (item == NULL) {
 532                         flow->inuse = B_FALSE;
 533                         /* Need to remove the flow, if one was added */
 534                         if (added_flow) {
 535                                 flowacct_del_obj(fhead, flow->hdr,
 536                                     FLOWACCT_DEL_OBJ);
 537                         }
 538                         mutex_exit(&fhead->lock);
 539                         atomic_dec_32(&flowacct_data->nflows);
 539                         atomic_add_32(&flowacct_data->nflows, -1);
 540                         flowacct0dbg(("flowacct_update_flows_tbl: mem alloc "\
 541                             "error"));
 542                         return (-1);
 543                 }
 544                 item->hdr = flowacct_add_obj(ihead, ihead->tail, (void *)item);
 545                 if (item->hdr == NULL) {
 546                         flow->inuse = B_FALSE;
 547                         /* Need to remove the flow, if one was added */
```

```
548                         if (added_flow) {
549                                 flowacct_del_obj(fhead, flow->hdr,
550                                     FLOWACCT_DEL_OBJ);
551                         }
552                         mutex_exit(&fhead->lock);
553                         atomic_dec_32(&flowacct_data->nflows);
553                         atomic_add_32(&flowacct_data->nflows, -1);
554                         kmem_free(item, FLOWACCT_ITEM_SZ);
555                         flowacct0dbg(("flowacct_update_flows_tbl: mem alloc "\
556                             "error\n"));
557                         return (-1);
558                 }
559                 /* If a flow was added, add it too */
560                 if (added_flow) {
561                         atomic_add_64(&flowacct_data->usedmem,
562                             FLOWACCT_FLOW_RECORD_SZ);
563                 }
564                 atomic_add_64(&flowacct_data->usedmem, FLOWACCT_ITEM_RECORD_SZ);

566                 item->type = FLOWACCT_ITEM;
567                 item->dsfield = header->dsfield;
568                 item->projid = header->projid;
569                 item->uid = header->uid;
570                 item->npackets = 1;
571                 item->nbytes = header->pktlen;
572                 item->creation_time = item->hdr->last_seen;
573         } else {
574                 item->npackets++;
575                 item->nbytes += header->pktlen;
576         }
577         gethrestime(&now);
578         flow->hdr->last_seen = item->hdr->last_seen = now;
579         mutex_exit(&fhead->lock);

581         /*
582          * Re-adjust the timeout list. The timer takes the thead lock
583          * follwed by fhead lock(s), so we release fhead, take thead
584          * and re-take fhead.
585          */
586         mutex_enter(&thead->lock);
587         mutex_enter(&fhead->lock);
588         /* If the flow was added, append it to the tail of the timeout list */
589         if (added_flow) {
590                 if (thead->head == NULL) {
591                         thead->head = flow->hdr;
592                         thead->tail = flow->hdr;
593                 } else {
594                         thead->tail->timeout_next = flow->hdr;
595                         flow->hdr->timeout_prev = thead->tail;
596                         thead->tail = flow->hdr;
597                 }
598         /*
599          * Else, move this flow to the tail of the timeout list, if it is not
600          * already.
601          * flow->hdr in the timeout list :-
602          * timeout_next = NULL, timeout_prev != NULL, at the tail end.
603          * timeout_next != NULL, timeout_prev = NULL, at the head.
604          * timeout_next != NULL, timeout_prev != NULL, in the middle.
605          * timeout_next = NULL, timeout_prev = NULL, not in the timeout list,
606          * ignore such flow.
607          */
608         } else if ((flow->hdr->timeout_next != NULL) ||
609             (flow->hdr->timeout_prev != NULL)) {
610                 if (flow->hdr != thead->tail) {
611                         if (flow->hdr == thead->head) {
612                                 thead->head->timeout_next->timeout_prev = NULL;
```



```
613                                 thead->head = thead->head->timeout_next;
614                                 flow->hdr->timeout_next = NULL;
615                                 thead->tail->timeout_next = flow->hdr;
616                                 flow->hdr->timeout_prev = thead->tail;
617                                 thead->tail = flow->hdr;
618                         } else {
619                                 flow->hdr->timeout_prev->timeout_next =
620                                     flow->hdr->timeout_next;
621                                 flow->hdr->timeout_next->timeout_prev =
622                                     flow->hdr->timeout_prev;
623                                 flow->hdr->timeout_next = NULL;
624                                 thead->tail->timeout_next = flow->hdr;
625                                 flow->hdr->timeout_prev = thead->tail;
626                                 thead->tail = flow->hdr;
627                         }
628                 }
629         }
630         /*
631          * Unset this variable, now it is fine even if this
632          * flow gets deleted (i.e. after timing out its
633          * flow items) since we are done using it.
634          */
635         flow->inuse = B_FALSE;
636         mutex_exit(&fhead->lock);
637         mutex_exit(&thead->lock);
638         atomic_add_64(&flowacct_data->tbytes, header->pktlen);
639         return (0);
640 }
_____unchanged_portion_omitted_

871 /*
872  * Get the IP header contents from the packet, update the flow table with
873  * this item and return.
874  */
875 int
876 flowacct_process(mblk_t **mpp, flowacct_data_t *flowacct_data)
877 {
878         header_t *header;
879         mblk_t *mp = *mpp;

881         ASSERT(mp != NULL);

883         /* If we don't find an M_DATA, return error */
884         if (mp->b_datap->db_type != M_DATA) {
885                 if ((mp->b_cont != NULL) &&
886                     (mp->b_cont->b_datap->db_type == M_DATA)) {
887                         mp = mp->b_cont;
888                 } else {
889                         flowacct0dbg(("flowacct_process: no data\n"));
890                         atomic_inc_64(&flowacct_data->epackets);
890                         atomic_add_64(&flowacct_data->epackets, 1);
891                         return (EINVAL);
892                 }
893         }

895         header = kmem_zalloc(FLOWACCT_HEADER_SZ, KM_NOSLEEP);
896         if (header == NULL) {
897                 flowacct0dbg(("flowacct_process: error allocing mem"));
898                 atomic_inc_64(&flowacct_data->epackets);
898                 atomic_add_64(&flowacct_data->epackets, 1);
899                 return (ENOMEM);
900         }

902         /* Get all the required information into header. */
903         if (flowacct_extract_header(mp, header) != 0) {
904                 kmem_free(header, FLOWACCT_HEADER_SZ);
```

```
905                 atomic_inc_64(&flowacct_data->epackets);
905                 atomic_add_64(&flowacct_data->epackets, 1);
906                 return (EINVAL);
907         }

909         /* Updated the flow table with this entry */
910         if (flowacct_update_flows_tbl(header, flowacct_data) != 0) {
911                 kmem_free(header, FLOWACCT_HEADER_SZ);
912                 atomic_inc_64(&flowacct_data->epackets);
912                 atomic_add_64(&flowacct_data->epackets, 1);
913                 return (ENOMEM);
914         }

916         /* Update global stats */
917         atomic_inc_64(&flowacct_data->npackets);
917         atomic_add_64(&flowacct_data->npackets, 1);
918         atomic_add_64(&flowacct_data->nbytes, header->pktlen);

920         kmem_free(header, FLOWACCT_HEADER_SZ);
921         if (flowacct_data->flow_tid == 0) {
922                 flowacct_data->flow_tid = timeout(flowacct_timeout_flows,
923                     flowacct_data, drv_usectohz(flowacct_data->timer));
924         }
925         return (0);
926 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   13488 Mon Jul 28 07:44:44 2014
new/usr/src/uts/common/ipp/ipgpc/classifierddi.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 372 /*
 373  * ipgpc_invoke_action(aid, packet)
 374  *
 375  * packet processing function for ipgpc
 376  *
 377  * given packet the selector information is parsed and the classify
 378  * function is called with those selectors.  The classify function will
 379  * return either a class or NULL, which represents a memory error and
 380  * ENOMEM is returned.  If the class returned is not NULL, the class and next
 381  * action, associated with that class, are added to packet
 382  */
 383 /* ARGSUSED */
 384 static int
 385 ipgpc_invoke_action(ipp_action_id_t aid, ipp_packet_t *packet)
 386 {
 387         ipgpc_class_t *out_class;
 388         hrtime_t start, end;
 389         mblk_t *mp = NULL;
 390         ip_priv_t *priv = NULL;
 391         ill_t *ill = NULL;
 392         ipha_t *ipha;
 393         ip_proc_t callout_pos;
 394         int af;
 395         int rc;
 396         ipgpc_packet_t pkt;
 397         uint_t ill_idx;

 399         /* extract packet data */
 400         mp = ipp_packet_get_data(packet);
 401         ASSERT(mp != NULL);

 403         priv = (ip_priv_t *)ipp_packet_get_private(packet);
 404         ASSERT(priv != NULL);

 406         callout_pos = priv->proc;
 407         ill_idx = priv->ill_index;

 409         /* If we don't get an M_DATA, then return an error */
 410         if (mp->b_datap->db_type != M_DATA) {
 411                 if ((mp->b_cont != NULL) &&
 412                     (mp->b_cont->b_datap->db_type == M_DATA)) {
 413                         mp = mp->b_cont; /* jump over the M_CTL into M_DATA */
 414                 } else {
 415                         ipgpc0dbg(("ipgpc_invoke_action: no data\n"));
 416                         atomic_inc_64(&ipgpc_epackets);
 416                         atomic_add_64(&ipgpc_epackets, 1);
 417                         return (EINVAL);
 418                 }
 419         }

 421         /*
 422          * Translate the callout_pos into the direction the packet is traveling
 423          */
 424         if (callout_pos != IPP_LOCAL_IN) {
 425                 if (callout_pos & IPP_LOCAL_OUT) {
 426                         callout_pos = IPP_LOCAL_OUT;
 427                 } else if (callout_pos & IPP_FWD_IN) {
 428                         callout_pos = IPP_FWD_IN;
 429                 } else {        /* IPP_FWD_OUT */
```



```
 430                         callout_pos = IPP_FWD_OUT;
 431                 }
 432         }

 434         /* parse the packet from the message block */
 435         ipha = (ipha_t *)mp->b_rptr;
 436         /* Determine IP Header Version */
 437         if (IPH_HDR_VERSION(ipha) == IPV4_VERSION) {
 438                 parse_packet(&pkt, mp);
 439                 af = AF_INET;
 440         } else {
 441                 parse_packet6(&pkt, mp);
 442                 af = AF_INET6;
 443         }

 445         pkt.direction = callout_pos; /* set packet direction */

 447         /* The ill_index could be 0 when called from forwarding (read) path */
 448         if (ill_idx > 0)
 449                 ill = ill_lookup_on_ifindex_global_instance(ill_idx, B_FALSE);

 451         if (ill != NULL) {
 452                 /*
 453                  * Since all IPP actions in an IPMP group are performed
 454                  * relative to the IPMP group interface, if this is an
 455                  * underlying interface in an IPMP group, use the IPMP
 456                  * group interface's index.
 457                  */
 458                 if (IS_UNDER_IPMP(ill))
 459                         pkt.if_index = ipmp_ill_get_ipmp_ifindex(ill);
 460                 else
 461                         pkt.if_index = ill->ill_phyint->phyint_ifindex;
 462                 /* Got the field from the ILL, go ahead and refrele */
 463                 ill_refrele(ill);
 464         } else {
 465                 /* unknown if_index */
 466                 pkt.if_index = IPGPC_UNSPECIFIED;
 467         }

 469         if (ipgpc_debug > 5) {
 470                 /* print pkt under high debug level */
 471 #ifdef  IPGPC_DEBUG
 472                 print_packet(af, &pkt);
 473 #endif
 474         }
 475         if (ipgpc_debug > 3) {
 476                 start = gethrtime(); /* start timer */
 477         }

 479         /* classify this packet */
 480         out_class = ipgpc_classify(af, &pkt);

 482         if (ipgpc_debug > 3) {
 483                 end = gethrtime(); /* stop timer */
 484         }

 486         /* ipgpc_classify will only return NULL if a memory error occured */
 487         if (out_class == NULL) {
 488                 atomic_inc_64(&ipgpc_epackets);
 488                 atomic_add_64(&ipgpc_epackets, 1);
 489                 return (ENOMEM);
 490         }

 492         ipgpc1dbg(("ipgpc_invoke_action: class = %s", out_class->class_name));
 493         /* print time to classify(..) */
 494         ipgpc2dbg(("ipgpc_invoke_action: time = %lld nsec\n", (end - start)));
```

```
 496         if ((rc = ipp_packet_add_class(packet, out_class->class_name,
 497             out_class->next_action)) != 0) {
 498                 atomic_inc_64(&ipgpc_epackets);
 498                 atomic_add_64(&ipgpc_epackets, 1);
 499                 ipgpc0dbg(("ipgpc_invoke_action: ipp_packet_add_class " \
 500                     "failed with error %d", rc));
 501                 return (rc);
 502         }
 503         return (ipp_packet_next(packet, IPP_ACTION_CONT));
 504 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   76710 Mon Jul 28 07:44:44 2014
new/usr/src/uts/common/ipp/ipgpc/filters.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 275 static void
 276 element_node_ref(element_node_t *element)
 277 {
 278         atomic_inc_32(&element->element_refcnt);
 278         atomic_add_32(&element->element_refcnt, 1);
 279         ASSERT(element->element_refcnt > 1);
 280 }

 282 static void
 283 element_node_unref(element_node_t *element)
 284 {
 285         ASSERT(element->element_refcnt > 0);
 286         if (atomic_dec_32_nv(&element->element_refcnt) == 0) {
 286         if (atomic_add_32_nv(&element->element_refcnt, -1) == 0) {
 287                 kmem_cache_free(element_node_cache, element);
 288         }
 289 }
______unchanged_portion_omitted_

1050 /*
1051  * ipgpc_addfilter(filter, class_name, flags)
1052  *
1053  * add the specified filter and associate it with the specified class
1054  * name
1055  * - add filter id to filter list
1056  * - add filter keys to selector structures
1057  * - ENOENT is returned if class does not exist
1058  * - EEXIST is returned if add failed because filter name exists
1059  * - ENOMEM is returned if no memory is available to add a new filter
1060  * - EINVAL if filter.filter_type is invalid
1061  * - 0 is returned on success
1062  * flags is unused currently
1063  */
1064 /* ARGSUSED1 */
1065 int
1066 ipgpc_addfilter(ipgpc_filter_t *filter, char *class_name, ipp_flags_t flags)
1067 {
1068         unsigned filter_id;
1069         int err = 0;
1070         fid_t *fid;
1071         unsigned class_id;

1073         err = class_name2id(&class_id, class_name, ipgpc_num_cls);
1074         if (err != EEXIST) {
1075                 ipgpc0dbg(("ipgpc_addfilter: class lookup error %d", err));
1076                 return (err);
1077         }
1078         mutex_enter(&ipgpc_fid_list_lock);
1079         /* make sure filter does not already exist */
1080         if ((err = filter_name2id(&filter_id, filter->filter_name,
1081             filter->filter_instance, ipgpc_num_fltrs + 1)) == EEXIST) {
1082                 ipgpc0dbg(("ipgpc_addfilter: filter name %s already exists",
1083                     filter->filter_name));
1084                 mutex_exit(&ipgpc_fid_list_lock);
1085                 return (err);
1086         } else if (err == ENOSPC) {
1087                 ipgpc0dbg(("ipgpc_addfilter: can not add filter %s, " \
1088                     "ipgpc_max_num_filteres has been reached",
1089                     filter->filter_name));
```



```
1090                 mutex_exit(&ipgpc_fid_list_lock);
1091                 return (err);
1092         }
1093         insertfid(filter_id, filter, class_id);

1095         fid = &ipgpc_fid_list[filter_id];
1096         /* add filter id to selector structures */
1097         switch (fid->filter.filter_type) {
1098         case IPGPC_GENERIC_FLTR:
1099                 /* add filter id to all selectors */
1100                 common_addfilter(fid, filter_id);
1101                 v4_addfilter(fid, filter_id);
1102                 v6_addfilter(fid, filter_id);
1103                 break;
1104         case IPGPC_V4_FLTR:
1105                 /* add filter to common and V4 selectors */
1106                 common_addfilter(fid, filter_id);
1107                 v4_addfilter(fid, filter_id);
1108                 break;
1109         case IPGPC_V6_FLTR:
1110                 /* add filter to common and V6 selectors */
1111                 common_addfilter(fid, filter_id);
1112                 v6_addfilter(fid, filter_id);
1113                 break;
1114         default:
1115                 ipgpc0dbg(("ipgpc_addfilter(): invalid filter type %d",
1116                     fid->filter.filter_type));
1117                 mutex_exit(&ipgpc_fid_list_lock);
1118                 return (EINVAL);
1119         }
1120         /* check to see if this is a catch all filter, which we reject */
1121         if (fid->insert_map == 0) {
1122                 ipgpc0dbg(("ipgpc_addfilter(): filter %s rejected because " \
1123                     "catch all filters are not supported\n",
1124                     filter->filter_name));
1125                 /* cleanup what we allocated */
1126                 /* remove filter from filter list */
1127                 ipgpc_fid_list[filter_id].info = -1;
1128                 ipgpc_fid_list[filter_id].filter.filter_name[0] = '\0';
1129                 reset_dontcare_stats(); /* need to fixup stats */
1130                 mutex_exit(&ipgpc_fid_list_lock);
1131                 return (EINVAL);
1132         } else {                /* associate filter with class */
1133                 mutex_enter(&ipgpc_cid_list_lock);
1134                 (void) ipgpc_list_insert(&ipgpc_cid_list[class_id].filter_list,
1135                     filter_id);
1136                 mutex_exit(&ipgpc_cid_list_lock);
1137         }
1138         mutex_exit(&ipgpc_fid_list_lock);
1139         atomic_inc_ulong(&ipgpc_num_fltrs);
1139         atomic_add_long(&ipgpc_num_fltrs, 1);
1140         ipgpc3dbg(("ipgpc_addfilter: adding filter %s", filter->filter_name));
1141         return (0);
1142 }

1144 /*
1145  * reset_dontcare_stats()
1146  *
1147  * when an insertion fails because zero selectors are specified in a filter
1148  * the number of dontcare's recorded for each selector structure needs to be
1149  * decremented
1150  */
1151 static void
1152 reset_dontcare_stats(void)
1153 {
1154         int i;
```

```
1156         for (i = 0; i < NUM_TRIES; ++i) {
1157                 atomic_dec_32(&ipgpc_trie_list[i].stats.num_dontcare);
1157                 atomic_add_32(&ipgpc_trie_list[i].stats.num_dontcare, -1);
1158         }
1159         for (i = 0; i < NUM_TABLES; ++i) {
1160                 atomic_dec_32(&ipgpc_table_list[i].stats.num_dontcare);
1160                 atomic_add_32(&ipgpc_table_list[i].stats.num_dontcare, -1);
1161         }
1162         atomic_dec_32(&ipgpc_ds_table_id.stats.num_dontcare);
1162         atomic_add_32(&ipgpc_ds_table_id.stats.num_dontcare, -1);
1163 }
______unchanged_portion_omitted_

1319 /*
1320  * insertcid(in_class, out_class_id)
1321  *
1322  * creates a class id (cid) structure for in_class, if in_class name
1323  * does not exist already.  id is associated with in_class. the internal
1324  * id of the cid associated with in_class is returned in out_class_id
1325  * - ENOENT is returned if in_class->class_name does not already exist
1326  * - EEXIST is returned if in_class->class_name does already exist
1327  * - ENOSPC is returned if by adding this class, the ipgpc_max_num_classes
1328  *   will be exceeded.
1329  */
1330 static int
1331 insertcid(ipgpc_class_t *in_class, int *out_class_id)
1332 {
1333         int err, rc;
1334         unsigned class_id;

1336         mutex_enter(&ipgpc_cid_list_lock);
1337         /* see if entry already exists for class */
1338         if ((err = class_name2id(&class_id, in_class->class_name,
1339             ipgpc_num_cls + 1)) == ENOENT) {
1340                 /* create new filter list for new class */
1341                 ipgpc_cid_list[class_id].info = 1;
1342                 ipgpc_cid_list[class_id].aclass = *in_class;
1343                 if (in_class->gather_stats == B_TRUE) {
1344                         /* init kstat entry */
1345                         if ((rc = class_statinit(in_class, class_id)) != 0) {
1346                                 ipgpc_cid_list[class_id].info = -1;
1347                                 ipgpc0dbg(("insertcid: "
1348                                     "class_statinit failed with error %d", rc));
1349                                 mutex_exit(&ipgpc_cid_list_lock);
1350                                 return (rc);
1351                         }
1352                 } else {
1353                         ipgpc_cid_list[class_id].cl_stats = NULL;
1354                 }
1355                 ipgpc3dbg(("insertcid: adding class %s",
1356                     in_class->class_name));
1357                 bcopy(in_class->class_name,
1358                     ipgpc_cid_list[class_id].aclass.class_name, MAXNAMELEN);
1359                 ipgpc_cid_list[class_id].filter_list = NULL;
1360                 atomic_inc_ulong(&ipgpc_num_cls);
1360                 atomic_add_long(&ipgpc_num_cls, 1);
1361         } else {
1362                 ipgpc0dbg(("insertcid: class name lookup error %d", err));
1363                 mutex_exit(&ipgpc_cid_list_lock);
1364                 return (err);
1365         }
1366         mutex_exit(&ipgpc_cid_list_lock);
1367         *out_class_id = class_id;
1368         return (err);
1369 }
______unchanged_portion_omitted_
```



```
1447 /*
1448  * ipgpc_removefilter(filter_name, filter_instance, flags)
1449  *
1450  * remove the filter associated with the specified name and instance
1451  * - remove filter keys from all search tries
1452  * - remove from filter id list
1453  * - ENOENT is returned if filter name does not exist
1454  * - returns 0 on success
1455  */
1456 /* ARGSUSED */
1457 int
1458 ipgpc_removefilter(char *filter_name, int32_t filter_instance,
1459     ipp_flags_t flags)
1460 {
1461         unsigned filter_id;
1462         fid_t *fid;
1463         int rc;

1465         /* check to see if any filters are loaded */
1466         if (ipgpc_num_fltrs == 0) {
1467                 return (ENOENT);
1468         }

1470         mutex_enter(&ipgpc_fid_list_lock);
1471         /* lookup filter name, only existing filters can be removed */
1472         if ((rc = filter_name2id(&filter_id, filter_name, filter_instance,
1473             ipgpc_num_fltrs)) != EEXIST) {
1474                 mutex_exit(&ipgpc_fid_list_lock);
1475                 return (rc);
1476         }
1477         fid = &ipgpc_fid_list[filter_id];
1478         switch (fid->filter.filter_type) {
1479         case IPGPC_GENERIC_FLTR:
1480                 common_removefilter(filter_id, fid);
1481                 v4_removefilter(filter_id, fid);
1482                 v6_removefilter(filter_id, fid);
1483                 break;
1484         case IPGPC_V4_FLTR:
1485                 common_removefilter(filter_id, fid);
1486                 v4_removefilter(filter_id, fid);
1487                 break;
1488         case IPGPC_V6_FLTR:
1489                 common_removefilter(filter_id, fid);
1490                 v6_removefilter(filter_id, fid);
1491                 break;
1492         default:
1493                 ipgpc0dbg(("ipgpc_removefilter(): invalid filter type %d",
1494                     fid->filter.filter_type));
1495                 mutex_exit(&ipgpc_fid_list_lock);
1496                 return (EINVAL);
1497         }
1498         /* remove filter from filter list */
1499         ipgpc_fid_list[filter_id].info = -1;
1500         ipgpc_fid_list[filter_id].insert_map = 0;
1501         ipgpc_fid_list[filter_id].filter.filter_name[0] = '\0';
1502         ipgpc_filter_destructor(&ipgpc_fid_list[filter_id].filter);
1503         mutex_exit(&ipgpc_fid_list_lock);
1504         /* remove filter id from class' list of filters */
1505         remove_from_cid_filter_list(ipgpc_fid_list[filter_id].class_id,
1506             filter_id);
1507         atomic_dec_ulong(&ipgpc_num_fltrs);
1507         atomic_add_long(&ipgpc_num_fltrs, -1);
1508         return (0);
1509 }
```

```
1511 /*
1512  * removecid(in_class_id)
1513  *
1514  * removes the cid entry from the cid list and frees allocated structures
1515  */
1516 static void
1517 removecid(int in_class_id)
1518 {
1519         ipgpc_cid_list[in_class_id].info = -1;
1520         ipgpc_cid_list[in_class_id].aclass.class_name[0] = '\0';
1521         ipgpc_cid_list[in_class_id].aclass.next_action = -1;
1522         /* delete kstat entry */
1523         if (ipgpc_cid_list[in_class_id].cl_stats != NULL) {
1524                 ipp_stat_destroy(ipgpc_cid_list[in_class_id].cl_stats);
1525                 ipgpc_cid_list[in_class_id].cl_stats = NULL;
1526         }
1527         /* decrement total number of classes loaded */
1528         atomic_dec_ulong(&ipgpc_num_cls);
1528         atomic_add_long(&ipgpc_num_cls, -1);
1529 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   67402 Mon Jul 28 07:44:45 2014
new/usr/src/uts/common/ipp/ippconf.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/types.h>
#include <sys/param.h>
#include <sys/modctl.h>
#include <sys/sysmacros.h>
#include <sys/kmem.h>
#include <sys/cmn_err.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/spl.h>
#include <sys/time.h>
#include <sys/varargs.h>
#include <ipp/ipp.h>
#include <ipp/ipp_impl.h>
#include <ipp/ipgpc/ipgpc.h>

/*
 * Debug switch.
 */

#if     defined(DEBUG)
#define IPP_DBG
#endif

/*
 * Globals
 */

/*
 * ipp_action_count is not static because it is imported by inet/ipp_common.h
 */
uint32_t                ipp_action_count = 0;

static kmem_cache_t     *ipp_mod_cache = NULL;
static uint32_t         ipp_mod_count = 0;
```



```
static uint32_t         ipp_max_mod = IPP_NMOD;
static ipp_mod_t        **ipp_mod_byid;
static krwlock_t        ipp_mod_byid_lock[1];

static ipp_mod_id_t     ipp_next_mid = IPP_MOD_RESERVED + 1;
static ipp_mod_id_t     ipp_mid_limit;

static ipp_ref_t        *ipp_mod_byname[IPP_NBUCKET];
static krwlock_t        ipp_mod_byname_lock[1];

static kmem_cache_t     *ipp_action_cache = NULL;
static uint32_t         ipp_max_action = IPP_NACTION;
static ipp_action_t     **ipp_action_byid;
static krwlock_t        ipp_action_byid_lock[1];

static ipp_action_id_t  ipp_next_aid = IPP_ACTION_RESERVED + 1;
static ipp_action_id_t  ipp_aid_limit;

static ipp_ref_t        *ipp_action_byname[IPP_NBUCKET];
static krwlock_t        ipp_action_byname_lock[1];
static ipp_ref_t        *ipp_action_noname;

static kmem_cache_t     *ipp_packet_cache = NULL;
static uint_t           ipp_packet_classes = IPP_NCLASS;
static uint_t           ipp_packet_logging = 0;
static uint_t           ipp_packet_log_entries = IPP_NLOG;

/*
 * Prototypes
 */

void                    ipp_init(void);

int                     ipp_list_mods(ipp_mod_id_t **, int *);

ipp_mod_id_t            ipp_mod_lookup(const char *);
int                     ipp_mod_name(ipp_mod_id_t, char **);
int                     ipp_mod_register(const char *, ipp_ops_t *);
int                     ipp_mod_unregister(ipp_mod_id_t);
int                     ipp_mod_list_actions(ipp_mod_id_t, ipp_action_id_t **,
    int *);

ipp_action_id_t         ipp_action_lookup(const char *);
int                     ipp_action_name(ipp_action_id_t, char **);
int                     ipp_action_mod(ipp_action_id_t, ipp_mod_id_t *);
int                     ipp_action_create(ipp_mod_id_t, const char *,
    nvlist_t **, ipp_flags_t, ipp_action_id_t *);
int                     ipp_action_modify(ipp_action_id_t, nvlist_t **,
    ipp_flags_t);
int                     ipp_action_destroy(ipp_action_id_t, ipp_flags_t);
int                     ipp_action_info(ipp_action_id_t, int (*)(nvlist_t *,
    void *), void *, ipp_flags_t);
void                    ipp_action_set_ptr(ipp_action_id_t, void *);
void                    *ipp_action_get_ptr(ipp_action_id_t);
int                     ipp_action_ref(ipp_action_id_t, ipp_action_id_t,
    ipp_flags_t);
int                     ipp_action_unref(ipp_action_id_t, ipp_action_id_t,
    ipp_flags_t);

int                     ipp_packet_alloc(ipp_packet_t **, const char *,
    ipp_action_id_t);
void                    ipp_packet_free(ipp_packet_t *);
int                     ipp_packet_add_class(ipp_packet_t *, const char *,
    ipp_action_id_t);
int                     ipp_packet_process(ipp_packet_t **);
int                     ipp_packet_next(ipp_packet_t *, ipp_action_id_t);
```

```
126 void                    ipp_packet_set_data(ipp_packet_t *, mblk_t *);
127 mblk_t                  *ipp_packet_get_data(ipp_packet_t *);
128 void                    ipp_packet_set_private(ipp_packet_t *, void *,
129     void (*)(void *));
130 void                    *ipp_packet_get_private(ipp_packet_t *);

132 int                     ipp_stat_create(ipp_action_id_t, const char *, int,
133     int (*)(ipp_stat_t *, void *, int), void *, ipp_stat_t **);
134 void                    ipp_stat_install(ipp_stat_t *);
135 void                    ipp_stat_destroy(ipp_stat_t *);
136 int                     ipp_stat_named_init(ipp_stat_t *, const char *, uchar_t,
137     ipp_named_t *);
138 int                     ipp_stat_named_op(ipp_named_t *, void *, int);

140 static int              ref_mod(ipp_action_t *, ipp_mod_t *);
141 static void             unref_mod(ipp_action_t *, ipp_mod_t *);
142 static int              is_mod_busy(ipp_mod_t *);
143 static int              get_mod_ref(ipp_mod_t *, ipp_action_id_t **, int *);
144 static int              get_mods(ipp_mod_id_t **bufp, int *);
145 static ipp_mod_id_t     find_mod(const char *);
146 static int              alloc_mod(const char *, ipp_mod_id_t *);
147 static void             free_mod(ipp_mod_t *);
148 static ipp_mod_t        *hold_mod(ipp_mod_id_t);
149 static void             rele_mod(ipp_mod_t *);
150 static ipp_mod_id_t     get_mid(void);

152 static int              condemn_action(ipp_ref_t **, ipp_action_t *);
153 static int              destroy_action(ipp_action_t *, ipp_flags_t);
154 static int              ref_action(ipp_action_t *, ipp_action_t *);
155 static int              unref_action(ipp_action_t *, ipp_action_t *);
156 static int              is_action_refd(ipp_action_t *);
157 static ipp_action_id_t  find_action(const char *);
158 static int              alloc_action(const char *, ipp_action_id_t *);
159 static void             free_action(ipp_action_t *);
160 static ipp_action_t     *hold_action(ipp_action_id_t);
161 static void             rele_action(ipp_action_t *);
162 static ipp_action_id_t  get_aid(void);

164 static int              alloc_packet(const char *, ipp_action_id_t,
165     ipp_packet_t **);
166 static int              realloc_packet(ipp_packet_t *);
167 static void             free_packet(ipp_packet_t *);

169 static int              hash(const char *);
170 static int              update_stats(kstat_t *, int);
171 static void             init_mods(void);
172 static void             init_actions(void);
173 static void             init_packets(void);
174 static int              mod_constructor(void *, void *, int);
175 static void             mod_destructor(void *, void *);
176 static int              action_constructor(void *, void *, int);
177 static void             action_destructor(void *, void *);
178 static int              packet_constructor(void *, void *, int);
179 static void             packet_destructor(void *, void *);

181 /*
182  * Debug message macros
183  */

185 #ifdef  IPP_DBG

187 #define DBG_MOD         0x00000001ull
188 #define DBG_ACTION      0x00000002ull
189 #define DBG_PACKET      0x00000004ull
190 #define DBG_STATS       0x00000008ull
191 #define DBG_LIST        0x00000010ull
```



```
193 static uint64_t         ipp_debug_flags =
194 /*
195  * DBG_PACKET |
196  * DBG_STATS |
197  * DBG_LIST |
198  * DBG_MOD |
199  * DBG_ACTION |
200  */
201 0;

203 static kmutex_t debug_mutex[1];

205 /*PRINTFLIKE3*/
206 static void ipp_debug(uint64_t, const char *, char *, ...)
207         __KPRINTFLIKE(3);

209 #define DBG0(_type, _fmt)                                       \
210         ipp_debug((_type), __FN__, (_fmt));

212 #define DBG1(_type, _fmt, _a1)                                  \
213         ipp_debug((_type), __FN__, (_fmt), (_a1));

215 #define DBG2(_type, _fmt, _a1, _a2)                             \
216         ipp_debug((_type), __FN__, (_fmt), (_a1), (_a2));

218 #define DBG3(_type, _fmt, _a1, _a2, _a3)                        \
219         ipp_debug((_type), __FN__, (_fmt), (_a1), (_a2),        \
220             (_a3));

222 #define DBG4(_type, _fmt, _a1, _a2, _a3, _a4)                   \
223         ipp_debug((_type), __FN__, (_fmt), (_a1), (_a2),        \
224             (_a3), (_a4));

226 #define DBG5(_type, _fmt, _a1, _a2, _a3, _a4, _a5)              \
227         ipp_debug((_type), __FN__, (_fmt), (_a1), (_a2),        \
228             (_a3), (_a4), (_a5));

230 #else   /* IPP_DBG */

232 #define DBG0(_type, _fmt)
233 #define DBG1(_type, _fmt, _a1)
234 #define DBG2(_type, _fmt, _a1, _a2)
235 #define DBG3(_type, _fmt, _a1, _a2, _a3)
236 #define DBG4(_type, _fmt, _a1, _a2, _a3, _a4)
237 #define DBG5(_type, _fmt, _a1, _a2, _a3, _a4, _a5)

239 #endif  /* IPP_DBG */

241 /*
242  * Lock macros
243  */

245 #define LOCK_MOD(_imp, _rw)                                     \
246         rw_enter((_imp)->ippm_lock, (_rw))
247 #define UNLOCK_MOD(_imp)                                        \
248         rw_exit((_imp)->ippm_lock)

250 #define LOCK_ACTION(_ap, _rw)                                   \
251         rw_enter((_ap)->ippa_lock, (_rw))
252 #define UNLOCK_ACTION(_imp)                                     \
253         rw_exit((_imp)->ippa_lock)

255 #define CONFIG_WRITE_START(_ap)                                 \
256         CONFIG_LOCK_ENTER((_ap)->ippa_config_lock, CL_WRITE)
```

```
 258 #define CONFIG_WRITE_END(_ap)                                           \
 259         CONFIG_LOCK_EXIT((_ap)->ippa_config_lock)

 261 #define CONFIG_READ_START(_ap)                                          \
 262         CONFIG_LOCK_ENTER((_ap)->ippa_config_lock, CL_READ)

 264 #define CONFIG_READ_END(_ap)                                            \
 265         CONFIG_LOCK_EXIT((_ap)->ippa_config_lock)

 267 /*
 268  * Exported functions
 269  */

 271 #define __FN__  "ipp_init"
 272 void
 273 ipp_init(
 274         void)
 275 {
 276 #ifdef  IPP_DBG
 277         mutex_init(debug_mutex, NULL, MUTEX_ADAPTIVE,
 278             (void *)ipltospl(LOCK_LEVEL));
 279 #endif  /* IPP_DBG */

 281         /*
 282          * Initialize module and action structure caches and associated locks.
 283          */

 285         init_mods();
 286         init_actions();
 287         init_packets();
 288 }
_____unchanged_portion_omitted_
1309 #undef  __FN__

1311 #define __FN__  "ipp_packet_process"
1312 int
1313 ipp_packet_process(
1314         ipp_packet_t    **ppp)
1315 {
1316         ipp_packet_t    *pp;
1317         ipp_action_id_t aid;
1318         ipp_class_t     *cp;
1319         ipp_log_t       *lp;
1320         ipp_action_t    *ap;
1321         ipp_mod_t       *imp;
1322         ipp_ops_t       *ippo;
1323         int             rc;

1325         ASSERT(ppp != NULL);
1326         pp = *ppp;
1327         ASSERT(pp != NULL);

1329         /*
1330          * Walk the class list.
1331          */

1333         while (pp->ippp_class_rindex < pp->ippp_class_windex) {
1334                 cp = &(pp->ippp_class_array[pp->ippp_class_rindex]);

1336                 /*
1337                  * While there is a real action to invoke...
1338                  */

1340                 aid = cp->ippc_aid;
1341                 while (aid != IPP_ACTION_CONT &&
1342                     aid != IPP_ACTION_DEFER &&
```



```
1343                     aid != IPP_ACTION_DROP) {

1345                         ASSERT(aid != IPP_ACTION_INVAL);

1347                         /*
1348                          * Translate the action id to the action pointer.
1349                          */

1351                         if ((ap = hold_action(aid)) == NULL) {
1352                                 DBG1(DBG_PACKET,
1353                                     "action id '%d' not found\n", aid);
1354                                 return (ENOENT);
1355                         }

1357                         /*
1358                          * Check that the action is available for use...
1359                          */
1360                         LOCK_ACTION(ap, RW_READER);
1361                         if (ap->ippa_state != IPP_ASTATE_AVAILABLE) {
1362                                 UNLOCK_ACTION(ap);
1363                                 rele_action(ap);
1364                                 return (EPROTO);
1365                         }

1367                         /*
1368                          * Increment the action's packet count to note that
1369                          * it's being used.
1370                          *
1371                          * NOTE: We only have a read lock, so we need to use
1372                          *       atomic_add_32(). The read lock is still
1373                          *       important though as it is crucial to block
1374                          *       out a destroy operation between the action
1375                          *       state being checked and the packet count
1376                          *       being incremented.
1377                          */

1379                         atomic_inc_32(&(ap->ippa_packets));
1381                         atomic_add_32(&(ap->ippa_packets), 1);

1381                         imp = ap->ippa_mod;
1382                         ASSERT(imp != NULL);
1383                         UNLOCK_ACTION(ap);

1385                         ippo = imp->ippm_ops;
1386                         ASSERT(ippo != NULL);

1388                         /*
1389                          * If there's a log, grab the next entry and fill it
1390                          * in.
1391                          */

1393                         if (pp->ippp_log != NULL &&
1394                             pp->ippp_log_windex <= pp->ippp_log_limit) {
1395                                 lp = &(pp->ippp_log[pp->ippp_log_windex++]);
1396                                 lp->ippl_aid = aid;
1397                                 (void) strcpy(lp->ippl_name, cp->ippc_name);
1398                                 gethrestime(&lp->ippl_begin);
1399                         } else {
1400                                 lp = NULL;
1401                         }

1403                         /*
1404                          * Invoke the action.
1405                          */

1407                         rc = ippo->ippo_action_invoke(aid, pp);
```

```
1409                        /*
1410                         * Also log the time that the action finished
1411                         * processing.
1412                         */
1413
1414                        if (lp != NULL)
1415                                gethrestime(&lp->ippl_end);
1416
1417                        /*
1418                         * Decrement the packet count.
1419                         */
1420
1421                        atomic_dec_32(&(ap->ippa_packets));
1423                        atomic_add_32(&(ap->ippa_packets), -1);
1423
1423                        /*
1424                         * If the class' action id is the same now as it was
1425                         * before then clearly no 'next action' has been set.
1426                         * This is a protocol error.
1427                         */
1428
1429                        if (cp->ippc_aid == aid) {
1430                                DBG1(DBG_PACKET,
1431                                    "action '%s' did not set next action\n",
1432                                    ap->ippa_name);
1433                                rele_action(ap);
1434                                return (EPROTO);
1435                        }
1436
1437                        /*
1438                         * The action did not complete successfully. Terminate
1439                         * packet processing.
1440                         */
1441
1442                        if (rc != 0) {
1443                                DBG2(DBG_PACKET,
1444                                    "action error '%d' from action '%s'\n",
1445                                    rc, ap->ippa_name);
1446                                rele_action(ap);
1447                                return (rc);
1448                        }
1449
1450                        rele_action(ap);
1451
1452                        /*
1453                         * Look at the next action.
1454                         */
1455
1456                        aid = cp->ippc_aid;
1457                }
1458
1459                /*
1460                 * No more real actions to invoke, check for 'virtual' ones.
1461                 */
1462
1463                /*
1464                 * Packet deferred: module has held onto packet for processing
1465                 * later.
1466                 */
1467
1468                if (cp->ippc_aid == IPP_ACTION_DEFER) {
1469                        *ppp = NULL;
1470                        return (0);
1471                }
```



```
1473                /*
1474                 * Packet dropped: free the packet and discontinue processing.
1475                 */
1476
1477                if (cp->ippc_aid == IPP_ACTION_DROP) {
1478                        freemsg(pp->ippp_data);
1479                        ipp_packet_free(pp);
1480                        *ppp = NULL;
1481                        return (0);
1482                }
1483
1484                /*
1485                 * Must be 'continue processing': move onto the next class.
1486                 */
1487
1488                ASSERT(cp->ippc_aid == IPP_ACTION_CONT);
1489                pp->ippp_class_rindex++;
1490        }
1491
1492        return (0);
1493 }
______unchanged_portion_omitted_
2336 #undef __FN__
2337
2338 #define __FN__  "hold_mod"
2339 static ipp_mod_t *
2340 hold_mod(
2341        ipp_mod_id_t    mid)
2342 {
2343        ipp_mod_t       *imp;
2344
2345        if (mid < 0)
2346                return (NULL);
2347
2348        /*
2349         * Use the module id as an index into the array of all module
2350         * structures.
2351         */
2352
2353        rw_enter(ipp_mod_byid_lock, RW_READER);
2354        if ((imp = ipp_mod_byid[mid]) == NULL) {
2355                rw_exit(ipp_mod_byid_lock);
2356                return (NULL);
2357        }
2358
2359        ASSERT(imp->ippm_id == mid);
2360
2361        /*
2362         * If the modul has 'destruct pending' set then it means it is either
2363         * still in the cache (i.e not allocated) or in the process of
2364         * being set up by alloc_mod().
2365         */
2366
2367        LOCK_MOD(imp, RW_READER);
2368        if (imp->ippm_destruct_pending) {
2369                UNLOCK_MOD(imp);
2370                rw_exit(ipp_mod_byid_lock);
2371                return (NULL);
2372        }
2373        UNLOCK_MOD(imp);
2374
2375        /*
2376         * Increment the hold count to prevent the structure from being
2377         * freed.
2378         */
```

```
2380          atomic_inc_32(&(imp->ippm_hold_count));
2382          atomic_add_32(&(imp->ippm_hold_count), 1);
2381          rw_exit(ipp_mod_byid_lock);

2383          return (imp);
2384  }
2385  #undef  __FN__

2387  #define __FN__  "rele_mod"
2388  static void
2389  rele_mod(
2390          ipp_mod_t       *imp)
2391  {
2392          /*
2393           * This call means we're done with the pointer so we can drop the
2394           * hold count.
2395           */

2397          ASSERT(imp->ippm_hold_count != 0);
2398          atomic_dec_32(&(imp->ippm_hold_count));
2400          atomic_add_32(&(imp->ippm_hold_count), -1);

2400          /*
2401           * If the structure has 'destruct pending' set then we tried to free
2402           * it but couldn't, so do it now.
2403           */

2405          LOCK_MOD(imp, RW_READER);
2406          if (imp->ippm_destruct_pending && imp->ippm_hold_count == 0) {
2407                  UNLOCK_MOD(imp);
2408                  kmem_cache_free(ipp_mod_cache, imp);
2409                  return;
2410          }

2412          UNLOCK_MOD(imp);
2413  }
______unchanged_portion_omitted_
3030  #undef __FN__

3032  #define __FN__  "hold_action"
3033  static ipp_action_t *
3034  hold_action(
3035          ipp_action_id_t aid)
3036  {
3037          ipp_action_t    *ap;

3039          if (aid < 0)
3040                  return (NULL);

3042          /*
3043           * Use the action id as an index into the array of all action
3044           * structures.
3045           */

3047          rw_enter(ipp_action_byid_lock, RW_READER);
3048          if ((ap = ipp_action_byid[aid]) == NULL) {
3049                  rw_exit(ipp_action_byid_lock);
3050                  return (NULL);
3051          }

3053          /*
3054           * If the action has 'destruct pending' set then it means it is either
3055           * still in the cache (i.e not allocated) or in the process of
3056           * being set up by alloc_action().
3057           */
```



```
3059          LOCK_ACTION(ap, RW_READER);
3060          if (ap->ippa_destruct_pending) {
3061                  UNLOCK_ACTION(ap);
3062                  rw_exit(ipp_action_byid_lock);
3063                  return (NULL);
3064          }
3065          UNLOCK_ACTION(ap);

3067          /*
3068           * Increment the hold count to prevent the structure from being
3069           * freed.
3070           */

3072          atomic_inc_32(&(ap->ippa_hold_count));
3074          atomic_add_32(&(ap->ippa_hold_count), 1);
3073          rw_exit(ipp_action_byid_lock);

3075          return (ap);
3076  }
3077  #undef  __FN__

3079  #define __FN__  "rele_action"
3080  static void
3081  rele_action(
3082          ipp_action_t    *ap)
3083  {
3084          /*
3085           * This call means we're done with the pointer so we can drop the
3086           * hold count.
3087           */

3089          ASSERT(ap->ippa_hold_count != 0);
3090          atomic_dec_32(&(ap->ippa_hold_count));
3092          atomic_add_32(&(ap->ippa_hold_count), -1);

3092          /*
3093           * If the structure has 'destruct pending' set then we tried to free
3094           * it but couldn't, so do it now.
3095           */

3097          LOCK_ACTION(ap, RW_READER);
3098          if (ap->ippa_destruct_pending && ap->ippa_hold_count == 0) {
3099                  UNLOCK_ACTION(ap);
3100                  kmem_cache_free(ipp_action_cache, ap);
3101                  return;
3102          }
3103          UNLOCK_ACTION(ap);
3104  }
______unchanged_portion_omitted_
```

```
**********************************************************
    9105 Mon Jul 28 07:44:45 2014
new/usr/src/uts/common/ipp/meters/tokenmt.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2002 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/types.h>
#include <sys/kmem.h>
#include <sys/conf.h>
#include <sys/sysmacros.h>
#include <netinet/in.h>
#include <netinet/in_systm.h>
#include <netinet/ip6.h>
#include <inet/common.h>
#include <inet/ip.h>
#include <inet/ip6.h>
#include <ipp/meters/meter_impl.h>

/*
 * Module : Single or Two Rate Metering module - tokenmt
 * Description
 * This module implements the metering part of RFC 2698 & 2697. It accepts the
 * committed rate, peak rate (optional), committed burst and peak burst for a
 * flow and determines if the flow is within the cfgd. rates and assigns
 * next action appropriately..
 * If the peak rate is provided this acts as a two rate meter (RFC 2698), else
 * a single rate meter (RFC 2697). If this is a two rate meter, then
 * the outcome is either green, red or yellow. Else if this a single rate
 * meter and the peak burst size is not provided, the outcome is either
 * green or red.
 * Internally, it maintains 2 token buckets, Tc & Tp, each filled with
 * tokens equal to committed burst & peak burst respectively initially.
 * When a packet arrives, tokens in Tc or Tp are updated at the committed
 * or the peak rate up to a maximum of the committed or peak burst size.
 * If there are enough tokens in Tc, the packet is Green, else if there are
 * enough tokens in Tp, the packet is Yellow, else the packet is Red. In case
 * of Green and Yellow packets, Tc and/or Tp is updated accordingly.
 */
```



```
int tokenmt_debug = 0;

/* Updating tokens */
static void tokenmt_update_tokens(tokenmt_data_t *, hrtime_t);

/*
 * Given a packet and the tokenmt_data it belongs to, this routine meters the
 * ToS or DSCP for IPv4 and IPv6 resp. with the values configured for
 * the tokenmt_data.
 */
int
tokenmt_process(mblk_t **mpp, tokenmt_data_t *tokenmt_data,
    ipp_action_id_t *next_action)
{
        uint8_t dscp;
        ipha_t *ipha;
        ip6_t *ip6_hdr;
        uint32_t pkt_len;
        mblk_t *mp = *mpp;
        hrtime_t now;
        enum meter_colour colour;
        tokenmt_cfg_t *cfg_parms = tokenmt_data->cfg_parms;

        if (mp == NULL) {
                tokenmt0dbg(("tokenmt_process: null mp!\n"));
                atomic_inc_64(&tokenmt_data->epackets);
                atomic_add_64(&tokenmt_data->epackets, 1);
                return (EINVAL);
        }

        if (mp->b_datap->db_type != M_DATA) {
                if ((mp->b_cont != NULL) &&
                    (mp->b_cont->b_datap->db_type == M_DATA)) {
                        mp = mp->b_cont;
                } else {
                        tokenmt0dbg(("tokenmt_process: no data\n"));
                        atomic_inc_64(&tokenmt_data->epackets);
                        atomic_add_64(&tokenmt_data->epackets, 1);
                        return (EINVAL);
                }
        }

        /* Figure out the ToS/Traffic Class and length from the message */
        if ((mp->b_wptr - mp->b_rptr) < IP_SIMPLE_HDR_LENGTH) {
                if (!pullupmsg(mp, IP_SIMPLE_HDR_LENGTH)) {
                        tokenmt0dbg(("tokenmt_process: pullup error\n"));
                        atomic_inc_64(&tokenmt_data->epackets);
                        atomic_add_64(&tokenmt_data->epackets, 1);
                        return (EINVAL);
                }
        }
        ipha = (ipha_t *)mp->b_rptr;
        if (IPH_HDR_VERSION(ipha) == IPV4_VERSION) {
                /* discard last 2 unused bits */
                dscp = ipha->ipha_type_of_service;
                pkt_len = ntohs(ipha->ipha_length);
        } else {
                ip6_hdr = (ip6_t *)mp->b_rptr;
                /* discard ECN bits */
                dscp = __IPV6_TCLASS_FROM_FLOW(ip6_hdr->ip6_vcf);
                pkt_len = ntohs(ip6_hdr->ip6_plen) +
                    ip_hdr_length_v6(mp, ip6_hdr);
        }

        /* Convert into bits */
        pkt_len <<= 3;
```

```
124         now = gethrtime();
126         mutex_enter(&tokenmt_data->tokenmt_lock);
127         /* Update the token counts */
128         tokenmt_update_tokens(tokenmt_data, now);
130         /*
131          * Figure out the drop preced. for the pkt. Need to be careful here
132          * because if the mode is set to COLOUR_AWARE, then the dscp value
133          * is used regardless of whether it was explicitly set or not.
134          * If the value is defaulted to 000 (drop precd.) then the pkt
135          * will always be coloured RED.
136          */
137         if (cfg_parms->tokenmt_type == SRTCL_TOKENMT) {
138                 if (!cfg_parms->colour_aware) {
139                         if (pkt_len <= tokenmt_data->committed_tokens) {
140                                 tokenmt_data->committed_tokens -= pkt_len;
141                                 *next_action = cfg_parms->green_action;
142                         } else if (pkt_len <= tokenmt_data->peak_tokens) {
143                                 /*
144                                  * Can't do this if yellow_action is not
145                                  * configured.
146                                  */
147                                 ASSERT(cfg_parms->yellow_action !=
148                                     TOKENMT_NO_ACTION);
149                                 tokenmt_data->peak_tokens -= pkt_len;
150                                 *next_action = cfg_parms->yellow_action;
151                         } else {
152                                 *next_action = cfg_parms->red_action;
153                         }
154                 } else {
155                         colour = cfg_parms->dscp_to_colour[dscp >> 2];
156                         if ((colour == TOKENMT_GREEN) &&
157                             (pkt_len <= tokenmt_data->committed_tokens)) {
158                                 tokenmt_data->committed_tokens -= pkt_len;
159                                 *next_action = cfg_parms->green_action;
160                         } else if (((colour == TOKENMT_GREEN) ||
161                             (colour == TOKENMT_YELLOW)) &&
162                             (pkt_len <= tokenmt_data->peak_tokens)) {
163                                 /*
164                                  * Can't do this if yellow_action is not
165                                  * configured.
166                                  */
167                                 ASSERT(cfg_parms->yellow_action !=
168                                     TOKENMT_NO_ACTION);
169                                 tokenmt_data->peak_tokens -= pkt_len;
170                                 *next_action = cfg_parms->yellow_action;
171                         } else {
172                                 *next_action = cfg_parms->red_action;
173                         }
174                 }
175         } else {
176                 if (!cfg_parms->colour_aware) {
177                         if (pkt_len > tokenmt_data->peak_tokens) {
178                                 *next_action = cfg_parms->red_action;
179                         } else if (pkt_len > tokenmt_data->committed_tokens) {
180                                 /*
181                                  * Can't do this if yellow_action is not
182                                  * configured.
183                                  */
184                                 ASSERT(cfg_parms->yellow_action !=
185                                     TOKENMT_NO_ACTION);
186                                 tokenmt_data->peak_tokens -= pkt_len;
187                                 *next_action = cfg_parms->yellow_action;
188                         } else {
```



```
189                                 tokenmt_data->committed_tokens -= pkt_len;
190                                 tokenmt_data->peak_tokens -= pkt_len;
191                                 *next_action = cfg_parms->green_action;
192                         }
193                 } else {
194                         colour = cfg_parms->dscp_to_colour[dscp >> 2];
195                         if ((colour == TOKENMT_RED) ||
196                             (pkt_len > tokenmt_data->peak_tokens)) {
197                                 *next_action = cfg_parms->red_action;
198                         } else if ((colour == TOKENMT_YELLOW) ||
199                             (pkt_len > tokenmt_data->committed_tokens)) {
200                                 /*
201                                  * Can't do this if yellow_action is not
202                                  * configured.
203                                  */
204                                 ASSERT(cfg_parms->yellow_action !=
205                                     TOKENMT_NO_ACTION);
206                                 tokenmt_data->peak_tokens -= pkt_len;
207                                 *next_action = cfg_parms->yellow_action;
208                         } else {
209                                 tokenmt_data->committed_tokens -= pkt_len;
210                                 tokenmt_data->peak_tokens -= pkt_len;
211                                 *next_action = cfg_parms->green_action;
212                         }
213                 }
214         }
215         mutex_exit(&tokenmt_data->tokenmt_lock);
217         /* Update Stats */
218         if (*next_action == cfg_parms->green_action) {
219                 atomic_inc_64(&tokenmt_data->green_packets);
221                 atomic_add_64(&tokenmt_data->green_packets, 1);
220                 atomic_add_64(&tokenmt_data->green_bits, pkt_len);
221         } else if (*next_action == cfg_parms->yellow_action) {
222                 atomic_inc_64(&tokenmt_data->yellow_packets);
224                 atomic_add_64(&tokenmt_data->yellow_packets, 1);
223                 atomic_add_64(&tokenmt_data->yellow_bits, pkt_len);
224         } else {
225                 ASSERT(*next_action == cfg_parms->red_action);
226                 atomic_inc_64(&tokenmt_data->red_packets);
228                 atomic_add_64(&tokenmt_data->red_packets, 1);
227                 atomic_add_64(&tokenmt_data->red_bits, pkt_len);
228         }
230         return (0);
231 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    5759 Mon Jul 28 07:44:45 2014
new/usr/src/uts/common/ipp/meters/tswtcl.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2005 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/types.h>
#include <sys/kmem.h>
#include <sys/random.h>
#include <netinet/in.h>
#include <netinet/in_systm.h>
#include <netinet/ip6.h>
#include <inet/common.h>
#include <inet/ip.h>
#include <inet/ip6.h>
#include <ipp/meters/meter_impl.h>

/*
 * Module : Time Sliding Window meter - tswtclmtr
 * Description
 * This module implements the metering part of RFC 2859. It accepts the
 * committed rate, peak rate and the window for a flow and determines
 * if the flow is within the committed/peak rate and assigns the appropriate
 * next action.
 * The meter provides an estimate of the running average bandwidth for the
 * flow over the specified window. It uses probability to benefit TCP flows
 * as it reduces the likelihood of dropping multiple packets within a TCP
 * window without adversely effecting UDP flows.
 */

int tswtcl_debug = 0;

/*
 * Given a packet and the tswtcl_data it belongs to, this routine meters the
 * ToS or DSCP for IPv4 and IPv6 resp. with the values configured for
 * the tswtcl_data.
 */
/* ARGSUSED */
int
```



```
tswtcl_process(mblk_t **mpp, tswtcl_data_t *tswtcl_data,
    ipp_action_id_t *next_action)
{
        ipha_t *ipha;
        hrtime_t now;
        ip6_t *ip6_hdr;
        uint32_t pkt_len;
        mblk_t *mp = *mpp;
        hrtime_t deltaT;
        uint64_t bitsinwin;
        uint32_t min = 0, additive, rnd;
        tswtcl_cfg_t *cfg_parms = tswtcl_data->cfg_parms;

        if (mp == NULL) {
                tswtcl0dbg(("tswtcl_process: null mp!\n"));
                atomic_inc_64(&tswtcl_data->epackets);
                atomic_add_64(&tswtcl_data->epackets, 1);
                return (EINVAL);
        }

        if (mp->b_datap->db_type != M_DATA) {
                if ((mp->b_cont != NULL) &&
                    (mp->b_cont->b_datap->db_type == M_DATA)) {
                        mp = mp->b_cont;
                } else {
                        tswtcl0dbg(("tswtcl_process: no data\n"));
                        atomic_inc_64(&tswtcl_data->epackets);
                        atomic_add_64(&tswtcl_data->epackets, 1);
                        return (EINVAL);
                }
        }

        /* Figure out the ToS/Traffic Class and length from the message */
        if ((mp->b_wptr - mp->b_rptr) < IP_SIMPLE_HDR_LENGTH) {
                if (!pullupmsg(mp, IP_SIMPLE_HDR_LENGTH)) {
                        tswtcl0dbg(("tswtcl_process: pullup error\n"));
                        atomic_inc_64(&tswtcl_data->epackets);
                        atomic_add_64(&tswtcl_data->epackets, 1);
                        return (EINVAL);
                }
        }
        ipha = (ipha_t *)mp->b_rptr;
        if (IPH_HDR_VERSION(ipha) == IPV4_VERSION) {
                pkt_len = ntohs(ipha->ipha_length);
        } else {
                ip6_hdr = (ip6_t *)mp->b_rptr;
                pkt_len = ntohs(ip6_hdr->ip6_plen) +
                    ip_hdr_length_v6(mp, ip6_hdr);
        }

        /* Convert into bits */
        pkt_len <<= 3;

        /* Get current time */
        now = gethrtime();

        /* Update the avg_rate and win_front tswtcl_data */
        mutex_enter(&tswtcl_data->tswtcl_lock);

        /* avg_rate = bits/sec and window in msec */
        bitsinwin = ((uint64_t)tswtcl_data->avg_rate * cfg_parms->window /
            1000) + pkt_len;

        deltaT = now - tswtcl_data->win_front + cfg_parms->nsecwindow;

        tswtcl_data->avg_rate = (uint64_t)bitsinwin * METER_SEC_TO_NSEC /
```

```
123             deltaT;
124         tswtcl_data->win_front = now;

126         if (tswtcl_data->avg_rate <= cfg_parms->committed_rate) {
127                 *next_action = cfg_parms->green_action;
128         } else if (tswtcl_data->avg_rate <= cfg_parms->peak_rate) {
129                 /*
130                  * Compute the probability:
131                  *
132                  * p0 = (avg_rate - committed_rate) / avg_rate
133                  *
134                  * Yellow with probability p0
135                  * Green with probability (1 - p0)
136                  *
137                  */
138                 uint32_t aminusc;

140                 /* Get a random no. betweeen 0 and avg_rate */
141                 (void) random_get_pseudo_bytes((uint8_t *)&additive,
142                     sizeof (additive));
143                 rnd = min + (additive % (tswtcl_data->avg_rate - min + 1));

145                 aminusc = tswtcl_data->avg_rate - cfg_parms->committed_rate;
146                 if (aminusc >= rnd) {
147                         *next_action = cfg_parms->yellow_action;
148                 } else {
149                         *next_action = cfg_parms->green_action;
150                 }
151         } else {
152                 /*
153                  * Compute the probability:
154                  *
155                  * p1 = (avg_rate - peak_rate) / avg_rate
156                  * p2 = (peak_rate - committed_rate) / avg_rate
157                  *
158                  * Red with probability p1
159                  * Yellow with probability p2
160                  * Green with probability (1 - (p1 + p2))
161                  *
162                  */
163                 uint32_t  aminusp;

165                 /* Get a random no. betweeen 0 and avg_rate */
166                 (void) random_get_pseudo_bytes((uint8_t *)&additive,
167                     sizeof (additive));
168                 rnd = min + (additive % (tswtcl_data->avg_rate - min + 1));

170                 aminusp = tswtcl_data->avg_rate - cfg_parms->peak_rate;

172                 if (aminusp >= rnd) {
173                         *next_action = cfg_parms->red_action;
174                 } else if ((cfg_parms->pminusc + aminusp) >= rnd) {
175                         *next_action = cfg_parms->yellow_action;
176                 } else {
177                         *next_action = cfg_parms->green_action;
178                 }

180         }
181         mutex_exit(&tswtcl_data->tswtcl_lock);

183         /* Update Stats */
184         if (*next_action == cfg_parms->green_action) {
185                 atomic_inc_64(&tswtcl_data->green_packets);
187                 atomic_add_64(&tswtcl_data->green_packets, 1);
186                 atomic_add_64(&tswtcl_data->green_bits, pkt_len);
187         } else if (*next_action == cfg_parms->yellow_action) {
```



```
188                 atomic_inc_64(&tswtcl_data->yellow_packets);
190                 atomic_add_64(&tswtcl_data->yellow_packets, 1);
189                 atomic_add_64(&tswtcl_data->yellow_bits, pkt_len);
190         } else {
191                 ASSERT(*next_action == cfg_parms->red_action);
192                 atomic_inc_64(&tswtcl_data->red_packets);
194                 atomic_add_64(&tswtcl_data->red_packets, 1);
193                 atomic_add_64(&tswtcl_data->red_bits, pkt_len);
194         }
195         return (0);
196 }
```

*unchanged_portion_omitted_*

****
2990 Mon Jul 28 07:44:45 2014
new/usr/src/uts/common/os/audit_memory.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/param.h>
#include <sys/types.h>
#include <sys/kmem.h>
#include <c2/audit.h>
#include <c2/audit_kernel.h>


/* process audit data (pad) cache */
kmem_cache_t *au_pad_cache;

/*
 * increment audit path reference count
 */
void
au_pathhold(struct audit_path *app)
{
        atomic_inc_32(&app->audp_ref);
        atomic_add_32(&app->audp_ref, 1);
}

/*
 * decrement audit path reference count
 */
void
au_pathrele(struct audit_path *app)
{
        if (atomic_dec_32_nv(&app->audp_ref) > 0)
        if (atomic_add_32_nv(&app->audp_ref, -1) > 0)
                return;
        kmem_free(app, app->audp_size);
}
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   49364 Mon Jul 28 07:44:45 2014
new/usr/src/uts/common/os/bio.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1202 /*
1203  * Wait for I/O completion on the buffer; return error code.
1204  * If bp was for synchronous I/O, bp is invalid and associated
1205  * resources are freed on return.
1206  */
1207 int
1208 biowait(struct buf *bp)
1209 {
1210         int error = 0;
1211         struct cpu *cpup;

1213         ASSERT(SEMA_HELD(&bp->b_sem));

1215         cpup = CPU;
1216         atomic_inc_64(&cpup->cpu_stats.sys.iowait);
1216         atomic_add_64(&cpup->cpu_stats.sys.iowait, 1);
1217         DTRACE_IO1(wait__start, struct buf *, bp);

1219         /*
1220          * In case of panic, busy wait for completion
1221          */
1222         if (panicstr) {
1223                 while ((bp->b_flags & B_DONE) == 0)
1224                         drv_usecwait(10);
1225         } else
1226                 sema_p(&bp->b_io);

1228         DTRACE_IO1(wait__done, struct buf *, bp);
1229         atomic_dec_64(&cpup->cpu_stats.sys.iowait);
1229         atomic_add_64(&cpup->cpu_stats.sys.iowait, -1);

1231         error = geterror(bp);
1232         if ((bp->b_flags & B_ASYNC) == 0) {
1233                 if (bp->b_flags & B_REMAPPED)
1234                         bp_mapout(bp);
1235         }
1236         return (error);
1237 }
______unchanged_portion_omitted_
```

```
**********************************************************
   74610 Mon Jul 28 07:44:46 2014
new/usr/src/uts/common/os/clock.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1922 static uint_t deadman_seconds;
1923 static uint32_t deadman_panics;
1924 static int deadman_enabled = 0;
1925 static int deadman_panic_timers = 1;

1927 static void
1928 deadman(void)
1929 {
1930         if (panicstr) {
1931                 /*
1932                  * During panic, other CPUs besides the panic
1933                  * master continue to handle cyclics and some other
1934                  * interrupts.  The code below is intended to be
1935                  * single threaded, so any CPU other than the master
1936                  * must keep out.
1937                  */
1938                 if (CPU->cpu_id != panic_cpu.cpu_id)
1939                         return;

1941                 if (!deadman_panic_timers)
1942                         return; /* allow all timers to be manually disabled */

1944                 /*
1945                  * If we are generating a crash dump or syncing filesystems and
1946                  * the corresponding timer is set, decrement it and re-enter
1947                  * the panic code to abort it and advance to the next state.
1948                  * The panic states and triggers are explained in panic.c.
1949                  */
1950                 if (panic_dump) {
1951                         if (dump_timeleft && (--dump_timeleft == 0)) {
1952                                 panic("panic dump timeout");
1953                                 /*NOTREACHED*/
1954                         }
1955                 } else if (panic_sync) {
1956                         if (sync_timeleft && (--sync_timeleft == 0)) {
1957                                 panic("panic sync timeout");
1958                                 /*NOTREACHED*/
1959                         }
1960                 }

1962                 return;
1963         }

1965         if (deadman_counter != CPU->cpu_deadman_counter) {
1966                 CPU->cpu_deadman_counter = deadman_counter;
1967                 CPU->cpu_deadman_countdown = deadman_seconds;
1968                 return;
1969         }

1971         if (--CPU->cpu_deadman_countdown > 0)
1972                 return;

1974         /*
1975          * Regardless of whether or not we actually bring the system down,
1976          * bump the deadman_panics variable.
1977          *
1978          * N.B. deadman_panics is incremented once for each CPU that
1979          * passes through here.  It's expected that all the CPUs will
1980          * detect this condition within one second of each other, so
```



```
1981          * when deadman_enabled is off, deadman_panics will
1982          * typically be a multiple of the total number of CPUs in
1983          * the system.
1984          */
1985         atomic_inc_32(&deadman_panics);
1985         atomic_add_32(&deadman_panics, 1);

1987         if (!deadman_enabled) {
1988                 CPU->cpu_deadman_countdown = deadman_seconds;
1989                 return;
1990         }

1992         /*
1993          * If we're here, we want to bring the system down.
1994          */
1995         panic("deadman: timed out after %d seconds of clock "
1996             "inactivity", deadman_seconds);
1997         /*NOTREACHED*/
1998 }
______unchanged_portion_omitted_
```

****
69487 Mon Jul 28 07:44:46 2014
new/usr/src/uts/common/os/contract.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____unchanged_portion_omitted_

```
2285 /*
2286  * cte_publish_all
2287  *
2288  * Publish an event to all necessary event queues.  The event, e, must
2289  * be zallocated by the caller, and the event's flags and type must be
2290  * set.  The rest of the event's fields are initialized here.
2291  */
2292 uint64_t
2293 cte_publish_all(contract_t *ct, ct_kevent_t *e, nvlist_t *data, nvlist_t *gdata)
2294 {
2295         ct_equeue_t *q;
2296         timespec_t ts;
2297         uint64_t evid;
2298         ct_kevent_t *negev;
2299         int negend;

2301         e->cte_contract = ct;
2302         e->cte_data = data;
2303         e->cte_gdata = gdata;
2304         e->cte_refs = 3;
2305         evid = e->cte_id = atomic_inc_64_nv(&ct->ct_type->ct_type_evid);
2305         evid = e->cte_id = atomic_add_64_nv(&ct->ct_type->ct_type_evid, 1);
2306         contract_hold(ct);

2308         /*
2309          * For a negotiation event we set the ct->ct_nevent field of the
2310          * contract for the duration of the negotiation
2311          */
2312         negend = 0;
2313         if (e->cte_flags & CTE_NEG) {
2314                 cte_hold(e);
2315                 ct->ct_nevent = e;
2316         } else if (e->cte_type == CT_EV_NEGEND) {
2317                 negend = 1;
2318         }

2320         gethrestime(&ts);

2322         /*
2323          * ct_evtlock simply (and only) ensures that two events sent
2324          * from the same contract are delivered to all queues in the
2325          * same order.
2326          */
2327         mutex_enter(&ct->ct_evtlock);

2329         /*
2330          * CTEL_CONTRACT - First deliver to the contract queue, acking
2331          * the event if the contract has been orphaned.
2332          */
2333         mutex_enter(&ct->ct_lock);
2334         mutex_enter(&ct->ct_events.ctq_lock);
2335         if ((e->cte_flags & CTE_INFO) == 0) {
2336                 if (ct->ct_state >= CTS_ORPHAN)
2337                         e->cte_flags |= CTE_ACK;
2338                 else
2339                         ct->ct_evcnt++;
2340         }
2341         mutex_exit(&ct->ct_lock);
2342         cte_publish(&ct->ct_events, e, &ts, B_FALSE);
```



```
2344         /*
2345          * CTEL_BUNDLE - Next deliver to the contract type's bundle
2346          * queue.
2347          */
2348         mutex_enter(&ct->ct_type->ct_type_events.ctq_lock);
2349         cte_publish(&ct->ct_type->ct_type_events, e, &ts, B_FALSE);

2351         /*
2352          * CTEL_PBUNDLE - Finally, if the contract has an owner,
2353          * deliver to the owner's process bundle queue.
2354          */
2355         mutex_enter(&ct->ct_lock);
2356         if (ct->ct_owner) {
2357                 /*
2358                  * proc_exit doesn't free event queues until it has
2359                  * abandoned all contracts.
2360                  */
2361                 ASSERT(ct->ct_owner->p_ct_equeue);
2362                 ASSERT(ct->ct_owner->p_ct_equeue[ct->ct_type->ct_type_index]);
2363                 q = ct->ct_owner->p_ct_equeue[ct->ct_type->ct_type_index];
2364                 mutex_enter(&q->ctq_lock);
2365                 mutex_exit(&ct->ct_lock);

2367                 /*
2368                  * It is possible for this code to race with adoption; we
2369                  * publish the event indicating that the event may already
2370                  * be enqueued because adoption beat us to it (in which case
2371                  * cte_pubish() does nothing).
2372                  */
2373                 cte_publish(q, e, &ts, B_TRUE);
2374         } else {
2375                 mutex_exit(&ct->ct_lock);
2376                 cte_rele(e);
2377         }

2379         if (negend) {
2380                 mutex_enter(&ct->ct_lock);
2381                 negev = ct->ct_nevent;
2382                 ct->ct_nevent = NULL;
2383                 cte_rele(negev);
2384                 mutex_exit(&ct->ct_lock);
2385         }

2387         mutex_exit(&ct->ct_evtlock);

2389         return (evid);
2390 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   32672 Mon Jul 28 07:44:46 2014
new/usr/src/uts/common/os/cred.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 340 /*
 341  * Put a hold on a cred structure.
 342  */
 343 void
 344 crhold(cred_t *cr)
 345 {
 346         ASSERT(cr->cr_ref != 0xdeadbeef && cr->cr_ref != 0);
 347         atomic_inc_32(&cr->cr_ref);
 347         atomic_add_32(&cr->cr_ref, 1);
 348 }

 350 /*
 351  * Release previous hold on a cred structure.  Free it if refcnt == 0.
 352  * If cred uses label different from zone label, free it.
 353  */
 354 void
 355 crfree(cred_t *cr)
 356 {
 357         ASSERT(cr->cr_ref != 0xdeadbeef && cr->cr_ref != 0);
 358         if (atomic_dec_32_nv(&cr->cr_ref) == 0) {
 358         if (atomic_add_32_nv(&cr->cr_ref, -1) == 0) {
 359                 ASSERT(cr != kcred);
 360                 if (cr->cr_label)
 361                         label_rele(cr->cr_label);
 362                 if (cr->cr_klpd)
 363                         crklpd_rele(cr->cr_klpd);
 364                 if (cr->cr_zone)
 365                         zone_cred_rele(cr->cr_zone);
 366                 if (cr->cr_ksid)
 367                         kcrsid_rele(cr->cr_ksid);
 368                 if (cr->cr_grps)
 369                         crgrprele(cr->cr_grps);

 371                 kmem_cache_free(cred_cache, cr);
 372         }
 373 }
______unchanged_portion_omitted_

1467 void
1468 crgrprele(credgrp_t *grps)
1469 {
1470         if (atomic_dec_32_nv(&grps->crg_ref) == 0)
1470         if (atomic_add_32_nv(&grps->crg_ref, -1) == 0)
1471                 kmem_free(grps, CREDGRPSZ(grps->crg_ngroups));
1472 }

1474 static void
1475 crgrphold(credgrp_t *grps)
1476 {
1477         atomic_inc_32(&grps->crg_ref);
1477         atomic_add_32(&grps->crg_ref, 1);
1478 }
______unchanged_portion_omitted_
```

****
41073 Mon Jul 28 07:44:46 2014
new/usr/src/uts/common/os/ddi_intr.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
645 int
646 ddi_intr_dup_handler(ddi_intr_handle_t org, int dup_inum,
647     ddi_intr_handle_t *dup)
648 {
649         ddi_intr_handle_impl_t  *hdlp = (ddi_intr_handle_impl_t *)org;
650         ddi_intr_handle_impl_t  *dup_hdlp;
651         int                     ret;

653         DDI_INTR_APIDBG((CE_CONT, "ddi_intr_dup_handler: hdlp = 0x%p\n",
654             (void *)hdlp));

656         /* Do some input argument checking ("dup" handle is not allocated) */
657         if ((hdlp == NULL) || (*dup != NULL) || (dup_inum < 0)) {
658                 DDI_INTR_APIDBG((CE_CONT, "ddi_intr_dup_handler: Invalid "
659                     "input args\n"));
660                 return (DDI_EINVAL);
661         }

663         rw_enter(&hdlp->ih_rwlock, RW_READER);

665         /* Do some input argument checking */
666         if ((hdlp->ih_state == DDI_IHDL_STATE_ALLOC) || /* intr handle alloc? */
667             (hdlp->ih_type != DDI_INTR_TYPE_MSIX) ||    /* only MSI-X allowed */
668             (hdlp->ih_flags & DDI_INTR_MSIX_DUP)) {     /* only dup original */
669                 rw_exit(&hdlp->ih_rwlock);
670                 return (DDI_EINVAL);
671         }

673         hdlp->ih_scratch1 = dup_inum;
674         ret = i_ddi_intr_ops(hdlp->ih_dip, hdlp->ih_dip,
675             DDI_INTROP_DUPVEC, hdlp, NULL);

677         if (ret == DDI_SUCCESS) {
678                 dup_hdlp = (ddi_intr_handle_impl_t *)
679                     kmem_alloc(sizeof (ddi_intr_handle_impl_t), KM_SLEEP);

681                 atomic_inc_32(&hdlp->ih_dup_cnt);
681                 atomic_add_32(&hdlp->ih_dup_cnt, 1);

683                 *dup = (ddi_intr_handle_t)dup_hdlp;
684                 bcopy(hdlp, dup_hdlp, sizeof (ddi_intr_handle_impl_t));

686                 /* These fields are unique to each dupped msi-x vector */
687                 rw_init(&dup_hdlp->ih_rwlock, NULL, RW_DRIVER, NULL);
688                 dup_hdlp->ih_state = DDI_IHDL_STATE_ADDED;
689                 dup_hdlp->ih_inum = dup_inum;
690                 dup_hdlp->ih_flags |= DDI_INTR_MSIX_DUP;
691                 dup_hdlp->ih_dup_cnt = 0;

693                 /* Point back to original vector */
694                 dup_hdlp->ih_main = hdlp;
695         }

697         rw_exit(&hdlp->ih_rwlock);
698         return (ret);
699 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   31530 Mon Jul 28 07:44:47 2014
new/usr/src/uts/common/os/ddifm.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 290 /*
 291  * fm_dev_ereport_postv: Common consolidation private interface to
 292  * post a device tree oriented dev_scheme ereport. The device tree is
 293  * composed of the following entities: devinfo nodes, minor nodes, and
 294  * pathinfo nodes. All entities are associated with some devinfo node,
 295  * either directly or indirectly. The intended devinfo node association
 296  * for the ereport is communicated by the 'dip' argument. A minor node,
 297  * an entity below 'dip', is represented by a non-null 'minor_name'
 298  * argument. An application specific caller, like scsi_fm_ereport_post,
 299  * can override the devinfo path with a pathinfo path via a non-null
 300  * 'devpath' argument - in this case 'dip' is the MPXIO client node and
 301  * devpath should be the path through the pHCI devinfo node to the
 302  * pathinfo node.
 303  *
 304  * This interface also allows the caller to decide if the error being
 305  * reported is know to be associated with a specific device identity
 306  * via the 'devid' argument. The caller needs to control wether the
 307  * devid appears as an authority in the FMRI because for some types of
 308  * errors, like transport errors, the identity of the device on the
 309  * other end of the transport is not guaranteed to be the current
 310  * identity of the dip. For transport errors the caller should specify
 311  * a NULL devid, even when there is a valid devid associated with the dip.
 312  *
 313  * The ddi_fm_ereport_post() implementation calls this interface with
 314  * just a dip: devpath, minor_name, and devid are all NULL. The
 315  * scsi_fm_ereport_post() implementation may call this interface with
 316  * non-null devpath, minor_name, and devid arguments depending on
 317  * wether MPXIO is enabled, and wether a transport or non-transport
 318  * error is being posted.
 319  *
 320  * Additional event payload is specified via the varargs plist and, if
 321  * not NULL, the nvlist passed in (such an nvlist will be merged into
 322  * the payload; the caller is responsible for freeing this nvlist).
 323  * Do not specify any high-level protocol event member names as part of the
 324  * payload - eg no payload to be named "class", "version", "detector" etc
 325  * or they will replace the members we construct here.
 326  *
 327  * The 'target-port-l0id' argument is SCSI specific. It is used
 328  * by SCSI enumeration code when a devid is unavailable. If non-NULL
 329  * the property-value becomes part of the ereport detector. The value
 330  * specified might match one of the target-port-l0ids values of a
 331  * libtopo disk chassis node. When libtopo finds a disk with a guaranteed
 332  * unique wWWN target-port of a single-lun 'real' disk, it can add
 333  * the target-port value to the libtopo disk chassis node target-port-l0ids
 334  * string array property. Kernel code has no idea if this type of
 335  * libtopo chassis node exists, or if matching will in fact occur.
 336  */
 337 void
 338 fm_dev_ereport_postv(dev_info_t *dip, dev_info_t *eqdip,
 339     const char *devpath, const char *minor_name, const char *devid,
 340     const char *tpl0, const char *error_class, uint64_t ena, int sflag,
 341     nvlist_t *pl, va_list ap)
 342 {
 343         nv_alloc_t              *nva = NULL;
 344         struct i_ddi_fmhdl      *fmhdl = NULL;
 345         errorq_elem_t           *eqep;
 346         nvlist_t                *ereport = NULL;
 347         nvlist_t                *detector = NULL;
 348         char                    *name;
```



```
 349         data_type_t             type;
 350         uint8_t                 version;
 351         char                    class[ERPT_CLASS_SZ];
 352         char                    path[MAXPATHLEN];

 354         ASSERT(ap != NULL);     /* must supply at least ereport version */
 355         ASSERT(dip && eqdip && error_class);

 357         /*
 358          * This interface should be called with a fm_capable eqdip. The
 359          * ddi_fm_ereport_post* interfaces call with eqdip == dip,
 360          * ndi_fm_ereport_post* interfaces call with eqdip == ddi_parent(dip).
 361          */
 362         if (!DDI_FM_EREPORT_CAP(ddi_fm_capable(eqdip)))
 363                 goto err;

 365         /* get ereport nvlist handle */
 366         if ((sflag == DDI_SLEEP) && !panicstr) {
 367                 /*
 368                  * Driver defect - should not call with DDI_SLEEP while in
 369                  * interrupt context.
 370                  */
 371                 if (servicing_interrupt()) {
 372                         i_ddi_drv_ereport_post(dip, DVR_ECONTEXT, NULL, sflag);
 373                         goto err;
 374                 }

 376                 /* Use normal interfaces to allocate memory. */
 377                 if ((ereport = fm_nvlist_create(NULL)) == NULL)
 378                         goto err;
 379                 ASSERT(nva == NULL);
 380         } else {
 381                 /* Use errorq interfaces to avoid memory allocation. */
 382                 fmhdl = DEVI(eqdip)->devi_fmhdl;
 383                 ASSERT(fmhdl);
 384                 eqep = errorq_reserve(fmhdl->fh_errorq);
 385                 if (eqep == NULL)
 386                         goto err;

 388                 ereport = errorq_elem_nvl(fmhdl->fh_errorq, eqep);
 389                 nva = errorq_elem_nva(fmhdl->fh_errorq, eqep);
 390                 ASSERT(nva);
 391         }
 392         ASSERT(ereport);

 394         /*
 395          * Form parts of an ereport:
 396          *      A: version
 397          *      B: error_class
 398          *      C: ena
 399          *      D: detector     (path and optional devid authority)
 400          *      E: payload
 401          *
 402          * A: ereport version: first payload tuple must be the version.
 403          */
 404         name = va_arg(ap, char *);
 405         type = va_arg(ap, data_type_t);
 406         version = va_arg(ap, uint_t);
 407         if ((strcmp(name, FM_VERSION) != 0) || (type != DATA_TYPE_UINT8)) {
 408                 i_ddi_drv_ereport_post(dip, DVR_EVER, NULL, sflag);
 409                 goto err;
 410         }

 412         /* B: ereport error_class: add "io." prefix to class. */
 413         (void) snprintf(class, ERPT_CLASS_SZ, "%s.%s",
 414             DDI_IO_CLASS, error_class);
```

```
 416         /* C: ereport ena: if not passed in, generate new ena. */
 417         if (ena == 0)
 418                 ena = fm_ena_generate(0, FM_ENA_FMT1);

 420         /* D: detector: form dev scheme fmri with path and devid. */
 421         if (devpath) {
 422                 (void) strlcpy(path, devpath, sizeof (path));
 423         } else {
 424                 /* derive devpath from dip */
 425                 if (dip == ddi_root_node())
 426                         (void) strcpy(path, "/");
 427                 else
 428                         (void) ddi_pathname(dip, path);
 429         }
 430         if (minor_name) {
 431                 (void) strlcat(path, ":", sizeof (path));
 432                 (void) strlcat(path, minor_name, sizeof (path));
 433         }
 434         detector = fm_nvlist_create(nva);
 435         fm_fmri_dev_set(detector, FM_DEV_SCHEME_VERSION, NULL, path,
 436             devid, tpl0);

 438         /* Pull parts of ereport together into ereport. */
 439         fm_ereport_set(ereport, version, class, ena, detector, NULL);

 441         /* Merge any preconstructed payload into the event. */
 442         if (pl)
 443                 (void) nvlist_merge(ereport, pl, 0);

 445         /* Add any remaining (after version) varargs payload to ereport. */
 446         name = va_arg(ap, char *);
 447         (void) i_fm_payload_set(ereport, name, ap);

 449         /* Post the ereport. */
 450         if (nva)
 451                 errorq_commit(fmhdl->fh_errorq, eqep, ERRORQ_ASYNC);
 452         else
 453                 fm_ereport_post(ereport, EVCH_SLEEP);
 454         goto out;

 456         /* Count errors as drops. */
 457 err:    if (fmhdl)
 458                 atomic_inc_64(&fmhdl->fh_kstat.fek_erpt_dropped.value.ui64);
 458                 atomic_add_64(&fmhdl->fh_kstat.fek_erpt_dropped.value.ui64, 1);

 460         /* Free up nvlists if normal interfaces were used to allocate memory */
 461 out:    if (ereport && (nva == NULL))
 462                 fm_nvlist_destroy(ereport, FM_NVA_FREE);
 463         if (detector && (nva == NULL))
 464                 fm_nvlist_destroy(detector, FM_NVA_FREE);
 465 }
_____unchanged_portion_omitted_

1011 void
1012 i_ddi_fm_acc_err_set(ddi_acc_handle_t handle, uint64_t ena, int status,
1013     int flag)
1014 {
1015         ddi_acc_hdl_t *hdlp = impl_acc_hdl_get(handle);
1016         ddi_acc_impl_t *i_hdlp = (ddi_acc_impl_t *)handle;
1017         struct i_ddi_fmhdl *fmhdl = DEVI(hdlp->ah_dip)->devi_fmhdl;

1019         i_hdlp->ahi_err->err_ena = ena;
1020         i_hdlp->ahi_err->err_status = status;
1021         i_hdlp->ahi_err->err_expected = flag;
1022         atomic_inc_64(&fmhdl->fh_kstat.fek_acc_err.value.ui64);
```



```
1022         atomic_add_64(&fmhdl->fh_kstat.fek_acc_err.value.ui64, 1);
1023 }

1025 void
1026 i_ddi_fm_dma_err_set(ddi_dma_handle_t handle, uint64_t ena, int status,
1027     int flag)
1028 {
1029         ddi_dma_impl_t *hdlp = (ddi_dma_impl_t *)handle;
1030         struct i_ddi_fmhdl *fmhdl = DEVI(hdlp->dmai_rdip)->devi_fmhdl;

1032         hdlp->dmai_error.err_ena = ena;
1033         hdlp->dmai_error.err_status = status;
1034         hdlp->dmai_error.err_expected = flag;
1035         atomic_inc_64(&fmhdl->fh_kstat.fek_dma_err.value.ui64);
1035         atomic_add_64(&fmhdl->fh_kstat.fek_dma_err.value.ui64, 1);
1036 }
_____unchanged_portion_omitted_
```

```
**********************************************************
  235631 Mon Jul 28 07:44:47 2014
new/usr/src/uts/common/os/devcfg.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1538 /*
1539  * Wrapper for making multiple state transitions
1540  */

1542 /*
1543  * i_ndi_config_node: upgrade dev_info node into a specified state.
1544  * It is a bit tricky because the locking protocol changes before and
1545  * after a node is bound to a driver. All locks are held external to
1546  * this function.
1547  */
1548 int
1549 i_ndi_config_node(dev_info_t *dip, ddi_node_state_t state, uint_t flag)
1550 {
1551         _NOTE(ARGUNUSED(flag))
1552         int rv = DDI_SUCCESS;

1554         ASSERT(DEVI_BUSY_OWNED(ddi_get_parent(dip)));

1556         while ((i_ddi_node_state(dip) < state) && (rv == DDI_SUCCESS)) {

1558                 /* don't allow any more changes to the device tree */
1559                 if (devinfo_freeze) {
1560                         rv = DDI_FAILURE;
1561                         break;
1562                 }

1564                 switch (i_ddi_node_state(dip)) {
1565                 case DS_PROTO:
1566                         /*
1567                          * only caller can reference this node, no external
1568                          * locking needed.
1569                          */
1570                         link_node(dip);
1571                         translate_devid((dev_info_t *)dip);
1572                         i_ddi_set_node_state(dip, DS_LINKED);
1573                         break;
1574                 case DS_LINKED:
1575                         /*
1576                          * Three code path may attempt to bind a node:
1577                          * - boot code
1578                          * - add_drv
1579                          * - hotplug thread
1580                          * Boot code is single threaded, add_drv synchronize
1581                          * on a userland lock, and hotplug synchronize on
1582                          * hotplug_lk. There could be a race between add_drv
1583                          * and hotplug thread. We'll live with this until the
1584                          * conversion to top-down loading.
1585                          */
1586                         if ((rv = bind_node(dip)) == DDI_SUCCESS)
1587                                 i_ddi_set_node_state(dip, DS_BOUND);

1589                         break;
1590                 case DS_BOUND:
1591                         /*
1592                          * The following transitions synchronizes on the
1593                          * per-driver busy changing flag, since we already
1594                          * have a driver.
1595                          */
1596                         if ((rv = init_node(dip)) == DDI_SUCCESS)
```



```
1597                                 i_ddi_set_node_state(dip, DS_INITIALIZED);
1598                         break;
1599                 case DS_INITIALIZED:
1600                         if ((rv = probe_node(dip)) == DDI_SUCCESS)
1601                                 i_ddi_set_node_state(dip, DS_PROBED);
1602                         break;
1603                 case DS_PROBED:
1604                         /*
1605                          * If node is retired and persistent, then prevent
1606                          * attach. We can't do this for non-persistent nodes
1607                          * as we would lose evidence that the node existed.
1608                          */
1609                         if (i_ddi_check_retire(dip) == 1 &&
1610                             ndi_dev_is_persistent_node(dip) &&
1611                             retire_prevents_attach == 1) {
1612                                 rv = DDI_FAILURE;
1613                                 break;
1614                         }
1615                         atomic_inc_ulong(&devinfo_attach_detach);
1615                         atomic_add_long(&devinfo_attach_detach, 1);
1616                         if ((rv = attach_node(dip)) == DDI_SUCCESS)
1617                                 i_ddi_set_node_state(dip, DS_ATTACHED);
1618                         atomic_dec_ulong(&devinfo_attach_detach);
1618                         atomic_add_long(&devinfo_attach_detach, -1);
1619                         break;
1620                 case DS_ATTACHED:
1621                         if ((rv = postattach_node(dip)) == DDI_SUCCESS)
1622                                 i_ddi_set_node_state(dip, DS_READY);
1623                         break;
1624                 case DS_READY:
1625                         break;
1626                 default:
1627                         /* should never reach here */
1628                         ASSERT("unknown devinfo state");
1629                 }
1630         }

1632         if (ddidebug & DDI_AUDIT)
1633                 da_log_enter(dip);
1634         return (rv);
1635 }

1637 /*
1638  * i_ndi_unconfig_node: downgrade dev_info node into a specified state.
1639  */
1640 int
1641 i_ndi_unconfig_node(dev_info_t *dip, ddi_node_state_t state, uint_t flag)
1642 {
1643         int     rv = DDI_SUCCESS;

1645         ASSERT(DEVI_BUSY_OWNED(ddi_get_parent(dip)));

1647         while ((i_ddi_node_state(dip) > state) && (rv == DDI_SUCCESS)) {

1649                 /* don't allow any more changes to the device tree */
1650                 if (devinfo_freeze) {
1651                         rv = DDI_FAILURE;
1652                         break;
1653                 }

1655                 switch (i_ddi_node_state(dip)) {
1656                 case DS_PROTO:
1657                         break;
1658                 case DS_LINKED:
1659                         /*
1660                          * Persistent nodes are only removed by hotplug code
```

```
1661                         * .conf nodes synchronizes on per-driver list.
1662                         */
1663                        if ((rv = unlink_node(dip)) == DDI_SUCCESS)
1664                                i_ddi_set_node_state(dip, DS_PROTO);
1665                        break;
1666                case DS_BOUND:
1667                        /*
1668                         * The following transitions synchronizes on the
1669                         * per-driver busy changing flag, since we already
1670                         * have a driver.
1671                         */
1672                        if ((rv = unbind_node(dip)) == DDI_SUCCESS)
1673                                i_ddi_set_node_state(dip, DS_LINKED);
1674                        break;
1675                case DS_INITIALIZED:
1676                        if ((rv = uninit_node(dip)) == DDI_SUCCESS)
1677                                i_ddi_set_node_state(dip, DS_BOUND);
1678                        break;
1679                case DS_PROBED:
1680                        if ((rv = unprobe_node(dip)) == DDI_SUCCESS)
1681                                i_ddi_set_node_state(dip, DS_INITIALIZED);
1682                        break;
1683                case DS_ATTACHED:
1684                        atomic_inc_ulong(&devinfo_attach_detach);
1684                        atomic_add_long(&devinfo_attach_detach, 1);

1686                        mutex_enter(&(DEVI(dip)->devi_lock));
1687                        DEVI_SET_DETACHING(dip);
1688                        mutex_exit(&(DEVI(dip)->devi_lock));

1690                        membar_enter(); /* ensure visibility for hold_devi */

1692                        if ((rv = detach_node(dip, flag)) == DDI_SUCCESS)
1693                                i_ddi_set_node_state(dip, DS_PROBED);

1695                        mutex_enter(&(DEVI(dip)->devi_lock));
1696                        DEVI_CLR_DETACHING(dip);
1697                        mutex_exit(&(DEVI(dip)->devi_lock));

1699                        atomic_dec_ulong(&devinfo_attach_detach);
1699                        atomic_add_long(&devinfo_attach_detach, -1);
1700                        break;
1701                case DS_READY:
1702                        if ((rv = predetach_node(dip, flag)) == DDI_SUCCESS)
1703                                i_ddi_set_node_state(dip, DS_ATTACHED);
1704                        break;
1705                default:
1706                        ASSERT("unknown devinfo state");
1707                }
1708        }
1709        da_log_enter(dip);
1710        return (rv);
1711 }
```

_____unchanged_portion_omitted_

****
17409 Mon Jul 28 07:44:47 2014
new/usr/src/uts/common/os/devpolicy.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
 180 void
 181 dphold(devplcy_t *dp)
 182 {
 183         ASSERT(dp->dp_ref != 0xdeadbeef && dp->dp_ref != 0);
 184         atomic_inc_32(&dp->dp_ref);
 184         atomic_add_32(&dp->dp_ref, 1);
 185 }

 187 void
 188 dpfree(devplcy_t *dp)
 189 {
 190         ASSERT(dp->dp_ref != 0xdeadbeef && dp->dp_ref != 0);
 191         if (atomic_dec_32_nv(&dp->dp_ref) == 0)
 191         if (atomic_add_32_nv(&dp->dp_ref, -1) == 0)
 192                 kmem_free(dp, sizeof (*dp));
 193 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   94995 Mon Jul 28 07:44:47 2014
new/usr/src/uts/common/os/driver_lyr.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 354 static struct ldi_handle *
 355 handle_alloc(vnode_t *vp, struct ldi_ident *ident)
 356 {
 357         struct ldi_handle       *lhp, **lhpp, *retlhp;
 358         uint_t                  index;

 360         ASSERT((vp != NULL) && (ident != NULL));

 362         /* allocate a new handle in case we need it */
 363         lhp = kmem_zalloc(sizeof (*lhp), KM_SLEEP);

 365         /* search the hash for a matching handle */
 366         index = LH_HASH(vp);
 367         mutex_enter(&ldi_handle_hash_lock[index]);
 368         lhpp = handle_find_ref_nolock(vp, ident);

 370         if (*lhpp != NULL) {
 371                 /* we found a handle in the hash */
 372                 (*lhpp)->lh_ref++;
 373                 retlhp = *lhpp;
 374                 mutex_exit(&ldi_handle_hash_lock[index]);

 376                 LDI_ALLOCFREE((CE_WARN, "ldi handle alloc: dup "
 377                     "lh=0x%p, ident=0x%p, vp=0x%p, drv=%s, minor=0x%x",
 378                     (void *)retlhp, (void *)ident, (void *)vp,
 379                     mod_major_to_name(getmajor(vp->v_rdev)),
 380                     getminor(vp->v_rdev)));

 382                 kmem_free(lhp, sizeof (struct ldi_handle));
 383                 return (retlhp);
 384         }

 386         /* initialize the new handle */
 387         lhp->lh_ref = 1;
 388         lhp->lh_vp = vp;
 389         lhp->lh_ident = ident;
 390 #ifdef  LDI_OBSOLETE_EVENT
 391         mutex_init(lhp->lh_lock, NULL, MUTEX_DEFAULT, NULL);
 392 #endif

 394         /* set the device type for this handle */
 395         lhp->lh_type = 0;
 396         if (vp->v_stream) {
 397                 ASSERT(vp->v_type == VCHR);
 398                 lhp->lh_type |= LH_STREAM;
 399         } else {
 400                 lhp->lh_type |= LH_CBDEV;
 401         }

 403         /* get holds on other objects */
 404         ident_hold(ident);
 405         ASSERT(vp->v_count >= 1);
 406         VN_HOLD(vp);

 408         /* add it to the handle hash */
 409         lhp->lh_next = ldi_handle_hash[index];
 410         ldi_handle_hash[index] = lhp;
 411         atomic_inc_ulong(&ldi_handle_hash_count);
 411         atomic_add_long(&ldi_handle_hash_count, 1);
```



```
 413         LDI_ALLOCFREE((CE_WARN, "ldi handle alloc: new "
 414             "lh=0x%p, ident=0x%p, vp=0x%p, drv=%s, minor=0x%x",
 415             (void *)lhp, (void *)ident, (void *)vp,
 416             mod_major_to_name(getmajor(vp->v_rdev)),
 417             getminor(vp->v_rdev)));

 419         mutex_exit(&ldi_handle_hash_lock[index]);
 420         return (lhp);
 421 }

 423 static void
 424 handle_release(struct ldi_handle *lhp)
 425 {
 426         struct ldi_handle       **lhpp;
 427         uint_t                  index;

 429         ASSERT(lhp != NULL);

 431         index = LH_HASH(lhp->lh_vp);
 432         mutex_enter(&ldi_handle_hash_lock[index]);

 434         LDI_ALLOCFREE((CE_WARN, "ldi handle release: "
 435             "lh=0x%p, ident=0x%p, vp=0x%p, drv=%s, minor=0x%x",
 436             (void *)lhp, (void *)lhp->lh_ident, (void *)lhp->lh_vp,
 437             mod_major_to_name(getmajor(lhp->lh_vp->v_rdev)),
 438             getminor(lhp->lh_vp->v_rdev)));

 440         ASSERT(lhp->lh_ref > 0);
 441         if (--lhp->lh_ref > 0) {
 442                 /* there are more references to this handle */
 443                 mutex_exit(&ldi_handle_hash_lock[index]);
 444                 return;
 445         }

 447         /* this was the last reference/open for this handle.  free it. */
 448         lhpp = handle_find_ref_nolock(lhp->lh_vp, lhp->lh_ident);
 449         ASSERT((lhpp != NULL) && (*lhpp != NULL));
 450         *lhpp = lhp->lh_next;
 451         atomic_dec_ulong(&ldi_handle_hash_count);
 451         atomic_add_long(&ldi_handle_hash_count, -1);
 452         mutex_exit(&ldi_handle_hash_lock[index]);

 454         VN_RELE(lhp->lh_vp);
 455         ident_release(lhp->lh_ident);
 456 #ifdef  LDI_OBSOLETE_EVENT
 457         mutex_destroy(lhp->lh_lock);
 458 #endif
 459         kmem_free(lhp, sizeof (struct ldi_handle));
 460 }
______unchanged_portion_omitted_
```

```
**********************************************************
   36391 Mon Jul 28 07:44:48 2014
new/usr/src/uts/common/os/errorq.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 514 /*
 515  * Dispatch a new error into the queue for later processing.  The specified
 516  * data buffer is copied into a preallocated queue element.  If 'len' is
 517  * smaller than the queue element size, the remainder of the queue element is
 518  * filled with zeroes.  This function may be called from any context subject
 519  * to the Platform Considerations described above.
 520  */
 521 void
 522 errorq_dispatch(errorq_t *eqp, const void *data, size_t len, uint_t flag)
 523 {
 524         errorq_elem_t *eep, *old;

 526         if (eqp == NULL || !(eqp->eq_flags & ERRORQ_ACTIVE)) {
 527                 atomic_inc_64(&errorq_lost);
 527                 atomic_add_64(&errorq_lost, 1);
 528                 return; /* drop error if queue is uninitialized or disabled */
 529         }

 531         for (;;) {
 532                 int i, rval;

 534                 if ((i = errorq_availbit(eqp->eq_bitmap, eqp->eq_qlen,
 535                     eqp->eq_rotor)) == -1) {
 536                         atomic_inc_64(&eqp->eq_kstat.eqk_dropped.value.ui64);
 536                         atomic_add_64(&eqp->eq_kstat.eqk_dropped.value.ui64, 1);
 537                         return;
 538                 }
 539                 BT_ATOMIC_SET_EXCL(eqp->eq_bitmap, i, rval);
 540                 if (rval == 0) {
 541                         eqp->eq_rotor = i;
 542                         eep = &eqp->eq_elems[i];
 543                         break;
 544                 }
 545         }

 547         ASSERT(len <= eqp->eq_size);
 548         bcopy(data, eep->eqe_data, MIN(eqp->eq_size, len));

 550         if (len < eqp->eq_size)
 551                 bzero((caddr_t)eep->eqe_data + len, eqp->eq_size - len);

 553         for (;;) {
 554                 old = eqp->eq_pend;
 555                 eep->eqe_prev = old;
 556                 membar_producer();

 558                 if (atomic_cas_ptr(&eqp->eq_pend, old, eep) == old)
 559                         break;
 560         }

 562         atomic_inc_64(&eqp->eq_kstat.eqk_dispatched.value.ui64);
 562         atomic_add_64(&eqp->eq_kstat.eqk_dispatched.value.ui64, 1);

 564         if (flag == ERRORQ_ASYNC && eqp->eq_id != NULL)
 565                 ddi_trigger_softintr(eqp->eq_id);
 566 }
_____unchanged_portion_omitted_

 857 /*
```



```
 858  * Reserve an error queue element for later processing and dispatching.  The
 859  * element is returned to the caller who may add error-specific data to
 860  * element.  The element is retured to the free pool when either
 861  * errorq_commit() is called and the element asynchronously processed
 862  * or immediately when errorq_cancel() is called.
 863  */
 864 errorq_elem_t *
 865 errorq_reserve(errorq_t *eqp)
 866 {
 867         errorq_elem_t *eqep;

 869         if (eqp == NULL || !(eqp->eq_flags & ERRORQ_ACTIVE)) {
 870                 atomic_inc_64(&errorq_lost);
 870                 atomic_add_64(&errorq_lost, 1);
 871                 return (NULL);
 872         }

 874         for (;;) {
 875                 int i, rval;

 877                 if ((i = errorq_availbit(eqp->eq_bitmap, eqp->eq_qlen,
 878                     eqp->eq_rotor)) == -1) {
 879                         atomic_inc_64(&eqp->eq_kstat.eqk_dropped.value.ui64);
 879                         atomic_add_64(&eqp->eq_kstat.eqk_dropped.value.ui64, 1);
 880                         return (NULL);
 881                 }
 882                 BT_ATOMIC_SET_EXCL(eqp->eq_bitmap, i, rval);
 883                 if (rval == 0) {
 884                         eqp->eq_rotor = i;
 885                         eqep = &eqp->eq_elems[i];
 886                         break;
 887                 }
 888         }

 890         if (eqp->eq_flags & ERRORQ_NVLIST) {
 891                 errorq_nvelem_t *eqnp = eqep->eqe_data;
 892                 nv_alloc_reset(eqnp->eqn_nva);
 893                 eqnp->eqn_nvl = fm_nvlist_create(eqnp->eqn_nva);
 894         }

 896         atomic_inc_64(&eqp->eq_kstat.eqk_reserved.value.ui64);
 896         atomic_add_64(&eqp->eq_kstat.eqk_reserved.value.ui64, 1);
 897         return (eqep);
 898 }

 900 /*
 901  * Commit an errorq element (eqep) for dispatching.
 902  * This function may be called from any context subject
 903  * to the Platform Considerations described above.
 904  */
 905 void
 906 errorq_commit(errorq_t *eqp, errorq_elem_t *eqep, uint_t flag)
 907 {
 908         errorq_elem_t *old;

 910         if (eqep == NULL || !(eqp->eq_flags & ERRORQ_ACTIVE)) {
 911                 atomic_inc_64(&eqp->eq_kstat.eqk_commit_fail.value.ui64);
 911                 atomic_add_64(&eqp->eq_kstat.eqk_commit_fail.value.ui64, 1);
 912                 return;
 913         }

 915         for (;;) {
 916                 old = eqp->eq_pend;
 917                 eqep->eqe_prev = old;
 918                 membar_producer();
```

```
 920                 if (atomic_cas_ptr(&eqp->eq_pend, old, eqep) == old)
 921                         break;
 922         }

 924         atomic_inc_64(&eqp->eq_kstat.eqk_committed.value.ui64);
 924         atomic_add_64(&eqp->eq_kstat.eqk_committed.value.ui64, 1);

 926         if (flag == ERRORQ_ASYNC && eqp->eq_id != NULL)
 927                 ddi_trigger_softintr(eqp->eq_id);
 928 }

 930 /*
 931  * Cancel an errorq element reservation by returning the specified element
 932  * to the free pool.  Duplicate or invalid frees are not supported.
 933  */
 934 void
 935 errorq_cancel(errorq_t *eqp, errorq_elem_t *eqep)
 936 {
 937         if (eqep == NULL || !(eqp->eq_flags & ERRORQ_ACTIVE))
 938                 return;

 940         BT_ATOMIC_CLEAR(eqp->eq_bitmap, eqep - eqp->eq_elems);

 942         atomic_inc_64(&eqp->eq_kstat.eqk_cancelled.value.ui64);
 942         atomic_add_64(&eqp->eq_kstat.eqk_cancelled.value.ui64, 1);
 943 }
```

*unchanged_portion_omitted_*

****
59742 Mon Jul 28 07:44:48 2014
new/usr/src/uts/common/os/evchannels.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
 372 /*
 373  * Frees evch_gevent_t structure including the payload, if the reference count
 374  * drops to or below zero. Below zero happens when the event is freed
 375  * without beeing queued into a queue.
 376  */
 377 static void
 378 evch_gevent_free(evch_gevent_t *evp)
 379 {
 380         int32_t refcnt;

 382         refcnt = (int32_t)atomic_dec_32_nv(&evp->ge_refcount);
 382         refcnt = (int32_t)atomic_add_32_nv(&evp->ge_refcount, -1);
 383         if (refcnt <= 0) {
 384                 if (evp->ge_destruct != NULL) {
 385                         evp->ge_destruct((void *)&(evp->ge_payload),
 386                             evp->ge_dstcookie);
 387                 }
 388                 kmem_free(evp, evp->ge_size);
 389         }
 390 }
```
_____***unchanged_portion_omitted_***

```
 628 /*
 629  * Publish an event. Returns 0 on success and -1 if memory alloc failed.
 630  */
 631 static int
 632 evch_evq_pub(evch_eventq_t *eqp, void *ev, int flags)
 633 {
 634         size_t size;
 635         evch_qelem_t    *qep;
 636         evch_gevent_t   *evp = GEVENT(ev);

 638         size = sizeof (evch_qelem_t);
 639         if (flags & EVCH_TRYHARD) {
 640                 qep = kmem_alloc_tryhard(size, &size, KM_NOSLEEP);
 641         } else {
 642                 qep = kmem_alloc(size, flags & EVCH_NOSLEEP ?
 643                     KM_NOSLEEP : KM_SLEEP);
 644         }
 645         if (qep == NULL) {
 646                 return (-1);
 647         }
 648         qep->q_objref = (void *)evp;
 649         qep->q_objsize = size;
 650         atomic_inc_32(&evp->ge_refcount);
 650         atomic_add_32(&evp->ge_refcount, 1);
 651         mutex_enter(&eqp->eq_queuemx);
 652         evch_q_in(&eqp->eq_eventq, qep);

 654         /* Wakeup delivery thread */
 655         cv_signal(&eqp->eq_thrsleepcv);
 656         mutex_exit(&eqp->eq_queuemx);
 657         return (0);
 658 }
```
_____***unchanged_portion_omitted_***

```
**********************************************************
   32372 Mon Jul 28 07:44:48 2014
new/usr/src/uts/common/os/exit.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 321 /*
 322  * Return value:
 323  *   1 - exitlwps() failed, call (or continue) lwp_exit()
 324  *   0 - restarting init.  Return through system call path
 325  */
 326 int
 327 proc_exit(int why, int what)
 328 {
 329         kthread_t *t = curthread;
 330         klwp_t *lwp = ttolwp(t);
 331         proc_t *p = ttoproc(t);
 332         zone_t *z = p->p_zone;
 333         timeout_id_t tmp_id;
 334         int rv;
 335         proc_t *q;
 336         task_t *tk;
 337         vnode_t *exec_vp, *execdir_vp, *cdir, *rdir;
 338         sigqueue_t *sqp;
 339         lwpdir_t *lwpdir;
 340         uint_t lwpdir_sz;
 341         tidhash_t *tidhash;
 342         uint_t tidhash_sz;
 343         ret_tidhash_t *ret_tidhash;
 344         refstr_t *cwd;
 345         hrtime_t hrutime, hrstime;
 346         int evaporate;

 348         /*
 349          * Stop and discard the process's lwps except for the current one,
 350          * unless some other lwp beat us to it.  If exitlwps() fails then
 351          * return and the calling lwp will call (or continue in) lwp_exit().
 352          */
 353         proc_is_exiting(p);
 354         if (exitlwps(0) != 0)
 355                 return (1);

 357         mutex_enter(&p->p_lock);
 358         if (p->p_ttime > 0) {
 359                 /*
 360                  * Account any remaining ticks charged to this process
 361                  * on its way out.
 362                  */
 363                 (void) task_cpu_time_incr(p->p_task, p->p_ttime);
 364                 p->p_ttime = 0;
 365         }
 366         mutex_exit(&p->p_lock);

 368         DTRACE_PROC(lwp__exit);
 369         DTRACE_PROC1(exit, int, why);

 371         /*
 372          * Will perform any brand specific proc exit processing, since this
 373          * is always the last lwp, will also perform lwp_exit and free brand
 374          * data
 375          */
 376         if (PROC_IS_BRANDED(p)) {
 377                 lwp_detach_brand_hdlrs(lwp);
 378                 brand_clearbrand(p, B_FALSE);
 379         }
```



```
 381         /*
 382          * Don't let init exit unless zone_start_init() failed its exec, or
 383          * we are shutting down the zone or the machine.
 384          *
 385          * Since we are single threaded, we don't need to lock the
 386          * following accesses to zone_proc_initpid.
 387          */
 388         if (p->p_pid == z->zone_proc_initpid) {
 389                 if (z->zone_boot_err == 0 &&
 390                     zone_status_get(z) < ZONE_IS_SHUTTING_DOWN &&
 391                     zone_status_get(global_zone) < ZONE_IS_SHUTTING_DOWN &&
 392                     z->zone_restart_init == B_TRUE &&
 393                     restart_init(what, why) == 0)
 394                         return (0);
 395                 /*
 396                  * Since we didn't or couldn't restart init, we clear
 397                  * the zone's init state and proceed with exit
 398                  * processing.
 399                  */
 400                 z->zone_proc_initpid = -1;
 401         }

 403         lwp_pcb_exit();

 405         /*
 406          * Allocate a sigqueue now, before we grab locks.
 407          * It will be given to sigcld(), below.
 408          * Special case:  If we will be making the process disappear
 409          * without a trace because it is either:
 410          *      * an exiting SSYS process, or
 411          *      * a posix_spawn() vfork child who requests it,
 412          * we don't bother to allocate a useless sigqueue.
 413          */
 414         evaporate = (p->p_flag & SSYS) || ((p->p_flag & SVFORK) &&
 415             why == CLD_EXITED && what == _EVAPORATE);
 416         if (!evaporate)
 417                 sqp = kmem_zalloc(sizeof (sigqueue_t), KM_SLEEP);

 419         /*
 420          * revoke any doors created by the process.
 421          */
 422         if (p->p_door_list)
 423                 door_exit();

 425         /*
 426          * Release schedctl data structures.
 427          */
 428         if (p->p_pagep)
 429                 schedctl_proc_cleanup();

 431         /*
 432          * make sure all pending kaio has completed.
 433          */
 434         if (p->p_aio)
 435                 aio_cleanup_exit();

 437         /*
 438          * discard the lwpchan cache.
 439          */
 440         if (p->p_lcp != NULL)
 441                 lwpchan_destroy_cache(0);

 443         /*
 444          * Clean up any DTrace helper actions or probes for the process.
 445          */
```

```
446         if (p->p_dtrace_helpers != NULL) {
447                 ASSERT(dtrace_helpers_cleanup != NULL);
448                 (*dtrace_helpers_cleanup)();
449         }
450
451         /* untimeout the realtime timers */
452         if (p->p_itimer != NULL)
453                 timer_exit();
454
455         if ((tmp_id = p->p_alarmid) != 0) {
456                 p->p_alarmid = 0;
457                 (void) untimeout(tmp_id);
458         }
459
460         /*
461          * Remove any fpollinfo_t's for this (last) thread from our file
462          * descriptors so closeall() can ASSERT() that they're all gone.
463          */
464         pollcleanup();
465
466         if (p->p_rprof_cyclic != CYCLIC_NONE) {
467                 mutex_enter(&cpu_lock);
468                 cyclic_remove(p->p_rprof_cyclic);
469                 mutex_exit(&cpu_lock);
470         }
471
472         mutex_enter(&p->p_lock);
473
474         /*
475          * Clean up any DTrace probes associated with this process.
476          */
477         if (p->p_dtrace_probes) {
478                 ASSERT(dtrace_fasttrap_exit_ptr != NULL);
479                 dtrace_fasttrap_exit_ptr(p);
480         }
481
482         while ((tmp_id = p->p_itimerid) != 0) {
483                 p->p_itimerid = 0;
484                 mutex_exit(&p->p_lock);
485                 (void) untimeout(tmp_id);
486                 mutex_enter(&p->p_lock);
487         }
488
489         lwp_cleanup();
490
491         /*
492          * We are about to exit; prevent our resource associations from
493          * being changed.
494          */
495         pool_barrier_enter();
496
497         /*
498          * Block the process against /proc now that we have really
499          * acquired p->p_lock (to manipulate p_tlist at least).
500          */
501         prbarrier(p);
502
503         sigfillset(&p->p_ignore);
504         sigemptyset(&p->p_siginfo);
505         sigemptyset(&p->p_sig);
506         sigemptyset(&p->p_extsig);
507         sigemptyset(&t->t_sig);
508         sigemptyset(&t->t_extsig);
509         sigemptyset(&p->p_sigmask);
510         sigdelq(p, t, 0);
511         lwp->lwp_cursig = 0;
```



```
512         lwp->lwp_extsig = 0;
513         p->p_flag &= ~(SKILLED | SEXTKILLED);
514         if (lwp->lwp_curinfo) {
515                 siginfofree(lwp->lwp_curinfo);
516                 lwp->lwp_curinfo = NULL;
517         }
518
519         t->t_proc_flag |= TP_LWPEXIT;
520         ASSERT(p->p_lwpcnt == 1 && p->p_zombcnt == 0);
521         prlwpexit(t);           /* notify /proc */
522         lwp_hash_out(p, t->t_tid);
523         prexit(p);
524
525         p->p_lwpcnt = 0;
526         p->p_tlist = NULL;
527         sigqfree(p);
528         term_mstate(t);
529         p->p_mterm = gethrtime();
530
531         exec_vp = p->p_exec;
532         execdir_vp = p->p_execdir;
533         p->p_exec = NULLVP;
534         p->p_execdir = NULLVP;
535         mutex_exit(&p->p_lock);
536
537         pr_free_watched_pages(p);
538
539         closeall(P_FINFO(p));
540
541         /* Free the controlling tty.  (freectty() always assumes curproc.) */
542         ASSERT(p == curproc);
543         (void) freectty(B_TRUE);
544
545 #if defined(__sparc)
546         if (p->p_utraps != NULL)
547                 utrap_free(p);
548 #endif
549         if (p->p_semacct)                       /* IPC semaphore exit */
550                 semexit(p);
551         rv = wstat(why, what);
552
553         acct(rv & 0xff);
554         exacct_commit_proc(p, rv);
555
556         /*
557          * Release any resources associated with C2 auditing
558          */
559         if (AU_AUDITING()) {
560                 /*
561                  * audit exit system call
562                  */
563                 audit_exit(why, what);
564         }
565
566         /*
567          * Free address space.
568          */
569         relvm();
570
571         if (exec_vp) {
572                 /*
573                  * Close this executable which has been opened when the process
574                  * was created by getproc().
575                  */
576                 (void) VOP_CLOSE(exec_vp, FREAD, 1, (offset_t)0, CRED(), NULL);
577                 VN_RELE(exec_vp);
```

```
578         }
579         if (execdir_vp)
580                 VN_RELE(execdir_vp);

582         /*
583          * Release held contracts.
584          */
585         contract_exit(p);

587         /*
588          * Depart our encapsulating process contract.
589          */
590         if ((p->p_flag & SSYS) == 0) {
591                 ASSERT(p->p_ct_process);
592                 contract_process_exit(p->p_ct_process, p, rv);
593         }

595         /*
596          * Remove pool association, and block if requested by pool_do_bind.
597          */
598         mutex_enter(&p->p_lock);
599         ASSERT(p->p_pool->pool_ref > 0);
600         atomic_dec_32(&p->p_pool->pool_ref);
600         atomic_add_32(&p->p_pool->pool_ref, -1);
601         p->p_pool = pool_default;
602         /*
603          * Now that our address space has been freed and all other threads
604          * in this process have exited, set the PEXITED pool flag.  This
605          * tells the pools subsystems to ignore this process if it was
606          * requested to rebind this process to a new pool.
607          */
608         p->p_poolflag |= PEXITED;
609         pool_barrier_exit();
610         mutex_exit(&p->p_lock);

612         mutex_enter(&pidlock);

614         /*
615          * Delete this process from the newstate list of its parent. We
616          * will put it in the right place in the sigcld in the end.
617          */
618         delete_ns(p->p_parent, p);

620         /*
621          * Reassign the orphans to the next of kin.
622          * Don't rearrange init's orphanage.
623          */
624         if ((q = p->p_orphan) != NULL && p != proc_init) {

626                 proc_t *nokp = p->p_nextofkin;

628                 for (;;) {
629                         q->p_nextofkin = nokp;
630                         if (q->p_nextorph == NULL)
631                                 break;
632                         q = q->p_nextorph;
633                 }
634                 q->p_nextorph = nokp->p_orphan;
635                 nokp->p_orphan = p->p_orphan;
636                 p->p_orphan = NULL;
637         }

639         /*
640          * Reassign the children to init.
641          * Don't try to assign init's children to init.
642          */
```



```
643         if ((q = p->p_child) != NULL && p != proc_init) {
644                 struct proc     *np;
645                 struct proc     *initp = proc_init;
646                 boolean_t       setzonetop = B_FALSE;

648                 if (!INGLOBALZONE(curproc))
649                         setzonetop = B_TRUE;

651                 pgdetach(p);

653                 do {
654                         np = q->p_sibling;
655                         /*
656                          * Delete it from its current parent new state
657                          * list and add it to init new state list
658                          */
659                         delete_ns(q->p_parent, q);

661                         q->p_ppid = 1;
662                         q->p_pidflag &= ~(CLDNOSIGCHLD | CLDWAITPID);
663                         if (setzonetop) {
664                                 mutex_enter(&q->p_lock);
665                                 q->p_flag |= SZONETOP;
666                                 mutex_exit(&q->p_lock);
667                         }
668                         q->p_parent = initp;

670                         /*
671                          * Since q will be the first child,
672                          * it will not have a previous sibling.
673                          */
674                         q->p_psibling = NULL;
675                         if (initp->p_child) {
676                                 initp->p_child->p_psibling = q;
677                         }
678                         q->p_sibling = initp->p_child;
679                         initp->p_child = q;
680                         if (q->p_proc_flag & P_PR_PTRACE) {
681                                 mutex_enter(&q->p_lock);
682                                 sigtoproc(q, NULL, SIGKILL);
683                                 mutex_exit(&q->p_lock);
684                         }
685                         /*
686                          * sigcld() will add the child to parents
687                          * newstate list.
688                          */
689                         if (q->p_stat == SZOMB)
690                                 sigcld(q, NULL);
691                 } while ((q = np) != NULL);

693                 p->p_child = NULL;
694                 ASSERT(p->p_child_ns == NULL);
695         }

697         TRACE_1(TR_FAC_PROC, TR_PROC_EXIT, "proc_exit: %p", p);

699         mutex_enter(&p->p_lock);
700         CL_EXIT(curthread); /* tell the scheduler that curthread is exiting */

702         /*
703          * Have our task accummulate our resource usage data before they
704          * become contaminated by p_cacct etc., and before we renounce
705          * membership of the task.
706          *
707          * We do this regardless of whether or not task accounting is active.
708          * This is to avoid having nonsense data reported for this task if
```

```
709          * task accounting is subsequently enabled. The overhead is minimal;
710          * by this point, this process has accounted for the usage of all its
711          * LWPs. We nonetheless do the work here, and under the protection of
712          * pidlock, so that the movement of the process's usage to the task
713          * happens at the same time as the removal of the process from the
714          * task, from the point of view of exacct_snapshot_task_usage().
715          */
716         exacct_update_task_mstate(p);

718         hrutime = mstate_aggr_state(p, LMS_USER);
719         hrstime = mstate_aggr_state(p, LMS_SYSTEM);
720         p->p_utime = (clock_t)NSEC_TO_TICK(hrutime) + p->p_cutime;
721         p->p_stime = (clock_t)NSEC_TO_TICK(hrstime) + p->p_cstime;

723         p->p_acct[LMS_USER]     += p->p_cacct[LMS_USER];
724         p->p_acct[LMS_SYSTEM]   += p->p_cacct[LMS_SYSTEM];
725         p->p_acct[LMS_TRAP]     += p->p_cacct[LMS_TRAP];
726         p->p_acct[LMS_TFAULT]   += p->p_cacct[LMS_TFAULT];
727         p->p_acct[LMS_DFAULT]   += p->p_cacct[LMS_DFAULT];
728         p->p_acct[LMS_KFAULT]   += p->p_cacct[LMS_KFAULT];
729         p->p_acct[LMS_USER_LOCK] += p->p_cacct[LMS_USER_LOCK];
730         p->p_acct[LMS_SLEEP]    += p->p_cacct[LMS_SLEEP];
731         p->p_acct[LMS_WAIT_CPU] += p->p_cacct[LMS_WAIT_CPU];
732         p->p_acct[LMS_STOPPED]  += p->p_cacct[LMS_STOPPED];

734         p->p_ru.minflt  += p->p_cru.minflt;
735         p->p_ru.majflt  += p->p_cru.majflt;
736         p->p_ru.nswap   += p->p_cru.nswap;
737         p->p_ru.inblock += p->p_cru.inblock;
738         p->p_ru.oublock += p->p_cru.oublock;
739         p->p_ru.msgsnd  += p->p_cru.msgsnd;
740         p->p_ru.msgrcv  += p->p_cru.msgrcv;
741         p->p_ru.nsignals += p->p_cru.nsignals;
742         p->p_ru.nvcsw   += p->p_cru.nvcsw;
743         p->p_ru.nivcsw  += p->p_cru.nivcsw;
744         p->p_ru.sysc    += p->p_cru.sysc;
745         p->p_ru.ioch    += p->p_cru.ioch;

747         p->p_stat = SZOMB;
748         p->p_proc_flag &= ~P_PR_PTRACE;
749         p->p_wdata = what;
750         p->p_wcode = (char)why;

752         cdir = PTOU(p)->u_cdir;
753         rdir = PTOU(p)->u_rdir;
754         cwd = PTOU(p)->u_cwd;

756         ASSERT(cdir != NULL || p->p_parent == &p0);

758         /*
759          * Release resource controls, as they are no longer enforceable.
760          */
761         rctl_set_free(p->p_rctls);

763         /*
764          * Decrement tk_nlwps counter for our task.max-lwps resource control.
765          * An extended accounting record, if that facility is active, is
766          * scheduled to be written.  We cannot give up task and project
767          * membership at this point because that would allow zombies to escape
768          * from the max-processes resource controls.  Zombies stay in their
769          * current task and project until the process table slot is released
770          * in freeproc().
771          */
772         tk = p->p_task;

774         mutex_enter(&p->p_zone->zone_nlwps_lock);
```



```
775         tk->tk_nlwps--;
776         tk->tk_proj->kpj_nlwps--;
777         p->p_zone->zone_nlwps--;
778         mutex_exit(&p->p_zone->zone_nlwps_lock);

780         /*
781          * Clear the lwp directory and the lwpid hash table
782          * now that /proc can't bother us any more.
783          * We free the memory below, after dropping p->p_lock.
784          */
785         lwpdir = p->p_lwpdir;
786         lwpdir_sz = p->p_lwpdir_sz;
787         tidhash = p->p_tidhash;
788         tidhash_sz = p->p_tidhash_sz;
789         ret_tidhash = p->p_ret_tidhash;
790         p->p_lwpdir = NULL;
791         p->p_lwpfree = NULL;
792         p->p_lwpdir_sz = 0;
793         p->p_tidhash = NULL;
794         p->p_tidhash_sz = 0;
795         p->p_ret_tidhash = NULL;

797         /*
798          * If the process has context ops installed, call the exit routine
799          * on behalf of this last remaining thread. Normally exitpctx() is
800          * called during thread_exit() or lwp_exit(), but because this is the
801          * last thread in the process, we must call it here. By the time
802          * thread_exit() is called (below), the association with the relevant
803          * process has been lost.
804          *
805          * We also free the context here.
806          */
807         if (p->p_pctx) {
808                 kpreempt_disable();
809                 exitpctx(p);
810                 kpreempt_enable();

812                 freepctx(p, 0);
813         }

815         /*
816          * curthread's proc pointer is changed to point to the 'sched'
817          * process for the corresponding zone, except in the case when
818          * the exiting process is in fact a zsched instance, in which
819          * case the proc pointer is set to p0.  We do so, so that the
820          * process still points at the right zone when we call the VN_RELE()
821          * below.
822          *
823          * This is because curthread's original proc pointer can be freed as
824          * soon as the child sends a SIGCLD to its parent.  We use zsched so
825          * that for user processes, even in the final moments of death, the
826          * process is still associated with its zone.
827          */
828         if (p != t->t_procp->p_zone->zone_zsched)
829                 t->t_procp = t->t_procp->p_zone->zone_zsched;
830         else
831                 t->t_procp = &p0;

833         mutex_exit(&p->p_lock);
834         if (!evaporate) {
835                 p->p_pidflag &= ~CLDPEND;
836                 sigcld(p, sqp);
837         } else {
838                 /*
839                  * Do what sigcld() would do if the disposition
840                  * of the SIGCHLD signal were set to be ignored.
```

```
841                  */
842                 cv_broadcast(&p->p_srwchan_cv);
843                 freeproc(p);
844         }
845         mutex_exit(&pidlock);

847         /*
848          * We don't release u_cdir and u_rdir until SZOMB is set.
849          * This protects us against dofusers().
850          */
851         if (cdir)
852                 VN_RELE(cdir);
853         if (rdir)
854                 VN_RELE(rdir);
855         if (cwd)
856                 refstr_rele(cwd);

858         /*
859          * task_rele() may ultimately cause the zone to go away (or
860          * may cause the last user process in a zone to go away, which
861          * signals zsched to go away).  So prior to this call, we must
862          * no longer point at zsched.
863          */
864         t->t_procp = &p0;

866         kmem_free(lwpdir, lwpdir_sz * sizeof (lwpdir_t));
867         kmem_free(tidhash, tidhash_sz * sizeof (tidhash_t));
868         while (ret_tidhash != NULL) {
869                 ret_tidhash_t *next = ret_tidhash->rth_next;
870                 kmem_free(ret_tidhash->rth_tidhash,
871                     ret_tidhash->rth_tidhash_sz * sizeof (tidhash_t));
872                 kmem_free(ret_tidhash, sizeof (*ret_tidhash));
873                 ret_tidhash = next;
874         }

876         thread_exit();
877         /* NOTREACHED */
878 }
```

unchanged_portion_omitted_

```
**********************************************************
   46741 Mon Jul 28 07:44:48 2014
new/usr/src/uts/common/os/fio.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 1989, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012, Joyent Inc. All rights reserved.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*      All Rights Reserved */

#include <sys/types.h>
#include <sys/sysmacros.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/errno.h>
#include <sys/signal.h>
#include <sys/cred.h>
#include <sys/user.h>
#include <sys/conf.h>
#include <sys/vfs.h>
#include <sys/vnode.h>
#include <sys/pathname.h>
#include <sys/file.h>
#include <sys/proc.h>
#include <sys/var.h>
#include <sys/cpuvar.h>
#include <sys/open.h>
#include <sys/cmn_err.h>
#include <sys/priocntl.h>
#include <sys/procset.h>
#include <sys/prsystm.h>
#include <sys/debug.h>
#include <sys/kmem.h>
#include <sys/atomic.h>
#include <sys/fcntl.h>
#include <sys/poll.h>
#include <sys/rctl.h>
#include <sys/port_impl.h>
#include <sys/dtrace.h>

#include <c2/audit.h>
#include <sys/nbmlock.h>
```



```
#ifdef DEBUG

static uint32_t afd_maxfd;      /* # of entries in maximum allocated array */
static uint32_t afd_alloc;      /* count of kmem_alloc()s */
static uint32_t afd_free;       /* count of kmem_free()s */
static uint32_t afd_wait;       /* count of waits on non-zero ref count */
#define MAXFD(x)        (afd_maxfd = ((afd_maxfd >= (x))? afd_maxfd : (x)))
#define COUNT(x)        atomic_inc_32(&x)
#define COUNT(x)        atomic_add_32(&x, 1)

#else   /* DEBUG */

#define MAXFD(x)
#define COUNT(x)

#endif  /* DEBUG */

kmem_cache_t *file_cache;

static void port_close_fd(portfd_t *);

/*
 * File descriptor allocation.
 *
 * fd_find(fip, minfd) finds the first available descriptor >= minfd.
 * The most common case is open(2), in which minfd = 0, but we must also
 * support fcntl(fd, F_DUPFD, minfd).
 *
 * The algorithm is as follows: we keep all file descriptors in an infix
 * binary tree in which each node records the number of descriptors
 * allocated in its right subtree, including itself.  Starting at minfd,
 * we ascend the tree until we find a non-fully allocated right subtree.
 * We then descend that subtree in a binary search for the smallest fd.
 * Finally, we ascend the tree again to increment the allocation count
 * of every subtree containing the newly-allocated fd.  Freeing an fd
 * requires only the last step: we ascend the tree to decrement allocation
 * counts.  Each of these three steps (ascent to find non-full subtree,
 * descent to find lowest fd, ascent to update allocation counts) is
 * O(log n), thus the algorithm as a whole is O(log n).
 *
 * We don't implement the fd tree using the customary left/right/parent
 * pointers, but instead take advantage of the glorious mathematics of
 * full infix binary trees.  For reference, here's an illustration of the
 * logical structure of such a tree, rooted at 4 (binary 100), covering
 * the range 1-7 (binary 001-111).  Our canonical trees do not include
 * fd 0; we'll deal with that later.
 *
 *           100
 *          /   \
 *         /     \
 *       010     110
 *       / \     / \
 *     001 011 101 111
 *
 * We make the following observations, all of which are easily proven by
 * induction on the depth of the tree:
 *
 * (T1) The least-significant bit (LSB) of any node is equal to its level
 *      in the tree.  In our example, nodes 001, 011, 101 and 111 are at
 *      level 0; nodes 010 and 110 are at level 1; and node 100 is at level 2.
 *
 * (T2) The child size (CSIZE) of node N -- that is, the total number of
 *      right-branch descendants in a child of node N, including itself -- is
 *      given by clearing all but the least significant bit of N.  This
 *      follows immediately from (T1).  Applying this rule to our example, we
```

```
 127  *      see that CSIZE(100) = 100, CSIZE(x10) = 10, and CSIZE(xx1) = 1.
 128  *
 129  * (T3) The nearest left ancestor (LPARENT) of node N -- that is, the nearest
 130  *      ancestor containing node N in its right child -- is given by clearing
 131  *      the LSB of N.  For example, LPARENT(111) = 110 and LPARENT(110) = 100.
 132  *      Clearing the LSB of nodes 001, 010 or 100 yields zero, reflecting
 133  *      the fact that these are leftmost nodes.  Note that this algorithm
 134  *      automatically skips generations as necessary.  For example, the parent
 135  *      of node 101 is 110, which is a *right* ancestor (not what we want);
 136  *      but its grandparent is 100, which is a left ancestor. Clearing the LSB
 137  *      of 101 gets us to 100 directly, skipping right past the uninteresting
 138  *      generation (110).
 139  *
 140  *      Note that since LPARENT clears the LSB, whereas CSIZE clears all *but*
 141  *      the LSB, we can express LPARENT() nicely in terms of CSIZE():
 142  *
 143  *      LPARENT(N) = N - CSIZE(N)
 144  *
 145  * (T4) The nearest right ancestor (RPARENT) of node N is given by:
 146  *
 147  *      RPARENT(N) = N + CSIZE(N)
 148  *
 149  * (T5) For every interior node, the children differ from their parent by
 150  *      CSIZE(parent) / 2.  In our example, CSIZE(100) / 2 = 2 = 10 binary,
 151  *      and indeed, the children of 100 are 100 +/- 10 = 010 and 110.
 152  *
 153  * Next, we'll need a few two's-complement math tricks.  Suppose a number,
 154  * N, has the following form:
 155  *
 156  *              N = xxxx10...0
 157  *
 158  * That is, the binary representation of N consists of some string of bits,
 159  * then a 1, then all zeroes.  This amounts to nothing more than saying that
 160  * N has a least-significant bit, which is true for any N != 0.  If we look
 161  * at N and N - 1 together, we see that we can combine them in useful ways:
 162  *
 163  *                N = xxxx10...0
 164  *            N - 1 = xxxx01...1
 165  *      ------------------------
 166  *      N & (N - 1) = xxxx000000
 167  *      N | (N - 1) = xxxx111111
 168  *      N ^ (N - 1) =     111111
 169  *
 170  * In particular, this suggests several easy ways to clear all but the LSB,
 171  * which by (T2) is exactly what we need to determine CSIZE(N) = 10...0.
 172  * We'll opt for this formulation:
 173  *
 174  *      (C1) CSIZE(N) = (N - 1) ^ (N | (N - 1))
 175  *
 176  * Similarly, we have an easy way to determine LPARENT(N), which requires
 177  * that we clear the LSB of N:
 178  *
 179  *      (L1) LPARENT(N) = N & (N - 1)
 180  *
 181  * We note in the above relations that (N | (N - 1)) - N = CSIZE(N) - 1.
 182  * When combined with (T4), this yields an easy way to compute RPARENT(N):
 183  *
 184  *      (R1) RPARENT(N) = (N | (N - 1)) + 1
 185  *
 186  * Finally, to accommodate fd 0 we must adjust all of our results by +/-1 to
 187  * move the fd range from [1, 2^n) to [0, 2^n - 1).  This is straightforward,
 188  * so there's no need to belabor the algebra; the revised relations become:
 189  *
 190  *      (C1a) CSIZE(N) = N ^ (N | (N + 1))
 191  *
 192  *      (L1a) LPARENT(N) = (N & (N + 1)) - 1
```



```
 193  *
 194  *      (R1a) RPARENT(N) = N | (N + 1)
 195  *
 196  * This completes the mathematical framework.  We now have all the tools
 197  * we need to implement fd_find() and fd_reserve().
 198  *
 199  * fd_find(fip, minfd) finds the smallest available file descriptor >= minfd.
 200  * It does not actually allocate the descriptor; that's done by fd_reserve().
 201  * fd_find() proceeds in two steps:
 202  *
 203  * (1) Find the leftmost subtree that contains a descriptor >= minfd.
 204  *     We start at the right subtree rooted at minfd.  If this subtree is
 205  *     not full -- if fip->fi_list[minfd].uf_alloc != CSIZE(minfd) -- then
 206  *     step 1 is done.  Otherwise, we know that all fds in this subtree
 207  *     are taken, so we ascend to RPARENT(minfd) using (R1a).  We repeat
 208  *     this process until we either find a candidate subtree or exceed
 209  *     fip->fi_nfiles.  We use (C1a) to compute CSIZE().
 210  *
 211  * (2) Find the smallest fd in the subtree discovered by step 1.
 212  *     Starting at the root of this subtree, we descend to find the
 213  *     smallest available fd.  Since the left children have the smaller
 214  *     fds, we will descend rightward only when the left child is full.
 215  *
 216  *     We begin by comparing the number of allocated fds in the root
 217  *     to the number of allocated fds in its right child; if they differ
 218  *     by exactly CSIZE(child), we know the left subtree is full, so we
 219  *     descend right; that is, the right child becomes the search root.
 220  *     Otherwise we leave the root alone and start following the right
 221  *     child's left children.  As fortune would have it, this is very
 222  *     simple computationally: by (T5), the right child of fd is just
 223  *     fd + size, where size = CSIZE(fd) / 2.  Applying (T5) again,
 224  *     we find that the right child's left child is fd + size - (size / 2) =
 225  *     fd + (size / 2); *its* left child is fd + (size / 2) - (size / 4) =
 226  *     fd + (size / 4), and so on.  In general, fd's right child's
 227  *     leftmost nth descendant is fd + (size >> n).  Thus, to follow
 228  *     the right child's left descendants, we just halve the size in
 229  *     each iteration of the search.
 230  *
 231  *     When we descend leftward, we must keep track of the number of fds
 232  *     that were allocated in all the right subtrees we rejected, so we
 233  *     know how many of the root fd's allocations are in the remaining
 234  *     (as yet unexplored) leftmost part of its right subtree.  When we
 235  *     encounter a fully-allocated left child -- that is, when we find
 236  *     that fip->fi_list[fd].uf_alloc == ralloc + size -- we descend right
 237  *     (as described earlier), resetting ralloc to zero.
 238  *
 239  * fd_reserve(fip, fd, incr) either allocates or frees fd, depending
 240  * on whether incr is 1 or -1.  Starting at fd, fd_reserve() ascends
 241  * the leftmost ancestors (see (T3)) and updates the allocation counts.
 242  * At each step we use (L1a) to compute LPARENT(), the next left ancestor.
 243  *
 244  * flist_minsize() finds the minimal tree that still covers all
 245  * used fds; as long as the allocation count of a root node is zero, we
 246  * don't need that node or its right subtree.
 247  *
 248  * flist_nalloc() counts the number of allocated fds in the tree, by starting
 249  * at the top of the tree and summing the right-subtree allocation counts as
 250  * it descends leftwards.
 251  *
 252  * Note: we assume that flist_grow() will keep fip->fi_nfiles of the form
 253  * 2^n - 1.  This ensures that the fd trees are always full, which saves
 254  * quite a bit of boundary checking.
 255  */
 256 static int
 257 fd_find(uf_info_t *fip, int minfd)
 258 {
```

```
259         int size, ralloc, fd;

261         ASSERT(MUTEX_HELD(&fip->fi_lock));
262         ASSERT((fip->fi_nfiles & (fip->fi_nfiles + 1)) == 0);

264         for (fd = minfd; (uint_t)fd < fip->fi_nfiles; fd |= fd + 1) {
265                 size = fd ^ (fd | (fd + 1));
266                 if (fip->fi_list[fd].uf_alloc == size)
267                         continue;
268                 for (ralloc = 0; size >>= 1; size != 0; size >>= 1) {
269                         ralloc += fip->fi_list[fd + size].uf_alloc;
270                         if (fip->fi_list[fd].uf_alloc == ralloc + size) {
271                                 fd += size;
272                                 ralloc = 0;
273                         }
274                 }
275                 return (fd);
276         }
277         return (-1);
278 }
```

unchanged_portion_omitted_

```
**********************************************************
   35793 Mon Jul 28 07:44:49 2014
new/usr/src/uts/common/os/fm.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 507 /*
 508  * Post an error report (ereport) to the sysevent error channel.  The error
 509  * channel must be established with a prior call to sysevent_evc_create()
 510  * before publication may occur.
 511  */
 512 void
 513 fm_ereport_post(nvlist_t *ereport, int evc_flag)
 514 {
 515         size_t nvl_size = 0;
 516         evchan_t *error_chan;

 518         (void) nvlist_size(ereport, &nvl_size, NV_ENCODE_NATIVE);
 519         if (nvl_size > ERPT_DATA_SZ || nvl_size == 0) {
 520                 atomic_inc_64(&erpt_kstat_data.erpt_dropped.value.ui64);
 520                 atomic_add_64(&erpt_kstat_data.erpt_dropped.value.ui64, 1);
 521                 return;
 522         }

 524         if (sysevent_evc_bind(FM_ERROR_CHAN, &error_chan,
 525             EVCH_CREAT|EVCH_HOLD_PEND) != 0) {
 526                 atomic_inc_64(&erpt_kstat_data.erpt_dropped.value.ui64);
 526                 atomic_add_64(&erpt_kstat_data.erpt_dropped.value.ui64, 1);
 527                 return;
 528         }

 530         if (sysevent_evc_publish(error_chan, EC_FM, ESC_FM_ERROR,
 531             SUNW_VENDOR, FM_PUB, ereport, evc_flag) != 0) {
 532                 atomic_inc_64(&erpt_kstat_data.erpt_dropped.value.ui64);
 532                 atomic_add_64(&erpt_kstat_data.erpt_dropped.value.ui64, 1);
 533                 (void) sysevent_evc_unbind(error_chan);
 534                 return;
 535         }
 536         (void) sysevent_evc_unbind(error_chan);
 537 }
______unchanged_portion_omitted_

 781 void
 782 fm_payload_set(nvlist_t *payload, ...)
 783 {
 784         int ret;
 785         const char *name;
 786         va_list ap;

 788         va_start(ap, payload);
 789         name = va_arg(ap, char *);
 790         ret = i_fm_payload_set(payload, name, ap);
 791         va_end(ap);

 793         if (ret)
 794                 atomic_inc_64(&erpt_kstat_data.payload_set_failed.value.ui64);
 794                 atomic_add_64(
 795                     &erpt_kstat_data.payload_set_failed.value.ui64, 1);
 795 }

 797 /*
 798  * Set-up and validate the members of an ereport event according to:
 799  *
 800  *      Member name             Type            Value
 801  *      ====================================================
```



```
 802  *      class                   string          ereport
 803  *      version                 uint8_t         0
 804  *      ena                     uint64_t        <ena>
 805  *      detector                nvlist_t        <detector>
 806  *      ereport-payload         nvlist_t        <var args>
 807  *
 808  * We don't actually add a 'version' member to the payload.  Really,
 809  * the version quoted to us by our caller is that of the category 1
 810  * "ereport" event class (and we require FM_EREPORT_VERS0) but
 811  * the payload version of the actual leaf class event under construction
 812  * may be something else.  Callers should supply a version in the varargs,
 813  * or (better) we could take two version arguments - one for the
 814  * ereport category 1 classification (expect FM_EREPORT_VERS0) and one
 815  * for the leaf class.
 816  */
 817 void
 818 fm_ereport_set(nvlist_t *ereport, int version, const char *erpt_class,
 819     uint64_t ena, const nvlist_t *detector, ...)
 820 {
 821         char ereport_class[FM_MAX_CLASS];
 822         const char *name;
 823         va_list ap;
 824         int ret;

 826         if (version != FM_EREPORT_VERS0) {
 827                 atomic_inc_64(&erpt_kstat_data.erpt_set_failed.value.ui64);
 828                 atomic_add_64(&erpt_kstat_data.erpt_set_failed.value.ui64, 1);
 828                 return;
 829         }

 831         (void) snprintf(ereport_class, FM_MAX_CLASS, "%s.%s",
 832             FM_EREPORT_CLASS, erpt_class);
 833         if (nvlist_add_string(ereport, FM_CLASS, ereport_class) != 0) {
 834                 atomic_inc_64(&erpt_kstat_data.erpt_set_failed.value.ui64);
 835                 atomic_add_64(&erpt_kstat_data.erpt_set_failed.value.ui64, 1);
 835                 return;
 836         }

 838         if (nvlist_add_uint64(ereport, FM_EREPORT_ENA, ena)) {
 839                 atomic_inc_64(&erpt_kstat_data.erpt_set_failed.value.ui64);
 840                 atomic_add_64(&erpt_kstat_data.erpt_set_failed.value.ui64, 1);
 840         }

 842         if (nvlist_add_nvlist(ereport, FM_EREPORT_DETECTOR,
 843             (nvlist_t *)detector) != 0) {
 844                 atomic_inc_64(&erpt_kstat_data.erpt_set_failed.value.ui64);
 845                 atomic_add_64(&erpt_kstat_data.erpt_set_failed.value.ui64, 1);
 845         }

 847         va_start(ap, detector);
 848         name = va_arg(ap, const char *);
 849         ret = i_fm_payload_set(ereport, name, ap);
 850         va_end(ap);

 852         if (ret)
 853                 atomic_inc_64(&erpt_kstat_data.erpt_set_failed.value.ui64);
 854                 atomic_add_64(&erpt_kstat_data.erpt_set_failed.value.ui64, 1);
 854 }

 856 /*
 857  * Set-up and validate the members of an hc fmri according to;
 858  *
 859  *      Member name             Type            Value
 860  *      ====================================================
 861  *      version                 uint8_t         0
 862  *      auth                    nvlist_t        <auth>
```

```
 *      hc-name                 string          <name>
 *      hc-id                   string          <id>
 *
 * Note that auth and hc-id are optional members.
 */

#define HC_MAXPAIRS     20
#define HC_MAXNAMELEN   50

static int
fm_fmri_hc_set_common(nvlist_t *fmri, int version, const nvlist_t *auth)
{
        if (version != FM_HC_SCHEME_VERSION) {
                atomic_inc_64(&erpt_kstat_data.fmri_set_failed.value.ui64);
                atomic_add_64(&erpt_kstat_data.fmri_set_failed.value.ui64, 1);
                return (0);
        }

        if (nvlist_add_uint8(fmri, FM_VERSION, version) != 0 ||
            nvlist_add_string(fmri, FM_FMRI_SCHEME, FM_FMRI_SCHEME_HC) != 0) {
                atomic_inc_64(&erpt_kstat_data.fmri_set_failed.value.ui64);
                atomic_add_64(&erpt_kstat_data.fmri_set_failed.value.ui64, 1);
                return (0);
        }

        if (auth != NULL && nvlist_add_nvlist(fmri, FM_FMRI_AUTHORITY,
            (nvlist_t *)auth) != 0) {
                atomic_inc_64(&erpt_kstat_data.fmri_set_failed.value.ui64);
                atomic_add_64(&erpt_kstat_data.fmri_set_failed.value.ui64, 1);
                return (0);
        }

        return (1);
}

void
fm_fmri_hc_set(nvlist_t *fmri, int version, const nvlist_t *auth,
    nvlist_t *snvl, int npairs, ...)
{
        nv_alloc_t *nva = nvlist_lookup_nv_alloc(fmri);
        nvlist_t *pairs[HC_MAXPAIRS];
        va_list ap;
        int i;

        if (!fm_fmri_hc_set_common(fmri, version, auth))
                return;

        npairs = MIN(npairs, HC_MAXPAIRS);

        va_start(ap, npairs);
        for (i = 0; i < npairs; i++) {
                const char *name = va_arg(ap, const char *);
                uint32_t id = va_arg(ap, uint32_t);
                char idstr[11];

                (void) snprintf(idstr, sizeof (idstr), "%u", id);

                pairs[i] = fm_nvlist_create(nva);
                if (nvlist_add_string(pairs[i], FM_FMRI_HC_NAME, name) != 0 ||
                    nvlist_add_string(pairs[i], FM_FMRI_HC_ID, idstr) != 0) {
                        atomic_inc_64(
                            &erpt_kstat_data.fmri_set_failed.value.ui64);
                        atomic_add_64(
                            &erpt_kstat_data.fmri_set_failed.value.ui64, 1);
                }
        }
```



```
        va_end(ap);

        if (nvlist_add_nvlist_array(fmri, FM_FMRI_HC_LIST, pairs, npairs) != 0)
                atomic_inc_64(&erpt_kstat_data.fmri_set_failed.value.ui64);
                atomic_add_64(&erpt_kstat_data.fmri_set_failed.value.ui64, 1);

        for (i = 0; i < npairs; i++)
                fm_nvlist_destroy(pairs[i], FM_NVA_RETAIN);

        if (snvl != NULL) {
                if (nvlist_add_nvlist(fmri, FM_FMRI_HC_SPECIFIC, snvl) != 0) {
                        atomic_inc_64(
                            &erpt_kstat_data.fmri_set_failed.value.ui64);
                        atomic_add_64(
                            &erpt_kstat_data.fmri_set_failed.value.ui64, 1);
                }
        }
}

/*
 * Set-up and validate the members of an dev fmri according to:
 *
 *      Member name             Type            Value
 *      ====================================================
 *      version                 uint8_t         0
 *      auth                    nvlist_t        <auth>
 *      devpath                 string          <devpath>
 *      [devid]                 string          <devid>
 *      [target-port-l0id]      string          <target-port-lun0-id>
 *
 * Note that auth and devid are optional members.
 */
void
fm_fmri_dev_set(nvlist_t *fmri_dev, int version, const nvlist_t *auth,
    const char *devpath, const char *devid, const char *tpl0)
{
        int err = 0;

        if (version != DEV_SCHEME_VERSION0) {
                atomic_inc_64(&erpt_kstat_data.fmri_set_failed.value.ui64);
                atomic_add_64(&erpt_kstat_data.fmri_set_failed.value.ui64, 1);
                return;
        }

        err |= nvlist_add_uint8(fmri_dev, FM_VERSION, version);
        err |= nvlist_add_string(fmri_dev, FM_FMRI_SCHEME, FM_FMRI_SCHEME_DEV);

        if (auth != NULL) {
                err |= nvlist_add_nvlist(fmri_dev, FM_FMRI_AUTHORITY,
                    (nvlist_t *)auth);
        }

        err |= nvlist_add_string(fmri_dev, FM_FMRI_DEV_PATH, devpath);

        if (devid != NULL)
                err |= nvlist_add_string(fmri_dev, FM_FMRI_DEV_ID, devid);

        if (tpl0 != NULL)
                err |= nvlist_add_string(fmri_dev, FM_FMRI_DEV_TGTPTLUN0, tpl0);

        if (err)
                atomic_inc_64(&erpt_kstat_data.fmri_set_failed.value.ui64);
                atomic_add_64(&erpt_kstat_data.fmri_set_failed.value.ui64, 1);

}
```

```
 985 /*
 986  * Set-up and validate the members of an cpu fmri according to:
 987  *
 988  *      Member name             Type            Value
 989  *      ================================================
 990  *      version                 uint8_t         0
 991  *      auth                    nvlist_t        <auth>
 992  *      cpuid                   uint32_t        <cpu_id>
 993  *      cpumask                 uint8_t         <cpu_mask>
 994  *      serial                  uint64_t        <serial_id>
 995  *
 996  * Note that auth, cpumask, serial are optional members.
 997  *
 998  */
 999 void
1000 fm_fmri_cpu_set(nvlist_t *fmri_cpu, int version, const nvlist_t *auth,
1001     uint32_t cpu_id, uint8_t *cpu_maskp, const char *serial_idp)
1002 {
1003         uint64_t *failedp = &erpt_kstat_data.fmri_set_failed.value.ui64;

1005         if (version < CPU_SCHEME_VERSION1) {
1006                 atomic_inc_64(failedp);
1007                 atomic_add_64(failedp, 1);
1007                 return;
1008         }

1010         if (nvlist_add_uint8(fmri_cpu, FM_VERSION, version) != 0) {
1011                 atomic_inc_64(failedp);
1012                 atomic_add_64(failedp, 1);
1012                 return;
1013         }

1015         if (nvlist_add_string(fmri_cpu, FM_FMRI_SCHEME,
1016             FM_FMRI_SCHEME_CPU) != 0) {
1017                 atomic_inc_64(failedp);
1018                 atomic_add_64(failedp, 1);
1018                 return;
1019         }

1021         if (auth != NULL && nvlist_add_nvlist(fmri_cpu, FM_FMRI_AUTHORITY,
1022             (nvlist_t *)auth) != 0)
1023                 atomic_inc_64(failedp);
1024                 atomic_add_64(failedp, 1);

1025         if (nvlist_add_uint32(fmri_cpu, FM_FMRI_CPU_ID, cpu_id) != 0)
1026                 atomic_inc_64(failedp);
1027                 atomic_add_64(failedp, 1);

1028         if (cpu_maskp != NULL && nvlist_add_uint8(fmri_cpu, FM_FMRI_CPU_MASK,
1029             *cpu_maskp) != 0)
1030                 atomic_inc_64(failedp);
1031                 atomic_add_64(failedp, 1);

1032         if (serial_idp == NULL || nvlist_add_string(fmri_cpu,
1033             FM_FMRI_CPU_SERIAL_ID, (char *)serial_idp) != 0)
1034                         atomic_inc_64(failedp);
1035                         atomic_add_64(failedp, 1);
1035 }

1037 /*
1038  * Set-up and validate the members of a mem according to:
1039  *
1040  *      Member name             Type            Value
1041  *      ================================================
1042  *      version                 uint8_t         0
1043  *      auth                    nvlist_t        <auth>          [optional]
```



```
1044  *      unum                    string          <unum>
1045  *      serial                  string          <serial>        [optional*]
1046  *      offset                  uint64_t        <offset>        [optional]
1047  *
1048  *      * serial is required if offset is present
1049  */
1050 void
1051 fm_fmri_mem_set(nvlist_t *fmri, int version, const nvlist_t *auth,
1052     const char *unum, const char *serial, uint64_t offset)
1053 {
1054         if (version != MEM_SCHEME_VERSION0) {
1055                 atomic_inc_64(&erpt_kstat_data.fmri_set_failed.value.ui64);
1056                 atomic_add_64(&erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1056                 return;
1057         }

1059         if (!serial && (offset != (uint64_t)-1)) {
1060                 atomic_inc_64(&erpt_kstat_data.fmri_set_failed.value.ui64);
1061                 atomic_add_64(&erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1061                 return;
1062         }

1064         if (nvlist_add_uint8(fmri, FM_VERSION, version) != 0) {
1065                 atomic_inc_64(&erpt_kstat_data.fmri_set_failed.value.ui64);
1066                 atomic_add_64(&erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1066                 return;
1067         }

1069         if (nvlist_add_string(fmri, FM_FMRI_SCHEME, FM_FMRI_SCHEME_MEM) != 0) {
1070                 atomic_inc_64(&erpt_kstat_data.fmri_set_failed.value.ui64);
1071                 atomic_add_64(&erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1071                 return;
1072         }

1074         if (auth != NULL) {
1075                 if (nvlist_add_nvlist(fmri, FM_FMRI_AUTHORITY,
1076                     (nvlist_t *)auth) != 0) {
1077                         atomic_inc_64(
1078                             &erpt_kstat_data.fmri_set_failed.value.ui64);
1078                         atomic_add_64(
1079                             &erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1079                 }
1080         }

1082         if (nvlist_add_string(fmri, FM_FMRI_MEM_UNUM, unum) != 0) {
1083                 atomic_inc_64(&erpt_kstat_data.fmri_set_failed.value.ui64);
1084                 atomic_add_64(&erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1084         }

1086         if (serial != NULL) {
1087                 if (nvlist_add_string_array(fmri, FM_FMRI_MEM_SERIAL_ID,
1088                     (char **)&serial, 1) != 0) {
1089                         atomic_inc_64(
1090                             &erpt_kstat_data.fmri_set_failed.value.ui64);
1090                         atomic_add_64(
1091                             &erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1092                 }
1093                 if (offset != (uint64_t)-1) {
1094                         if (nvlist_add_uint64(fmri, FM_FMRI_MEM_OFFSET,
1095                             offset) != 0) {
1096                                 atomic_add_64(&erpt_kstat_data.
1097                                     fmri_set_failed.value.ui64, 1);
1091                 }
1092                 if (offset != (uint64_t)-1 && nvlist_add_uint64(fmri,
1093                     FM_FMRI_MEM_OFFSET, offset) != 0) {
1094                         atomic_inc_64(
```

```
1095                            &erpt_kstat_data.fmri_set_failed.value.ui64);
1096 #endif /* ! codereview */
1097                 }
1098         }
1099 }

1101 void
1102 fm_fmri_zfs_set(nvlist_t *fmri, int version, uint64_t pool_guid,
1103     uint64_t vdev_guid)
1104 {
1105         if (version != ZFS_SCHEME_VERSION0) {
1106                 atomic_inc_64(&erpt_kstat_data.fmri_set_failed.value.ui64);
1099                 atomic_add_64(&erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1107                 return;
1108         }

1110         if (nvlist_add_uint8(fmri, FM_VERSION, version) != 0) {
1111                 atomic_inc_64(&erpt_kstat_data.fmri_set_failed.value.ui64);
1104                 atomic_add_64(&erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1112                 return;
1113         }

1115         if (nvlist_add_string(fmri, FM_FMRI_SCHEME, FM_FMRI_SCHEME_ZFS) != 0) {
1116                 atomic_inc_64(&erpt_kstat_data.fmri_set_failed.value.ui64);
1109                 atomic_add_64(&erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1117                 return;
1118         }

1120         if (nvlist_add_uint64(fmri, FM_FMRI_ZFS_POOL, pool_guid) != 0) {
1121                 atomic_inc_64(&erpt_kstat_data.fmri_set_failed.value.ui64);
1114                 atomic_add_64(&erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1122         }

1124         if (vdev_guid != 0) {
1125                 if (nvlist_add_uint64(fmri, FM_FMRI_ZFS_VDEV, vdev_guid) != 0) {
1126                         atomic_inc_64(
1127                             &erpt_kstat_data.fmri_set_failed.value.ui64);
1119                         atomic_add_64(
1120                             &erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1128                 }
1129         }
1130 }
_____unchanged_portion_omitted_

1287 void
1288 fm_fmri_hc_create(nvlist_t *fmri, int version, const nvlist_t *auth,
1289     nvlist_t *snvl, nvlist_t *bboard, int npairs, ...)
1290 {
1291         nv_alloc_t *nva = nvlist_lookup_nv_alloc(fmri);
1292         nvlist_t *pairs[HC_MAXPAIRS];
1293         nvlist_t **hcl;
1294         uint_t n;
1295         int i, j;
1296         va_list ap;
1297         char *hcname, *hcid;

1299         if (!fm_fmri_hc_set_common(fmri, version, auth))
1300                 return;

1302         /*
1303          * copy the bboard nvpairs to the pairs array
1304          */
1305         if (nvlist_lookup_nvlist_array(bboard, FM_FMRI_HC_LIST, &hcl, &n)
1306             != 0) {
1307                 atomic_inc_64(&erpt_kstat_data.fmri_set_failed.value.ui64);
1300                 atomic_add_64(&erpt_kstat_data.fmri_set_failed.value.ui64, 1);
```



```
1308                 return;
1309         }

1311         for (i = 0; i < n; i++) {
1312                 if (nvlist_lookup_string(hcl[i], FM_FMRI_HC_NAME,
1313                     &hcname) != 0) {
1314                         atomic_inc_64(
1315                             &erpt_kstat_data.fmri_set_failed.value.ui64);
1307                         atomic_add_64(
1308                             &erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1316                         return;
1317                 }
1318                 if (nvlist_lookup_string(hcl[i], FM_FMRI_HC_ID, &hcid) != 0) {
1319                         atomic_inc_64(
1320                             &erpt_kstat_data.fmri_set_failed.value.ui64);
1312                         atomic_add_64(
1313                             &erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1321                         return;
1322                 }

1324                 pairs[i] = fm_nvlist_create(nva);
1325                 if (nvlist_add_string(pairs[i], FM_FMRI_HC_NAME, hcname) != 0 ||
1326                     nvlist_add_string(pairs[i], FM_FMRI_HC_ID, hcid) != 0) {
1327                         for (j = 0; j <= i; j++) {
1328                                 if (pairs[j] != NULL)
1329                                         fm_nvlist_destroy(pairs[j],
1330                                             FM_NVA_RETAIN);
1331                         }
1332                         atomic_inc_64(
1333                             &erpt_kstat_data.fmri_set_failed.value.ui64);
1325                         atomic_add_64(
1326                             &erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1334                         return;
1335                 }
1336         }

1338         /*
1339          * create the pairs from passed in pairs
1340          */
1341         npairs = MIN(npairs, HC_MAXPAIRS);

1343         va_start(ap, npairs);
1344         for (i = n; i < npairs + n; i++) {
1345                 const char *name = va_arg(ap, const char *);
1346                 uint32_t id = va_arg(ap, uint32_t);
1347                 char idstr[11];
1348                 (void) snprintf(idstr, sizeof (idstr), "%u", id);
1349                 pairs[i] = fm_nvlist_create(nva);
1350                 if (nvlist_add_string(pairs[i], FM_FMRI_HC_NAME, name) != 0 ||
1351                     nvlist_add_string(pairs[i], FM_FMRI_HC_ID, idstr) != 0) {
1352                         for (j = 0; j <= i; j++) {
1353                                 if (pairs[j] != NULL)
1354                                         fm_nvlist_destroy(pairs[j],
1355                                             FM_NVA_RETAIN);
1356                         }
1357                         atomic_inc_64(
1358                             &erpt_kstat_data.fmri_set_failed.value.ui64);
1350                         atomic_add_64(
1351                             &erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1359                         return;
1360                 }
1361         }
1362         va_end(ap);

1364         /*
1365          * Create the fmri hc list
```

```
1366          */
1367         if (nvlist_add_nvlist_array(fmri, FM_FMRI_HC_LIST, pairs,
1368             npairs + n) != 0) {
1369                 atomic_inc_64(&erpt_kstat_data.fmri_set_failed.value.ui64);
1362                 atomic_add_64(&erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1370                 return;
1371         }

1373         for (i = 0; i < npairs + n; i++) {
1374                         fm_nvlist_destroy(pairs[i], FM_NVA_RETAIN);
1375         }

1377         if (snvl != NULL) {
1378                 if (nvlist_add_nvlist(fmri, FM_FMRI_HC_SPECIFIC, snvl) != 0) {
1379                         atomic_inc_64(
1380                             &erpt_kstat_data.fmri_set_failed.value.ui64);
1372                         atomic_add_64(
1373                             &erpt_kstat_data.fmri_set_failed.value.ui64, 1);
1381                         return;
1382                 }
1383         }
1384 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   36739 Mon Jul 28 07:44:49 2014
new/usr/src/uts/common/os/fork.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 127 /* ARGSUSED */
 128 static int64_t
 129 cfork(int isvfork, int isfork1, int flags)
 130 {
 131         proc_t *p = ttoproc(curthread);
 132         struct as *as;
 133         proc_t *cp, **orphpp;
 134         klwp_t *clone;
 135         kthread_t *t;
 136         task_t *tk;
 137         rval_t  r;
 138         int error;
 139         int i;
 140         rctl_set_t *dup_set;
 141         rctl_alloc_gp_t *dup_gp;
 142         rctl_entity_p_t e;
 143         lwpdir_t *ldp;
 144         lwpent_t *lep;
 145         lwpent_t *clep;

 147         /*
 148          * Allow only these two flags.
 149          */
 150         if ((flags & ~(FORK_NOSIGCHLD | FORK_WAITPID)) != 0) {
 151                 error = EINVAL;
 152                 goto forkerr;
 153         }

 155         /*
 156          * fork is not supported for the /proc agent lwp.
 157          */
 158         if (curthread == p->p_agenttp) {
 159                 error = ENOTSUP;
 160                 goto forkerr;
 161         }

 163         if ((error = secpolicy_basic_fork(CRED())) != 0)
 164                 goto forkerr;

 166         /*
 167          * If the calling lwp is doing a fork1() then the
 168          * other lwps in this process are not duplicated and
 169          * don't need to be held where their kernel stacks can be
 170          * cloned.  If doing forkall(), the process is held with
 171          * SHOLDFORK, so that the lwps are at a point where their
 172          * stacks can be copied which is on entry or exit from
 173          * the kernel.
 174          */
 175         if (!holdlwps(isfork1 ? SHOLDFORK1 : SHOLDFORK)) {
 176                 aston(curthread);
 177                 error = EINTR;
 178                 goto forkerr;
 179         }

 181 #if defined(__sparc)
 182         /*
 183          * Ensure that the user stack is fully constructed
 184          * before creating the child process structure.
 185          */
```



```
 186         (void) flush_user_windows_to_stack(NULL);
 187 #endif

 189         mutex_enter(&p->p_lock);
 190         /*
 191          * If this is vfork(), cancel any suspend request we might
 192          * have gotten from some other thread via lwp_suspend().
 193          * Otherwise we could end up with a deadlock on return
 194          * from the vfork() in both the parent and the child.
 195          */
 196         if (isvfork)
 197                 curthread->t_proc_flag &= ~TP_HOLDLWP;
 198         /*
 199          * Prevent our resource set associations from being changed during fork.
 200          */
 201         pool_barrier_enter();
 202         mutex_exit(&p->p_lock);

 204         /*
 205          * Create a child proc struct. Place a VN_HOLD on appropriate vnodes.
 206          */
 207         if (getproc(&cp, 0, GETPROC_USER) < 0) {
 208                 mutex_enter(&p->p_lock);
 209                 pool_barrier_exit();
 210                 continuelwps(p);
 211                 mutex_exit(&p->p_lock);
 212                 error = EAGAIN;
 213                 goto forkerr;
 214         }

 216         TRACE_2(TR_FAC_PROC, TR_PROC_FORK, "proc_fork:cp %p p %p", cp, p);

 218         /*
 219          * Assign an address space to child
 220          */
 221         if (isvfork) {
 222                 /*
 223                  * Clear any watched areas and remember the
 224                  * watched pages for restoring in vfwait().
 225                  */
 226                 as = p->p_as;
 227                 if (avl_numnodes(&as->a_wpage) != 0) {
 228                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
 229                         as_clearwatch(as);
 230                         p->p_wpage = as->a_wpage;
 231                         avl_create(&as->a_wpage, wp_compare,
 232                             sizeof (struct watched_page),
 233                             offsetof(struct watched_page, wp_link));
 234                         AS_LOCK_EXIT(as, &as->a_lock);
 235                 }
 236                 cp->p_as = as;
 237                 cp->p_flag |= SVFORK;

 239                 /*
 240                  * Use the parent's shm segment list information for
 241                  * the child as it uses its address space till it execs.
 242                  */
 243                 cp->p_segacct = p->p_segacct;
 244         } else {
 245                 /*
 246                  * We need to hold P_PR_LOCK until the address space has
 247                  * been duplicated and we've had a chance to remove from the
 248                  * child any DTrace probes that were in the parent. Holding
 249                  * P_PR_LOCK prevents any new probes from being added and any
 250                  * extant probes from being removed.
 251                  */
```

```
252                 mutex_enter(&p->p_lock);
253                 sprlock_proc(p);
254                 p->p_flag |= SFORKING;
255                 mutex_exit(&p->p_lock);

257                 error = as_dup(p->p_as, cp);
258                 if (error != 0) {
259                         mutex_enter(&p->p_lock);
260                         sprunlock(p);
261                         fork_fail(cp);
262                         mutex_enter(&pidlock);
263                         orphpp = &p->p_orphan;
264                         while (*orphpp != cp)
265                                 orphpp = &(*orphpp)->p_nextorph;
266                         *orphpp = cp->p_nextorph;
267                         if (p->p_child == cp)
268                                 p->p_child = cp->p_sibling;
269                         if (cp->p_sibling)
270                                 cp->p_sibling->p_psibling = cp->p_psibling;
271                         if (cp->p_psibling)
272                                 cp->p_psibling->p_sibling = cp->p_sibling;
273                         mutex_enter(&cp->p_lock);
274                         tk = cp->p_task;
275                         task_detach(cp);
276                         ASSERT(cp->p_pool->pool_ref > 0);
277                         atomic_dec_32(&cp->p_pool->pool_ref);
277                         atomic_add_32(&cp->p_pool->pool_ref, -1);
278                         mutex_exit(&cp->p_lock);
279                         pid_exit(cp, tk);
280                         mutex_exit(&pidlock);
281                         task_rele(tk);

283                         mutex_enter(&p->p_lock);
284                         p->p_flag &= ~SFORKING;
285                         pool_barrier_exit();
286                         continuelwps(p);
287                         mutex_exit(&p->p_lock);
288                         /*
289                          * Preserve ENOMEM error condition but
290                          * map all others to EAGAIN.
291                          */
292                         error = (error == ENOMEM) ? ENOMEM : EAGAIN;
293                         goto forkerr;
294                 }

296                 /*
297                  * Remove all DTrace tracepoints from the child process. We
298                  * need to do this _before_ duplicating USDT providers since
299                  * any associated probes may be immediately enabled.
300                  */
301                 if (p->p_dtrace_count > 0)
302                         dtrace_fasttrap_fork(p, cp);

304                 mutex_enter(&p->p_lock);
305                 sprunlock(p);

307                 /* Duplicate parent's shared memory */
308                 if (p->p_segacct)
309                         shmfork(p, cp);

311                 /*
312                  * Duplicate any helper actions and providers. The SFORKING
313                  * we set above informs the code to enable USDT probes that
314                  * sprlock() may fail because the child is being forked.
315                  */
316                 if (p->p_dtrace_helpers != NULL) {
```



```
317                         ASSERT(dtrace_helpers_fork != NULL);
318                         (*dtrace_helpers_fork)(p, cp);
319                 }

321                 mutex_enter(&p->p_lock);
322                 p->p_flag &= ~SFORKING;
323                 mutex_exit(&p->p_lock);
324         }

326         /*
327          * Duplicate parent's resource controls.
328          */
329         dup_set = rctl_set_create();
330         for (;;) {
331                 dup_gp = rctl_set_dup_prealloc(p->p_rctls);
332                 mutex_enter(&p->p_rctls->rcs_lock);
333                 if (rctl_set_dup_ready(p->p_rctls, dup_gp))
334                         break;
335                 mutex_exit(&p->p_rctls->rcs_lock);
336                 rctl_prealloc_destroy(dup_gp);
337         }
338         e.rcep_p.proc = cp;
339         e.rcep_t = RCENTITY_PROCESS;
340         cp->p_rctls = rctl_set_dup(p->p_rctls, p, cp, &e, dup_set, dup_gp,
341             RCD_DUP | RCD_CALLBACK);
342         mutex_exit(&p->p_rctls->rcs_lock);

344         rctl_prealloc_destroy(dup_gp);

346         /*
347          * Allocate the child's lwp directory and lwpid hash table.
348          */
349         if (isfork1)
350                 cp->p_lwpdir_sz = 2;
351         else
352                 cp->p_lwpdir_sz = p->p_lwpdir_sz;
353         cp->p_lwpdir = cp->p_lwpfree = ldp =
354             kmem_zalloc(cp->p_lwpdir_sz * sizeof (lwpdir_t), KM_SLEEP);
355         for (i = 1; i < cp->p_lwpdir_sz; i++, ldp++)
356                 ldp->ld_next = ldp + 1;
357         cp->p_tidhash_sz = (cp->p_lwpdir_sz + 2) / 2;
358         cp->p_tidhash =
359             kmem_zalloc(cp->p_tidhash_sz * sizeof (tidhash_t), KM_SLEEP);

361         /*
362          * Duplicate parent's lwps.
363          * Mutual exclusion is not needed because the process is
364          * in the hold state and only the current lwp is running.
365          */
366         klgrpset_clear(cp->p_lgrpset);
367         if (isfork1) {
368                 clone = forklwp(ttolwp(curthread), cp, curthread->t_tid);
369                 if (clone == NULL)
370                         goto forklwperr;
371                 /*
372                  * Inherit only the lwp_wait()able flag,
373                  * Daemon threads should not call fork1(), but oh well...
374                  */
375                 lwptot(clone)->t_proc_flag |=
376                     (curthread->t_proc_flag & TP_TWAIT);
377         } else {
378                 /* this is forkall(), no one can be in lwp_wait() */
379                 ASSERT(p->p_lwpwait == 0 && p->p_lwpdwait == 0);
380                 /* for each entry in the parent's lwp directory... */
381                 for (i = 0, ldp = p->p_lwpdir; i < p->p_lwpdir_sz; i++, ldp++) {
382                         klwp_t *clwp;
```

```
383                         kthread_t *ct;
384
385                         if ((lep = ldp->ld_entry) == NULL)
386                                 continue;
387
388                         if ((t = lep->le_thread) != NULL) {
389                                 clwp = forklwp(ttolwp(t), cp, t->t_tid);
390                                 if (clwp == NULL)
391                                         goto forklwperr;
392                                 ct = lwptot(clwp);
393                                 /*
394                                  * Inherit lwp_wait()able and daemon flags.
395                                  */
396                                 ct->t_proc_flag |=
397                                     (t->t_proc_flag & (TP_TWAIT|TP_DAEMON));
398                                 /*
399                                  * Keep track of the clone of curthread to
400                                  * post return values through lwp_setrval().
401                                  * Mark other threads for special treatment
402                                  * by lwp_rtt() / post_syscall().
403                                  */
404                                 if (t == curthread)
405                                         clone = clwp;
406                                 else
407                                         ct->t_flag |= T_FORKALL;
408                         } else {
409                                 /*
410                                  * Replicate zombie lwps in the child.
411                                  */
412                                 clep = kmem_zalloc(sizeof (*clep), KM_SLEEP);
413                                 clep->le_lwpid = lep->le_lwpid;
414                                 clep->le_start = lep->le_start;
415                                 lwp_hash_in(cp, clep,
416                                     cp->p_tidhash, cp->p_tidhash_sz, 0);
417                         }
418                 }
419         }
420
421         /*
422          * Put new process in the parent's process contract, or put it
423          * in a new one if there is an active process template.  Send a
424          * fork event (if requested) to whatever contract the child is
425          * a member of.  Fails if the parent has been SIGKILLed.
426          */
427         if (contract_process_fork(NULL, cp, p, B_TRUE) == NULL)
428                 goto forklwperr;
429
430         /*
431          * No fork failures occur beyond this point.
432          */
433
434         cp->p_lwpid = p->p_lwpid;
435         if (!isfork1) {
436                 cp->p_lwpdaemon = p->p_lwpdaemon;
437                 cp->p_zombcnt = p->p_zombcnt;
438                 /*
439                  * If the parent's lwp ids have wrapped around, so have the
440                  * child's.
441                  */
442                 cp->p_flag |= p->p_flag & SLWPWRAP;
443         }
444
445         mutex_enter(&p->p_lock);
446         corectl_path_hold(cp->p_corefile = p->p_corefile);
447         corectl_content_hold(cp->p_content = p->p_content);
448         mutex_exit(&p->p_lock);
```



```
449
450         /*
451          * Duplicate process context ops, if any.
452          */
453         if (p->p_pctx)
454                 forkpctx(p, cp);
455
456 #ifdef __sparc
457         utrap_dup(p, cp);
458 #endif
459         /*
460          * If the child process has been marked to stop on exit
461          * from this fork, arrange for all other lwps to stop in
462          * sympathy with the active lwp.
463          */
464         if (PTOU(cp)->u_systrap &&
465             prismember(&PTOU(cp)->u_exitmask, curthread->t_sysnum)) {
466                 mutex_enter(&cp->p_lock);
467                 t = cp->p_tlist;
468                 do {
469                         t->t_proc_flag |= TP_PRSTOP;
470                         aston(t);       /* so TP_PRSTOP will be seen */
471                 } while ((t = t->t_forw) != cp->p_tlist);
472                 mutex_exit(&cp->p_lock);
473         }
474         /*
475          * If the parent process has been marked to stop on exit
476          * from this fork, and its asynchronous-stop flag has not
477          * been set, arrange for all other lwps to stop before
478          * they return back to user level.
479          */
480         if (!(p->p_proc_flag & P_PR_ASYNC) && PTOU(p)->u_systrap &&
481             prismember(&PTOU(p)->u_exitmask, curthread->t_sysnum)) {
482                 mutex_enter(&p->p_lock);
483                 t = p->p_tlist;
484                 do {
485                         t->t_proc_flag |= TP_PRSTOP;
486                         aston(t);       /* so TP_PRSTOP will be seen */
487                 } while ((t = t->t_forw) != p->p_tlist);
488                 mutex_exit(&p->p_lock);
489         }
490
491         if (PROC_IS_BRANDED(p))
492                 BROP(p)->b_lwp_setrval(clone, p->p_pid, 1);
493         else
494                 lwp_setrval(clone, p->p_pid, 1);
495
496         /* set return values for parent */
497         r.r_val1 = (int)cp->p_pid;
498         r.r_val2 = 0;
499
500         /*
501          * pool_barrier_exit() can now be called because the child process has:
502          * - all identifying features cloned or set (p_pid, p_task, p_pool)
503          * - all resource sets associated (p_tlist->*->t_cpupart, p_as->a_mset)
504          * - any other fields set which are used in resource set binding.
505          */
506         mutex_enter(&p->p_lock);
507         pool_barrier_exit();
508         mutex_exit(&p->p_lock);
509
510         mutex_enter(&pidlock);
511         mutex_enter(&cp->p_lock);
512
513         /*
514          * Set flags telling the child what (not) to do on exit.
```

```
515          */
516         if (flags & FORK_NOSIGCHLD)
517                 cp->p_pidflag |= CLDNOSIGCHLD;
518         if (flags & FORK_WAITPID)
519                 cp->p_pidflag |= CLDWAITPID;

521         /*
522          * Now that there are lwps and threads attached, add the new
523          * process to the process group.
524          */
525         pgjoin(cp, p->p_pgidp);
526         cp->p_stat = SRUN;
527         /*
528          * We are now done with all the lwps in the child process.
529          */
530         t = cp->p_tlist;
531         do {
532                 /*
533                  * Set the lwp_suspend()ed lwps running.
534                  * They will suspend properly at syscall exit.
535                  */
536                 if (t->t_proc_flag & TP_HOLDLWP)
537                         lwp_create_done(t);
538                 else {
539                         /* set TS_CREATE to allow continuelwps() to work */
540                         thread_lock(t);
541                         ASSERT(t->t_state == TS_STOPPED &&
542                             !(t->t_schedflag & (TS_CREATE|TS_CSTART)));
543                         t->t_schedflag |= TS_CREATE;
544                         thread_unlock(t);
545                 }
546         } while ((t = t->t_forw) != cp->p_tlist);
547         mutex_exit(&cp->p_lock);

549         if (isvfork) {
550                 CPU_STATS_ADDQ(CPU, sys, sysvfork, 1);
551                 mutex_enter(&p->p_lock);
552                 p->p_flag |= SVFWAIT;
553                 curthread->t_flag |= T_VFPARENT;
554                 DTRACE_PROC1(create, proc_t *, cp);
555                 cv_broadcast(&pr_pid_cv[p->p_slot]);    /* inform /proc */
556                 mutex_exit(&p->p_lock);
557                 /*
558                  * Grab child's p_lock before dropping pidlock to ensure
559                  * the process will not disappear before we set it running.
560                  */
561                 mutex_enter(&cp->p_lock);
562                 mutex_exit(&pidlock);
563                 sigdefault(cp);
564                 continuelwps(cp);
565                 mutex_exit(&cp->p_lock);
566         } else {
567                 CPU_STATS_ADDQ(CPU, sys, sysfork, 1);
568                 DTRACE_PROC1(create, proc_t *, cp);
569                 /*
570                  * It is CL_FORKRET's job to drop pidlock.
571                  * If we do it here, the process could be set running
572                  * and disappear before CL_FORKRET() is called.
573                  */
574                 CL_FORKRET(curthread, cp->p_tlist);
575                 schedctl_set_cidpri(curthread);
576                 ASSERT(MUTEX_NOT_HELD(&pidlock));
577         }

579         return (r.r_vals);
```



```
581 forklwperr:
582         if (isvfork) {
583                 if (avl_numnodes(&p->p_wpage) != 0) {
584                         /* restore watchpoints to parent */
585                         as = p->p_as;
586                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
587                         as->a_wpage = p->p_wpage;
588                         avl_create(&p->p_wpage, wp_compare,
589                             sizeof (struct watched_page),
590                             offsetof(struct watched_page, wp_link));
591                         as_setwatch(as);
592                         AS_LOCK_EXIT(as, &as->a_lock);
593                 }
594         } else {
595                 if (cp->p_segacct)
596                         shmexit(cp);
597                 as = cp->p_as;
598                 cp->p_as = &kas;
599                 as_free(as);
600         }

602         if (cp->p_lwpdir) {
603                 for (i = 0, ldp = cp->p_lwpdir; i < cp->p_lwpdir_sz; i++, ldp++)
604                         if ((lep = ldp->ld_entry) != NULL)
605                                 kmem_free(lep, sizeof (*lep));
606                 kmem_free(cp->p_lwpdir,
607                     cp->p_lwpdir_sz * sizeof (*cp->p_lwpdir));
608         }
609         cp->p_lwpdir = NULL;
610         cp->p_lwpfree = NULL;
611         cp->p_lwpdir_sz = 0;

613         if (cp->p_tidhash)
614                 kmem_free(cp->p_tidhash,
615                     cp->p_tidhash_sz * sizeof (*cp->p_tidhash));
616         cp->p_tidhash = NULL;
617         cp->p_tidhash_sz = 0;

619         forklwp_fail(cp);
620         fork_fail(cp);
621         rctl_set_free(cp->p_rctls);
622         mutex_enter(&pidlock);

624         /*
625          * Detach failed child from task.
626          */
627         mutex_enter(&cp->p_lock);
628         tk = cp->p_task;
629         task_detach(cp);
630         ASSERT(cp->p_pool->pool_ref > 0);
631         atomic_dec_32(&cp->p_pool->pool_ref);
631         atomic_add_32(&cp->p_pool->pool_ref, -1);
632         mutex_exit(&cp->p_lock);

634         orphpp = &p->p_orphan;
635         while (*orphpp != cp)
636                 orphpp = &(*orphpp)->p_nextorph;
637         *orphpp = cp->p_nextorph;
638         if (p->p_child == cp)
639                 p->p_child = cp->p_sibling;
640         if (cp->p_sibling)
641                 cp->p_sibling->p_psibling = cp->p_psibling;
642         if (cp->p_psibling)
643                 cp->p_psibling->p_sibling = cp->p_sibling;
644         pid_exit(cp, tk);
645         mutex_exit(&pidlock);
```

```
 647         task_rele(tk);

 649         mutex_enter(&p->p_lock);
 650         pool_barrier_exit();
 651         continuelwps(p);
 652         mutex_exit(&p->p_lock);
 653         error = EAGAIN;
 654 forkerr:
 655         return ((int64_t)set_errno(error));
 656 }
_____unchanged_portion_omitted_

 916 /*
 917  * create a child proc struct.
 918  */
 919 static int
 920 getproc(proc_t **cpp, pid_t pid, uint_t flags)
 921 {
 922         proc_t          *pp, *cp;
 923         pid_t           newpid;
 924         struct user     *uarea;
 925         extern uint_t   nproc;
 926         struct cred     *cr;
 927         uid_t           ruid;
 928         zoneid_t        zoneid;
 929         task_t          *task;
 930         kproject_t      *proj;
 931         zone_t          *zone;
 932         int             rctlfail = 0;

 934         if (zone_status_get(curproc->p_zone) >= ZONE_IS_SHUTTING_DOWN)
 935                 return (-1);    /* no point in starting new processes */

 937         pp = (flags & GETPROC_KERNEL) ? &p0 : curproc;
 938         task = pp->p_task;
 939         proj = task->tk_proj;
 940         zone = pp->p_zone;

 942         mutex_enter(&pp->p_lock);
 943         mutex_enter(&zone->zone_nlwps_lock);
 944         if (proj != proj0p) {
 945                 if (task->tk_nprocs >= task->tk_nprocs_ctl)
 946                         if (rctl_test(rc_task_nprocs, task->tk_rctls,
 947                             pp, 1, 0) & RCT_DENY)
 948                                 rctlfail = 1;

 950                 if (proj->kpj_nprocs >= proj->kpj_nprocs_ctl)
 951                         if (rctl_test(rc_project_nprocs, proj->kpj_rctls,
 952                             pp, 1, 0) & RCT_DENY)
 953                                 rctlfail = 1;

 955                 if (zone->zone_nprocs >= zone->zone_nprocs_ctl)
 956                         if (rctl_test(rc_zone_nprocs, zone->zone_rctls,
 957                             pp, 1, 0) & RCT_DENY)
 958                                 rctlfail = 1;

 960                 if (rctlfail) {
 961                         mutex_exit(&zone->zone_nlwps_lock);
 962                         mutex_exit(&pp->p_lock);
 963                         goto punish;
 964                 }
 965         }
 966         task->tk_nprocs++;
 967         proj->kpj_nprocs++;
 968         zone->zone_nprocs++;
```



```
 969         mutex_exit(&zone->zone_nlwps_lock);
 970         mutex_exit(&pp->p_lock);

 972         cp = kmem_cache_alloc(process_cache, KM_SLEEP);
 973         bzero(cp, sizeof (proc_t));

 975         /*
 976          * Make proc entry for child process
 977          */
 978         mutex_init(&cp->p_splock, NULL, MUTEX_DEFAULT, NULL);
 979         mutex_init(&cp->p_crlock, NULL, MUTEX_DEFAULT, NULL);
 980         mutex_init(&cp->p_pflock, NULL, MUTEX_DEFAULT, NULL);
 981 #if defined(__x86)
 982         mutex_init(&cp->p_ldtlock, NULL, MUTEX_DEFAULT, NULL);
 983 #endif
 984         mutex_init(&cp->p_maplock, NULL, MUTEX_DEFAULT, NULL);
 985         cp->p_stat = SIDL;
 986         cp->p_mstart = gethrtime();
 987         cp->p_as = &kas;
 988         /*
 989          * p_zone must be set before we call pid_allocate since the process
 990          * will be visible after that and code such as prfind_zone will
 991          * look at the p_zone field.
 992          */
 993         cp->p_zone = pp->p_zone;
 994         cp->p_t1_lgrpid = LGRP_NONE;
 995         cp->p_tr_lgrpid = LGRP_NONE;

 997         if ((newpid = pid_allocate(cp, pid, PID_ALLOC_PROC)) == -1) {
 998                 if (nproc == v.v_proc) {
 999                         CPU_STATS_ADDQ(CPU, sys, procovf, 1);
1000                         cmn_err(CE_WARN, "out of processes");
1001                 }
1002                 goto bad;
1003         }

1005         mutex_enter(&pp->p_lock);
1006         cp->p_exec = pp->p_exec;
1007         cp->p_execdir = pp->p_execdir;
1008         mutex_exit(&pp->p_lock);

1010         if (cp->p_exec) {
1011                 VN_HOLD(cp->p_exec);
1012                 /*
1013                  * Each VOP_OPEN() must be paired with a corresponding
1014                  * VOP_CLOSE(). In this case, the executable will be
1015                  * closed for the child in either proc_exit() or gexec().
1016                  */
1017                 if (VOP_OPEN(&cp->p_exec, FREAD, CRED(), NULL) != 0) {
1018                         VN_RELE(cp->p_exec);
1019                         cp->p_exec = NULLVP;
1020                         cp->p_execdir = NULLVP;
1021                         goto bad;
1022                 }
1023         }
1024         if (cp->p_execdir)
1025                 VN_HOLD(cp->p_execdir);

1027         /*
1028          * If not privileged make sure that this user hasn't exceeded
1029          * v.v_maxup processes, and that users collectively haven't
1030          * exceeded v.v_maxupttl processes.
1031          */
1032         mutex_enter(&pidlock);
1033         ASSERT(nproc < v.v_proc);       /* otherwise how'd we get our pid? */
1034         cr = CRED();
```

```
1035         ruid = crgetruid(cr);
1036         zoneid = crgetzoneid(cr);
1037         if (nproc >= v.v_maxup &&       /* short-circuit; usually false */
1038             (nproc >= v.v_maxupttl ||
1039             upcount_get(ruid, zoneid) >= v.v_maxup) &&
1040             secpolicy_newproc(cr) != 0) {
1041                 mutex_exit(&pidlock);
1042                 zcmn_err(zoneid, CE_NOTE,
1043                     "out of per-user processes for uid %d", ruid);
1044                 goto bad;
1045         }

1047         /*
1048          * Everything is cool, put the new proc on the active process list.
1049          * It is already on the pid list and in /proc.
1050          * Increment the per uid process count (upcount).
1051          */
1052         nproc++;
1053         upcount_inc(ruid, zoneid);

1055         cp->p_next = practive;
1056         practive->p_prev = cp;
1057         practive = cp;

1059         cp->p_ignore = pp->p_ignore;
1060         cp->p_siginfo = pp->p_siginfo;
1061         cp->p_flag = pp->p_flag & (SJCTL|SNOWAIT|SNOCD);
1062         cp->p_sessp = pp->p_sessp;
1063         sess_hold(pp);
1064         cp->p_brand = pp->p_brand;
1065         if (PROC_IS_BRANDED(pp))
1066                 BROP(pp)->b_copy_procdata(cp, pp);
1067         cp->p_bssbase = pp->p_bssbase;
1068         cp->p_brkbase = pp->p_brkbase;
1069         cp->p_brksize = pp->p_brksize;
1070         cp->p_brkpageszc = pp->p_brkpageszc;
1071         cp->p_stksize = pp->p_stksize;
1072         cp->p_stkpageszc = pp->p_stkpageszc;
1073         cp->p_stkprot = pp->p_stkprot;
1074         cp->p_datprot = pp->p_datprot;
1075         cp->p_usrstack = pp->p_usrstack;
1076         cp->p_model = pp->p_model;
1077         cp->p_ppid = pp->p_pid;
1078         cp->p_ancpid = pp->p_pid;
1079         cp->p_portcnt = pp->p_portcnt;

1081         /*
1082          * Initialize watchpoint structures
1083          */
1084         avl_create(&cp->p_warea, wa_compare, sizeof (struct watched_area),
1085             offsetof(struct watched_area, wa_link));

1087         /*
1088          * Initialize immediate resource control values.
1089          */
1090         cp->p_stk_ctl = pp->p_stk_ctl;
1091         cp->p_fsz_ctl = pp->p_fsz_ctl;
1092         cp->p_vmem_ctl = pp->p_vmem_ctl;
1093         cp->p_fno_ctl = pp->p_fno_ctl;

1095         /*
1096          * Link up to parent-child-sibling chain.  No need to lock
1097          * in general since only a call to freeproc() (done by the
1098          * same parent as newproc()) diddles with the child chain.
1099          */
1100         cp->p_sibling = pp->p_child;
```



```
1101         if (pp->p_child)
1102                 pp->p_child->p_psibling = cp;

1104         cp->p_parent = pp;
1105         pp->p_child = cp;

1107         cp->p_child_ns = NULL;
1108         cp->p_sibling_ns = NULL;

1110         cp->p_nextorph = pp->p_orphan;
1111         cp->p_nextofkin = pp;
1112         pp->p_orphan = cp;

1114         /*
1115          * Inherit profiling state; do not inherit REALPROF profiling state.
1116          */
1117         cp->p_prof = pp->p_prof;
1118         cp->p_rprof_cyclic = CYCLIC_NONE;

1120         /*
1121          * Inherit pool pointer from the parent.  Kernel processes are
1122          * always bound to the default pool.
1123          */
1124         mutex_enter(&pp->p_lock);
1125         if (flags & GETPROC_KERNEL) {
1126                 cp->p_pool = pool_default;
1127                 cp->p_flag |= SSYS;
1128         } else {
1129                 cp->p_pool = pp->p_pool;
1130         }
1131         atomic_inc_32(&cp->p_pool->pool_ref);
1131         atomic_add_32(&cp->p_pool->pool_ref, 1);
1132         mutex_exit(&pp->p_lock);

1134         /*
1135          * Add the child process to the current task.  Kernel processes
1136          * are always attached to task0.
1137          */
1138         mutex_enter(&cp->p_lock);
1139         if (flags & GETPROC_KERNEL)
1140                 task_attach(task0p, cp);
1141         else
1142                 task_attach(pp->p_task, cp);
1143         mutex_exit(&cp->p_lock);
1144         mutex_exit(&pidlock);

1146         avl_create(&cp->p_ct_held, contract_compar, sizeof (contract_t),
1147             offsetof(contract_t, ct_ctlist));

1149         /*
1150          * Duplicate any audit information kept in the process table
1151          */
1152         if (audit_active)       /* copy audit data to cp */
1153                 audit_newproc(cp);

1155         crhold(cp->p_cred = cr);

1157         /*
1158          * Bump up the counts on the file structures pointed at by the
1159          * parent's file table since the child will point at them too.
1160          */
1161         fcnt_add(P_FINFO(pp), 1);

1163         if (PTOU(pp)->u_cdir) {
1164                 VN_HOLD(PTOU(pp)->u_cdir);
1165         } else {
```

```
1166                 ASSERT(pp == &p0);
1167                 /*
1168                  * We must be at or before vfs_mountroot(); it will take care of
1169                  * assigning our current directory.
1170                  */
1171         }
1172         if (PTOU(pp)->u_rdir)
1173                 VN_HOLD(PTOU(pp)->u_rdir);
1174         if (PTOU(pp)->u_cwd)
1175                 refstr_hold(PTOU(pp)->u_cwd);

1177         /*
1178          * copy the parent's uarea.
1179          */
1180         uarea = PTOU(cp);
1181         bcopy(PTOU(pp), uarea, sizeof (*uarea));
1182         flist_fork(P_FINFO(pp), P_FINFO(cp));

1184         gethrestime(&uarea->u_start);
1185         uarea->u_ticks = ddi_get_lbolt();
1186         uarea->u_mem = rm_asrss(pp->p_as);
1187         uarea->u_acflag = AFORK;

1189         /*
1190          * If inherit-on-fork, copy /proc tracing flags to child.
1191          */
1192         if ((pp->p_proc_flag & P_PR_FORK) != 0) {
1193                 cp->p_proc_flag |= pp->p_proc_flag & (P_PR_TRACE|P_PR_FORK);
1194                 cp->p_sigmask = pp->p_sigmask;
1195                 cp->p_fltmask = pp->p_fltmask;
1196         } else {
1197                 sigemptyset(&cp->p_sigmask);
1198                 premptyset(&cp->p_fltmask);
1199                 uarea->u_systrap = 0;
1200                 premptyset(&uarea->u_entrymask);
1201                 premptyset(&uarea->u_exitmask);
1202         }
1203         /*
1204          * If microstate accounting is being inherited, mark child
1205          */
1206         if ((pp->p_flag & SMSFORK) != 0)
1207                 cp->p_flag |= pp->p_flag & (SMSFORK|SMSACCT);

1209         /*
1210          * Inherit fixalignment flag from the parent
1211          */
1212         cp->p_fixalignment = pp->p_fixalignment;

1214         *cpp = cp;
1215         return (0);

1217 bad:
1218         ASSERT(MUTEX_NOT_HELD(&pidlock));

1220         mutex_destroy(&cp->p_crlock);
1221         mutex_destroy(&cp->p_pflock);
1222 #if defined(__x86)
1223         mutex_destroy(&cp->p_ldtlock);
1224 #endif
1225         if (newpid != -1) {
1226                 proc_entry_free(cp->p_pidp);
1227                 (void) pid_rele(cp->p_pidp);
1228         }
1229         kmem_cache_free(process_cache, cp);

1231         mutex_enter(&zone->zone_nlwps_lock);
```



```
1232         task->tk_nprocs--;
1233         proj->kpj_nprocs--;
1234         zone->zone_nprocs--;
1235         mutex_exit(&zone->zone_nlwps_lock);

1237 punish:
1238         /*
1239          * We most likely got into this situation because some process is
1240          * forking out of control.  As punishment, put it to sleep for a
1241          * bit so it can't eat the machine alive.  Sleep interval is chosen
1242          * to allow no more than one fork failure per cpu per clock tick
1243          * on average (yes, I just made this up).  This has two desirable
1244          * properties: (1) it sets a constant limit on the fork failure
1245          * rate, and (2) the busier the system is, the harsher the penalty
1246          * for abusing it becomes.
1247          */
1248         INCR_COUNT(&fork_fail_pending, &pidlock);
1249         delay(fork_fail_pending / ncpus + 1);
1250         DECR_COUNT(&fork_fail_pending, &pidlock);

1252         return (-1); /* out of memory or proc slots */
1253 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   63738 Mon Jul 28 07:44:49 2014
new/usr/src/uts/common/os/kcpc.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 912 /*
 913  * Generic interrupt handler used on hardware that generates
 914  * overflow interrupts.
 915  *
 916  * Note: executed at high-level interrupt context!
 917  */
 918 /*ARGSUSED*/
 919 kcpc_ctx_t *
 920 kcpc_overflow_intr(caddr_t arg, uint64_t bitmap)
 921 {
 922         kcpc_ctx_t      *ctx;
 923         kthread_t       *t = curthread;
 924         int             i;

 926         /*
 927          * On both x86 and UltraSPARC, we may deliver the high-level
 928          * interrupt in kernel mode, just after we've started to run an
 929          * interrupt thread.  (That's because the hardware helpfully
 930          * delivers the overflow interrupt some random number of cycles
 931          * after the instruction that caused the overflow by which time
 932          * we're in some part of the kernel, not necessarily running on
 933          * the right thread).
 934          *
 935          * Check for this case here -- find the pinned thread
 936          * that was running when the interrupt went off.
 937          */
 938         if (t->t_flag & T_INTR_THREAD) {
 939                 klwp_t *lwp;

 941                 atomic_inc_32(&kcpc_intrctx_count);
 941                 atomic_add_32(&kcpc_intrctx_count, 1);

 943                 /*
 944                  * Note that t_lwp is always set to point at the underlying
 945                  * thread, thus this will work in the presence of nested
 946                  * interrupts.
 947                  */
 948                 ctx = NULL;
 949                 if ((lwp = t->t_lwp) != NULL) {
 950                         t = lwptot(lwp);
 951                         ctx = t->t_cpc_ctx;
 952                 }
 953         } else
 954                 ctx = t->t_cpc_ctx;

 956         if (ctx == NULL) {
 957                 /*
 958                  * This can easily happen if we're using the counters in
 959                  * "shared" mode, for example, and an overflow interrupt
 960                  * occurs while we are running cpustat.  In that case, the
 961                  * bound thread that has the context that belongs to this
 962                  * CPU is almost certainly sleeping (if it was running on
 963                  * the CPU we'd have found it above), and the actual
 964                  * interrupted thread has no knowledge of performance counters!
 965                  */
 966                 ctx = curthread->t_cpu->cpu_cpc_ctx;
 967                 if (ctx != NULL) {
 968                         /*
 969                          * Return the bound context for this CPU to
```



```
 970                          * the interrupt handler so that it can synchronously
 971                          * sample the hardware counters and restart them.
 972                          */
 973                         return (ctx);
 974                 }

 976                 /*
 977                  * As long as the overflow interrupt really is delivered early
 978                  * enough after trapping into the kernel to avoid switching
 979                  * threads, we must always be able to find the cpc context,
 980                  * or something went terribly wrong i.e. we ended up
 981                  * running a passivated interrupt thread, a kernel
 982                  * thread or we interrupted idle, all of which are Very Bad.
 983                  *
 984                  * We also could end up here owing to an incredibly unlikely
 985                  * race condition that exists on x86 based architectures when
 986                  * the cpc provider is in use; overflow interrupts are directed
 987                  * to the cpc provider if the 'dtrace_cpc_in_use' variable is
 988                  * set when we enter the handler. This variable is unset after
 989                  * overflow interrupts have been disabled on all CPUs and all
 990                  * contexts have been torn down. To stop interrupts, the cpc
 991                  * provider issues a xcall to the remote CPU before it tears
 992                  * down that CPUs context. As high priority xcalls, on an x86
 993                  * architecture, execute at a higher PIL than this handler, it
 994                  * is possible (though extremely unlikely) that the xcall could
 995                  * interrupt the overflow handler before the handler has
 996                  * checked the 'dtrace_cpc_in_use' variable, stop the counters,
 997                  * return to the cpc provider which could then rip down
 998                  * contexts and unset 'dtrace_cpc_in_use' *before* the CPUs
 999                  * overflow handler has had a chance to check the variable. In
1000                  * that case, the handler would direct the overflow into this
1001                  * code and no valid context will be found. The default behavior
1002                  * when no valid context is found is now to shout a warning to
1003                  * the console and bump the 'kcpc_nullctx_count' variable.
1004                  */
1005                 if (kcpc_nullctx_panic)
1006                         panic("null cpc context, thread %p", (void *)t);
1007 #ifdef DEBUG
1008                 cmn_err(CE_NOTE,
1009                     "null cpc context found in overflow handler!\n");
1010 #endif
1011                 atomic_inc_32(&kcpc_nullctx_count);
1011                 atomic_add_32(&kcpc_nullctx_count, 1);
1012         } else if ((ctx->kc_flags & KCPC_CTX_INVALID) == 0) {
1013                 /*
1014                  * Schedule an ast to sample the counters, which will
1015                  * propagate any overflow into the virtualized performance
1016                  * counter(s), and may deliver a signal.
1017                  */
1018                 ttolwp(t)->lwp_pcb.pcb_flags |= CPC_OVERFLOW;
1019                 /*
1020                  * If a counter has overflowed which was counting on behalf of
1021                  * a request which specified CPC_OVF_NOTIFY_EMT, send the
1022                  * process a signal.
1023                  */
1024                 for (i = 0; i < cpc_ncounters; i++) {
1025                         if (ctx->kc_pics[i].kp_req != NULL &&
1026                             bitmap & (1 << i) &&
1027                             ctx->kc_pics[i].kp_req->kr_flags &
1028                             CPC_OVF_NOTIFY_EMT) {
1029                                 /*
1030                                  * A signal has been requested for this PIC, so
1031                                  * so freeze the context. The interrupt handler
1032                                  * has already stopped the counter hardware.
1033                                  */
1034                                 KCPC_CTX_FLAG_SET(ctx, KCPC_CTX_FREEZE);
```

```
1035                                atomic_or_uint(&ctx->kc_pics[i].kp_flags,
1036                                    KCPC_PIC_OVERFLOWED);
1037                        }
1038                }
1039                aston(t);
1040        } else if (ctx->kc_flags & KCPC_CTX_INVALID_STOPPED) {
1041                /*
1042                 * Thread context is no longer valid, but here may be a valid
1043                 * CPU context.
1044                 */
1045                return (curthread->t_cpu->cpu_cpc_ctx);
1046        }

1048        return (NULL);
1049 }
```

unchanged_portion_omitted_

```
**********************************************************
   24888 Mon Jul 28 07:44:49 2014
new/usr/src/uts/common/os/klpd.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

  76 klpd_reg_t *klpd_list;

  78 static void klpd_unlink(klpd_reg_t *);
  79 static int klpd_unreg_dh(door_handle_t);

  81 static credklpd_t *crklpd_alloc(void);

  83 void crklpd_setreg(credklpd_t *, klpd_reg_t *);

  85 extern size_t max_vnode_path;

  87 void
  88 klpd_rele(klpd_reg_t *p)
  89 {
  90         if (atomic_dec_32_nv(&p->klpd_ref) == 0) {
  90         if (atomic_add_32_nv(&p->klpd_ref, -1) == 0) {
  91                 if (p->klpd_refp != NULL)
  92                         klpd_unlink(p);
  93                 if (p->klpd_cred != NULL)
  94                         crfree(p->klpd_cred);
  95                 door_ki_rele(p->klpd_door);
  96                 kmem_free(p, sizeof (*p));
  97         }
  98 }
______unchanged_portion_omitted_


 116 static void
 117 klpd_hold(klpd_reg_t *p)
 118 {
 119         atomic_inc_32(&p->klpd_ref);
 119         atomic_add_32(&p->klpd_ref, 1);
 120 }
______unchanged_portion_omitted_

 334 uint32_t klpd_bad_locks;

 336 int
 337 klpd_call(const cred_t *cr, const priv_set_t *req, va_list ap)
 338 {
 339         klpd_reg_t *p;
 340         int rv = -1;
 341         credklpd_t *ckp;
 342         zone_t *ckzone;

 344         /*
 345          * These locks must not be held when this code is called;
 346          * callbacks to userland with these locks held will result
 347          * in issues.  That said, the code at the call sides was
 348          * restructured not to call with any of the locks held and
 349          * no policies operate by default on most processes.
 350          */
 351         if (mutex_owned(&pidlock) || mutex_owned(&curproc->p_lock) ||
 352             mutex_owned(&curproc->p_crlock)) {
 353                 atomic_inc_32(&klpd_bad_locks);
 353                 atomic_add_32(&klpd_bad_locks, 1);
 354                 return (-1);
 355         }
```



```
 357         /*
 358          * Enforce the limit set for the call process (still).
 359          */
 360         if (!priv_issubset(req, &CR_LPRIV(cr)))
 361                 return (-1);

 363         /* Try 1: get the credential specific klpd */
 364         if ((ckp = crgetcrklpd(cr)) != NULL) {
 365                 mutex_enter(&ckp->crkl_lock);
 366                 if ((p = ckp->crkl_reg) != NULL &&
 367                     p->klpd_indel == 0 &&
 368                     priv_issubset(req, &p->klpd_pset)) {
 369                         klpd_hold(p);
 370                         mutex_exit(&ckp->crkl_lock);
 371                         rv = klpd_do_call(p, req, ap);
 372                         mutex_enter(&ckp->crkl_lock);
 373                         klpd_rele(p);
 374                         mutex_exit(&ckp->crkl_lock);
 375                         if (rv != -1)
 376                                 return (rv == 0 ? 0 : -1);
 377                 } else {
 378                         mutex_exit(&ckp->crkl_lock);
 379                 }
 380         }

 382         /* Try 2: get the project specific klpd */
 383         mutex_enter(&klpd_mutex);

 385         if ((p = curproj->kpj_klpd) != NULL) {
 386                 klpd_hold(p);
 387                 mutex_exit(&klpd_mutex);
 388                 if (p->klpd_indel == 0 &&
 389                     priv_issubset(req, &p->klpd_pset)) {
 390                         rv = klpd_do_call(p, req, ap);
 391                 }
 392                 mutex_enter(&klpd_mutex);
 393                 klpd_rele(p);
 394                 mutex_exit(&klpd_mutex);

 396                 if (rv != -1)
 397                         return (rv == 0 ? 0 : -1);
 398         } else {
 399                 mutex_exit(&klpd_mutex);
 400         }

 402         /* Try 3: get the global klpd list */
 403         ckzone = crgetzone(cr);
 404         mutex_enter(&klpd_mutex);

 406         for (p = klpd_list; p != NULL; ) {
 407                 zone_t *kkzone = crgetzone(p->klpd_cred);
 408                 if ((kkzone == &zone0 || kkzone == ckzone) &&
 409                     p->klpd_indel == 0 &&
 410                     priv_issubset(req, &p->klpd_pset)) {
 411                         klpd_hold(p);
 412                         mutex_exit(&klpd_mutex);
 413                         rv = klpd_do_call(p, req, ap);
 414                         mutex_enter(&klpd_mutex);

 416                         p = klpd_rele_next(p);

 418                         if (rv != -1)
 419                                 break;
 420                 } else {
 421                         p = p->klpd_next;
 422                 }
```

```
423         }
424         mutex_exit(&klpd_mutex);
425         return (rv == 0 ? 0 : -1);
426 }
______unchanged_portion_omitted_

674 void
675 crklpd_hold(credklpd_t *crkpd)
676 {
677         atomic_inc_32(&crkpd->crkl_ref);
677         atomic_add_32(&crkpd->crkl_ref, 1);
678 }

680 void
681 crklpd_rele(credklpd_t *crkpd)
682 {
683         if (atomic_dec_32_nv(&crkpd->crkl_ref) == 0) {
683         if (atomic_add_32_nv(&crkpd->crkl_ref, -1) == 0) {
684                 if (crkpd->crkl_reg != NULL)
685                         klpd_rele(crkpd->crkl_reg);
686                 mutex_destroy(&crkpd->crkl_lock);
687                 kmem_free(crkpd, sizeof (*crkpd));
688         }
689 }
______unchanged_portion_omitted_
```

```
**********************************************************
  181700 Mon Jul 28 07:44:49 2014
new/usr/src/uts/common/os/kmem.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1521 /*
1522  * Create a new slab for cache cp.
1523  */
1524 static kmem_slab_t *
1525 kmem_slab_create(kmem_cache_t *cp, int kmflag)
1526 {
1527         size_t slabsize = cp->cache_slabsize;
1528         size_t chunksize = cp->cache_chunksize;
1529         int cache_flags = cp->cache_flags;
1530         size_t color, chunks;
1531         char *buf, *slab;
1532         kmem_slab_t *sp;
1533         kmem_bufctl_t *bcp;
1534         vmem_t *vmp = cp->cache_arena;

1536         ASSERT(MUTEX_NOT_HELD(&cp->cache_lock));

1538         color = cp->cache_color + cp->cache_align;
1539         if (color > cp->cache_maxcolor)
1540                 color = cp->cache_mincolor;
1541         cp->cache_color = color;

1543         slab = vmem_alloc(vmp, slabsize, kmflag & KM_VMFLAGS);

1545         if (slab == NULL)
1546                 goto vmem_alloc_failure;

1548         ASSERT(P2PHASE((uintptr_t)slab, vmp->vm_quantum) == 0);

1550         /*
1551          * Reverify what was already checked in kmem_cache_set_move(), since the
1552          * consolidator depends (for correctness) on slabs being initialized
1553          * with the 0xbaddcafe memory pattern (setting a low order bit usable by
1554          * clients to distinguish uninitialized memory from known objects).
1555          */
1556         ASSERT((cp->cache_move == NULL) || !(cp->cache_cflags & KMC_NOTOUCH));
1557         if (!(cp->cache_cflags & KMC_NOTOUCH))
1558                 copy_pattern(KMEM_UNINITIALIZED_PATTERN, slab, slabsize);

1560         if (cache_flags & KMF_HASH) {
1561                 if ((sp = kmem_cache_alloc(kmem_slab_cache, kmflag)) == NULL)
1562                         goto slab_alloc_failure;
1563                 chunks = (slabsize - color) / chunksize;
1564         } else {
1565                 sp = KMEM_SLAB(cp, slab);
1566                 chunks = (slabsize - sizeof (kmem_slab_t) - color) / chunksize;
1567         }

1569         sp->slab_cache  = cp;
1570         sp->slab_head   = NULL;
1571         sp->slab_refcnt = 0;
1572         sp->slab_base   = buf = slab + color;
1573         sp->slab_chunks = chunks;
1574         sp->slab_stuck_offset = (uint32_t)-1;
1575         sp->slab_later_count = 0;
1576         sp->slab_flags = 0;

1578         ASSERT(chunks > 0);
1579         while (chunks-- != 0) {
```



```
1580                 if (cache_flags & KMF_HASH) {
1581                         bcp = kmem_cache_alloc(cp->cache_bufctl_cache, kmflag);
1582                         if (bcp == NULL)
1583                                 goto bufctl_alloc_failure;
1584                         if (cache_flags & KMF_AUDIT) {
1585                                 kmem_bufctl_audit_t *bcap =
1586                                     (kmem_bufctl_audit_t *)bcp;
1587                                 bzero(bcap, sizeof (kmem_bufctl_audit_t));
1588                                 bcap->bc_cache = cp;
1589                         }
1590                         bcp->bc_addr = buf;
1591                         bcp->bc_slab = sp;
1592                 } else {
1593                         bcp = KMEM_BUFCTL(cp, buf);
1594                 }
1595                 if (cache_flags & KMF_BUFTAG) {
1596                         kmem_buftag_t *btp = KMEM_BUFTAG(cp, buf);
1597                         btp->bt_redzone = KMEM_REDZONE_PATTERN;
1598                         btp->bt_bufctl = bcp;
1599                         btp->bt_bxstat = (intptr_t)bcp ^ KMEM_BUFTAG_FREE;
1600                         if (cache_flags & KMF_DEADBEEF) {
1601                                 copy_pattern(KMEM_FREE_PATTERN, buf,
1602                                     cp->cache_verify);
1603                         }
1604                 }
1605                 bcp->bc_next = sp->slab_head;
1606                 sp->slab_head = bcp;
1607                 buf += chunksize;
1608         }

1610         kmem_log_event(kmem_slab_log, cp, sp, slab);

1612         return (sp);

1614 bufctl_alloc_failure:

1616         while ((bcp = sp->slab_head) != NULL) {
1617                 sp->slab_head = bcp->bc_next;
1618                 kmem_cache_free(cp->cache_bufctl_cache, bcp);
1619         }
1620         kmem_cache_free(kmem_slab_cache, sp);

1622 slab_alloc_failure:

1624         vmem_free(vmp, slab, slabsize);

1626 vmem_alloc_failure:

1628         kmem_log_event(kmem_failure_log, cp, NULL, NULL);
1629         atomic_inc_64(&cp->cache_alloc_fail);
1629         atomic_add_64(&cp->cache_alloc_fail, 1);

1631         return (NULL);
1632 }
______unchanged_portion_omitted_

1941 /*
1942  * Return -1 if kmem_error, 1 if constructor fails, 0 if successful.
1943  */
1944 static int
1945 kmem_cache_alloc_debug(kmem_cache_t *cp, void *buf, int kmflag, int construct,
1946     caddr_t caller)
1947 {
1948         kmem_buftag_t *btp = KMEM_BUFTAG(cp, buf);
1949         kmem_bufctl_audit_t *bcp = (kmem_bufctl_audit_t *)btp->bt_bufctl;
1950         uint32_t mtbf;
```

```
1952         if (btp->bt_bxstat != ((intptr_t)bcp ^ KMEM_BUFTAG_FREE)) {
1953                 kmem_error(KMERR_BADBUFTAG, cp, buf);
1954                 return (-1);
1955         }

1957         btp->bt_bxstat = (intptr_t)bcp ^ KMEM_BUFTAG_ALLOC;

1959         if ((cp->cache_flags & KMF_HASH) && bcp->bc_addr != buf) {
1960                 kmem_error(KMERR_BADBUFCTL, cp, buf);
1961                 return (-1);
1962         }

1964         if (cp->cache_flags & KMF_DEADBEEF) {
1965                 if (!construct && (cp->cache_flags & KMF_LITE)) {
1966                         if (*(uint64_t *)buf != KMEM_FREE_PATTERN) {
1967                                 kmem_error(KMERR_MODIFIED, cp, buf);
1968                                 return (-1);
1969                         }
1970                         if (cp->cache_constructor != NULL)
1971                                 *(uint64_t *)buf = btp->bt_redzone;
1972                         else
1973                                 *(uint64_t *)buf = KMEM_UNINITIALIZED_PATTERN;
1974                 } else {
1975                         construct = 1;
1976                         if (verify_and_copy_pattern(KMEM_FREE_PATTERN,
1977                             KMEM_UNINITIALIZED_PATTERN, buf,
1978                             cp->cache_verify)) {
1979                                 kmem_error(KMERR_MODIFIED, cp, buf);
1980                                 return (-1);
1981                         }
1982                 }
1983         }
1984         btp->bt_redzone = KMEM_REDZONE_PATTERN;

1986         if ((mtbf = kmem_mtbf | cp->cache_mtbf) != 0 &&
1987             gethrtime() % mtbf == 0 &&
1988             (kmflag & (KM_NOSLEEP | KM_PANIC)) == KM_NOSLEEP) {
1989                 kmem_log_event(kmem_failure_log, cp, NULL, NULL);
1990                 if (!construct && cp->cache_destructor != NULL)
1991                         cp->cache_destructor(buf, cp->cache_private);
1992         } else {
1993                 mtbf = 0;
1994         }

1996         if (mtbf || (construct && cp->cache_constructor != NULL &&
1997             cp->cache_constructor(buf, cp->cache_private, kmflag) != 0)) {
1998                 atomic_inc_64(&cp->cache_alloc_fail);
1998                 atomic_add_64(&cp->cache_alloc_fail, 1);
1999                 btp->bt_bxstat = (intptr_t)bcp ^ KMEM_BUFTAG_FREE;
2000                 if (cp->cache_flags & KMF_DEADBEEF)
2001                         copy_pattern(KMEM_FREE_PATTERN, buf, cp->cache_verify);
2002                 kmem_slab_free(cp, buf);
2003                 return (1);
2004         }

2006         if (cp->cache_flags & KMF_AUDIT) {
2007                 KMEM_AUDIT(kmem_transaction_log, cp, bcp);
2008         }

2010         if ((cp->cache_flags & KMF_LITE) &&
2011             !(cp->cache_cflags & KMC_KMEM_ALLOC)) {
2012                 KMEM_BUFTAG_LITE_ENTER(btp, kmem_lite_count, caller);
2013         }

2015         return (0);
```



```
2016 }
______unchanged_portion_omitted_

2483 /*
2484  * Allocate a constructed object from cache cp.
2485  */
2486 void *
2487 kmem_cache_alloc(kmem_cache_t *cp, int kmflag)
2488 {
2489         kmem_cpu_cache_t *ccp = KMEM_CPU_CACHE(cp);
2490         kmem_magazine_t *fmp;
2491         void *buf;

2493         mutex_enter(&ccp->cc_lock);
2494         for (;;) {
2495                 /*
2496                  * If there's an object available in the current CPU's
2497                  * loaded magazine, just take it and return.
2498                  */
2499                 if (ccp->cc_rounds > 0) {
2500                         buf = ccp->cc_loaded->mag_round[--ccp->cc_rounds];
2501                         ccp->cc_alloc++;
2502                         mutex_exit(&ccp->cc_lock);
2503                         if (ccp->cc_flags & (KMF_BUFTAG | KMF_DUMPUNSAFE)) {
2504                                 if (ccp->cc_flags & KMF_DUMPUNSAFE) {
2505                                         ASSERT(!(ccp->cc_flags &
2506                                             KMF_DUMPDIVERT));
2507                                         KDI_LOG(cp, kdl_unsafe);
2508                                 }
2509                                 if ((ccp->cc_flags & KMF_BUFTAG) &&
2510                                     kmem_cache_alloc_debug(cp, buf, kmflag, 0,
2511                                     caller()) != 0) {
2512                                         if (kmflag & KM_NOSLEEP)
2513                                                 return (NULL);
2514                                         mutex_enter(&ccp->cc_lock);
2515                                         continue;
2516                                 }
2517                         }
2518                         return (buf);
2519                 }

2521                 /*
2522                  * The loaded magazine is empty.  If the previously loaded
2523                  * magazine was full, exchange them and try again.
2524                  */
2525                 if (ccp->cc_prounds > 0) {
2526                         kmem_cpu_reload(ccp, ccp->cc_ploaded, ccp->cc_prounds);
2527                         continue;
2528                 }

2530                 /*
2531                  * Return an alternate buffer at dump time to preserve
2532                  * the heap.
2533                  */
2534                 if (ccp->cc_flags & (KMF_DUMPDIVERT | KMF_DUMPUNSAFE)) {
2535                         if (ccp->cc_flags & KMF_DUMPUNSAFE) {
2536                                 ASSERT(!(ccp->cc_flags & KMF_DUMPDIVERT));
2537                                 /* log it so that we can warn about it */
2538                                 KDI_LOG(cp, kdl_unsafe);
2539                         } else {
2540                                 if ((buf = kmem_cache_alloc_dump(cp, kmflag)) !=
2541                                     NULL) {
2542                                         mutex_exit(&ccp->cc_lock);
2543                                         return (buf);
2544                                 }
2545                                 break;          /* fall back to slab layer */
```

```
2546                        }
2547                }

2549                /*
2550                 * If the magazine layer is disabled, break out now.
2551                 */
2552                if (ccp->cc_magsize == 0)
2553                        break;

2555                /*
2556                 * Try to get a full magazine from the depot.
2557                 */
2558                fmp = kmem_depot_alloc(cp, &cp->cache_full);
2559                if (fmp != NULL) {
2560                        if (ccp->cc_ploaded != NULL)
2561                                kmem_depot_free(cp, &cp->cache_empty,
2562                                    ccp->cc_ploaded);
2563                        kmem_cpu_reload(ccp, fmp, ccp->cc_magsize);
2564                        continue;
2565                }

2567                /*
2568                 * There are no full magazines in the depot,
2569                 * so fall through to the slab layer.
2570                 */
2571                break;
2572        }
2573        mutex_exit(&ccp->cc_lock);

2575        /*
2576         * We couldn't allocate a constructed object from the magazine layer,
2577         * so get a raw buffer from the slab layer and apply its constructor.
2578         */
2579        buf = kmem_slab_alloc(cp, kmflag);

2581        if (buf == NULL)
2582                return (NULL);

2584        if (cp->cache_flags & KMF_BUFTAG) {
2585                /*
2586                 * Make kmem_cache_alloc_debug() apply the constructor for us.
2587                 */
2588                int rc = kmem_cache_alloc_debug(cp, buf, kmflag, 1, caller());
2589                if (rc != 0) {
2590                        if (kmflag & KM_NOSLEEP)
2591                                return (NULL);
2592                        /*
2593                         * kmem_cache_alloc_debug() detected corruption
2594                         * but didn't panic (kmem_panic <= 0). We should not be
2595                         * here because the constructor failed (indicated by a
2596                         * return code of 1). Try again.
2597                         */
2598                        ASSERT(rc == -1);
2599                        return (kmem_cache_alloc(cp, kmflag));
2600                }
2601                return (buf);
2602        }

2604        if (cp->cache_constructor != NULL &&
2605            cp->cache_constructor(buf, cp->cache_private, kmflag) != 0) {
2606                atomic_inc_64(&cp->cache_alloc_fail);
2606                atomic_add_64(&cp->cache_alloc_fail, 1);
2607                kmem_slab_free(cp, buf);
2608                return (NULL);
2609        }
```



```
2611        return (buf);
2612 }
_____unchanged_portion_omitted_

4789 static void kmem_move_end(kmem_cache_t *, kmem_move_t *);

4791 /*
4792  * The move callback takes two buffer addresses, the buffer to be moved, and a
4793  * newly allocated and constructed buffer selected by kmem as the destination.
4794  * It also takes the size of the buffer and an optional user argument specified
4795  * at cache creation time. kmem guarantees that the buffer to be moved has not
4796  * been unmapped by the virtual memory subsystem. Beyond that, it cannot
4797  * guarantee the present whereabouts of the buffer to be moved, so it is up to
4798  * the client to safely determine whether or not it is still using the buffer.
4799  * The client must not free either of the buffers passed to the move callback,
4800  * since kmem wants to free them directly to the slab layer. The client response
4801  * tells kmem which of the two buffers to free:
4802  *
4803  * YES          kmem frees the old buffer (the move was successful)
4804  * NO           kmem frees the new buffer, marks the slab of the old buffer
4805  *              non-reclaimable to avoid bothering the client again
4806  * LATER        kmem frees the new buffer, increments slab_later_count
4807  * DONT_KNOW    kmem frees the new buffer, searches mags for the old buffer
4808  * DONT_NEED    kmem frees both the old buffer and the new buffer
4809  *
4810  * The pending callback argument now being processed contains both of the
4811  * buffers (old and new) passed to the move callback function, the slab of the
4812  * old buffer, and flags related to the move request, such as whether or not the
4813  * system was desperate for memory.
4814  *
4815  * Slabs are not freed while there is a pending callback, but instead are kept
4816  * on a deadlist, which is drained after the last callback completes. This means
4817  * that slabs are safe to access until kmem_move_end(), no matter how many of
4818  * their buffers have been freed. Once slab_refcnt reaches zero, it stays at
4819  * zero for as long as the slab remains on the deadlist and until the slab is
4820  * freed.
4821  */
4822 static void
4823 kmem_move_buffer(kmem_move_t *callback)
4824 {
4825        kmem_cbrc_t response;
4826        kmem_slab_t *sp = callback->kmm_from_slab;
4827        kmem_cache_t *cp = sp->slab_cache;
4828        boolean_t free_on_slab;

4830        ASSERT(taskq_member(kmem_move_taskq, curthread));
4831        ASSERT(MUTEX_NOT_HELD(&cp->cache_lock));
4832        ASSERT(KMEM_SLAB_MEMBER(sp, callback->kmm_from_buf));

4834        /*
4835         * The number of allocated buffers on the slab may have changed since we
4836         * last checked the slab's reclaimability (when the pending move was
4837         * enqueued), or the client may have responded NO when asked to move
4838         * another buffer on the same slab.
4839         */
4840        if (!kmem_slab_is_reclaimable(cp, sp, callback->kmm_flags)) {
4841                KMEM_STAT_ADD(kmem_move_stats.kms_no_longer_reclaimable);
4842                KMEM_STAT_COND_ADD((callback->kmm_flags & KMM_NOTIFY),
4843                    kmem_move_stats.kms_notify_no_longer_reclaimable);
4844                kmem_slab_free(cp, callback->kmm_to_buf);
4845                kmem_move_end(cp, callback);
4846                return;
4847        }

4849        /*
4850         * Hunting magazines is expensive, so we'll wait to do that until the
```

```
4851          * client responds KMEM_CBRC_DONT_KNOW. However, checking the slab layer
4852          * is cheap, so we might as well do that here in case we can avoid
4853          * bothering the client.
4854          */
4855         mutex_enter(&cp->cache_lock);
4856         free_on_slab = (kmem_slab_allocated(cp, sp,
4857             callback->kmm_from_buf) == NULL);
4858         mutex_exit(&cp->cache_lock);
4859 
4860         if (free_on_slab) {
4861                 KMEM_STAT_ADD(kmem_move_stats.kms_hunt_found_slab);
4862                 kmem_slab_free(cp, callback->kmm_to_buf);
4863                 kmem_move_end(cp, callback);
4864                 return;
4865         }
4866 
4867         if (cp->cache_flags & KMF_BUFTAG) {
4868                 /*
4869                  * Make kmem_cache_alloc_debug() apply the constructor for us.
4870                  */
4871                 if (kmem_cache_alloc_debug(cp, callback->kmm_to_buf,
4872                     KM_NOSLEEP, 1, caller()) != 0) {
4873                         KMEM_STAT_ADD(kmem_move_stats.kms_alloc_fail);
4874                         kmem_move_end(cp, callback);
4875                         return;
4876                 }
4877         } else if (cp->cache_constructor != NULL &&
4878             cp->cache_constructor(callback->kmm_to_buf, cp->cache_private,
4879             KM_NOSLEEP) != 0) {
4880                 atomic_inc_64(&cp->cache_alloc_fail);
4880                 atomic_add_64(&cp->cache_alloc_fail, 1);
4881                 KMEM_STAT_ADD(kmem_move_stats.kms_constructor_fail);
4882                 kmem_slab_free(cp, callback->kmm_to_buf);
4883                 kmem_move_end(cp, callback);
4884                 return;
4885         }
4886 
4887         KMEM_STAT_ADD(kmem_move_stats.kms_callbacks);
4888         KMEM_STAT_COND_ADD((callback->kmm_flags & KMM_NOTIFY),
4889             kmem_move_stats.kms_notify_callbacks);
4890         cp->cache_defrag->kmd_callbacks++;
4891         cp->cache_defrag->kmd_thread = curthread;
4892         cp->cache_defrag->kmd_from_buf = callback->kmm_from_buf;
4893         cp->cache_defrag->kmd_to_buf = callback->kmm_to_buf;
4894         DTRACE_PROBE2(kmem__move__start, kmem_cache_t *, cp, kmem_move_t *,
4895             callback);
4896 
4897         response = cp->cache_move(callback->kmm_from_buf,
4898             callback->kmm_to_buf, cp->cache_bufsize, cp->cache_private);
4899 
4900         DTRACE_PROBE3(kmem__move__end, kmem_cache_t *, cp, kmem_move_t *,
4901             callback, kmem_cbrc_t, response);
4902         cp->cache_defrag->kmd_thread = NULL;
4903         cp->cache_defrag->kmd_from_buf = NULL;
4904         cp->cache_defrag->kmd_to_buf = NULL;
4905 
4906         if (response == KMEM_CBRC_YES) {
4907                 KMEM_STAT_ADD(kmem_move_stats.kms_yes);
4908                 cp->cache_defrag->kmd_yes++;
4909                 kmem_slab_free_constructed(cp, callback->kmm_from_buf, B_FALSE);
4910                 /* slab safe to access until kmem_move_end() */
4911                 if (sp->slab_refcnt == 0)
4912                         cp->cache_defrag->kmd_slabs_freed++;
4913                 mutex_enter(&cp->cache_lock);
4914                 kmem_slab_move_yes(cp, sp, callback->kmm_from_buf);
4915                 mutex_exit(&cp->cache_lock);
```



```
4916                 kmem_move_end(cp, callback);
4917                 return;
4918         }
4919 
4920         switch (response) {
4921         case KMEM_CBRC_NO:
4922                 KMEM_STAT_ADD(kmem_move_stats.kms_no);
4923                 cp->cache_defrag->kmd_no++;
4924                 mutex_enter(&cp->cache_lock);
4925                 kmem_slab_move_no(cp, sp, callback->kmm_from_buf);
4926                 mutex_exit(&cp->cache_lock);
4927                 break;
4928         case KMEM_CBRC_LATER:
4929                 KMEM_STAT_ADD(kmem_move_stats.kms_later);
4930                 cp->cache_defrag->kmd_later++;
4931                 mutex_enter(&cp->cache_lock);
4932                 if (!KMEM_SLAB_IS_PARTIAL(sp)) {
4933                         mutex_exit(&cp->cache_lock);
4934                         break;
4935                 }
4936 
4937                 if (++sp->slab_later_count >= KMEM_DISBELIEF) {
4938                         KMEM_STAT_ADD(kmem_move_stats.kms_disbelief);
4939                         kmem_slab_move_no(cp, sp, callback->kmm_from_buf);
4940                 } else if (!(sp->slab_flags & KMEM_SLAB_NOMOVE)) {
4941                         sp->slab_stuck_offset = KMEM_SLAB_OFFSET(sp,
4942                             callback->kmm_from_buf);
4943                 }
4944                 mutex_exit(&cp->cache_lock);
4945                 break;
4946         case KMEM_CBRC_DONT_NEED:
4947                 KMEM_STAT_ADD(kmem_move_stats.kms_dont_need);
4948                 cp->cache_defrag->kmd_dont_need++;
4949                 kmem_slab_free_constructed(cp, callback->kmm_from_buf, B_FALSE);
4950                 if (sp->slab_refcnt == 0)
4951                         cp->cache_defrag->kmd_slabs_freed++;
4952                 mutex_enter(&cp->cache_lock);
4953                 kmem_slab_move_yes(cp, sp, callback->kmm_from_buf);
4954                 mutex_exit(&cp->cache_lock);
4955                 break;
4956         case KMEM_CBRC_DONT_KNOW:
4957                 KMEM_STAT_ADD(kmem_move_stats.kms_dont_know);
4958                 cp->cache_defrag->kmd_dont_know++;
4959                 if (kmem_hunt_mags(cp, callback->kmm_from_buf) != NULL) {
4960                         KMEM_STAT_ADD(kmem_move_stats.kms_hunt_found_mag);
4961                         cp->cache_defrag->kmd_hunt_found++;
4962                         kmem_slab_free_constructed(cp, callback->kmm_from_buf,
4963                             B_TRUE);
4964                         if (sp->slab_refcnt == 0)
4965                                 cp->cache_defrag->kmd_slabs_freed++;
4966                         mutex_enter(&cp->cache_lock);
4967                         kmem_slab_move_yes(cp, sp, callback->kmm_from_buf);
4968                         mutex_exit(&cp->cache_lock);
4969                 }
4970                 break;
4971         default:
4972                 panic("'%s' (%p) unexpected move callback response %d\n",
4973                     cp->cache_name, (void *)cp, response);
4974         }
4975 
4976         kmem_slab_free_constructed(cp, callback->kmm_to_buf, B_FALSE);
4977         kmem_move_end(cp, callback);
4978 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  119430 Mon Jul 28 07:44:50 2014
new/usr/src/uts/common/os/lgrp.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 555 /*
 556  * Handle lgroup (re)configuration events (eg. addition of CPU, etc.)
 557  */
 558 void
 559 lgrp_config(lgrp_config_flag_t event, uintptr_t resource, uintptr_t where)
 560 {
 561         klgrpset_t      changed;
 562         cpu_t           *cp;
 563         lgrp_id_t       id;
 564         int             rc;

 566         switch (event) {
 567         /*
 568          * The following (re)configuration events are common code
 569          * initiated. lgrp_plat_config() is called here to inform the
 570          * platform of the reconfiguration event.
 571          */
 572         case LGRP_CONFIG_CPU_ADD:
 573                 cp = (cpu_t *)resource;

 575                 /*
 576                  * Initialize the new CPU's lgrp related next/prev
 577                  * links, and give it a bootstrap lpl so that it can
 578                  * survive should it need to enter the dispatcher.
 579                  */
 580                 cp->cpu_next_lpl = cp;
 581                 cp->cpu_prev_lpl = cp;
 582                 cp->cpu_next_lgrp = cp;
 583                 cp->cpu_prev_lgrp = cp;
 584                 cp->cpu_lpl = lpl_bootstrap;

 586                 lgrp_plat_config(event, resource);
 587                 atomic_inc_32(&lgrp_gen);
 587                 atomic_add_32(&lgrp_gen, 1);

 589                 break;
 590         case LGRP_CONFIG_CPU_DEL:
 591                 lgrp_plat_config(event, resource);
 592                 atomic_inc_32(&lgrp_gen);
 592                 atomic_add_32(&lgrp_gen, 1);

 594                 break;
 595         case LGRP_CONFIG_CPU_ONLINE:
 596                 cp = (cpu_t *)resource;
 597                 lgrp_cpu_init(cp);
 598                 lgrp_part_add_cpu(cp, cp->cpu_lpl->lpl_lgrpid);
 599                 rc = lpl_topo_verify(cp->cpu_part);
 600                 if (rc != LPL_TOPO_CORRECT) {
 601                         panic("lpl_topo_verify failed: %d", rc);
 602                 }
 603                 lgrp_plat_config(event, resource);
 604                 atomic_inc_32(&lgrp_gen);
 604                 atomic_add_32(&lgrp_gen, 1);

 606                 break;
 607         case LGRP_CONFIG_CPU_OFFLINE:
 608                 cp = (cpu_t *)resource;
 609                 id = cp->cpu_lpl->lpl_lgrpid;
 610                 lgrp_part_del_cpu(cp);
```



```
 611                 lgrp_cpu_fini(cp, id);
 612                 rc = lpl_topo_verify(cp->cpu_part);
 613                 if (rc != LPL_TOPO_CORRECT) {
 614                         panic("lpl_topo_verify failed: %d", rc);
 615                 }
 616                 lgrp_plat_config(event, resource);
 617                 atomic_inc_32(&lgrp_gen);
 617                 atomic_add_32(&lgrp_gen, 1);

 619                 break;
 620         case LGRP_CONFIG_CPUPART_ADD:
 621                 cp = (cpu_t *)resource;
 622                 lgrp_part_add_cpu((cpu_t *)resource, (lgrp_id_t)where);
 623                 rc = lpl_topo_verify(cp->cpu_part);
 624                 if (rc != LPL_TOPO_CORRECT) {
 625                         panic("lpl_topo_verify failed: %d", rc);
 626                 }
 627                 lgrp_plat_config(event, resource);

 629                 break;
 630         case LGRP_CONFIG_CPUPART_DEL:
 631                 cp = (cpu_t *)resource;
 632                 lgrp_part_del_cpu((cpu_t *)resource);
 633                 rc = lpl_topo_verify(cp->cpu_part);
 634                 if (rc != LPL_TOPO_CORRECT) {
 635                         panic("lpl_topo_verify failed: %d", rc);
 636                 }
 637                 lgrp_plat_config(event, resource);

 639                 break;
 640         /*
 641          * The following events are initiated by the memnode
 642          * subsystem.
 643          */
 644         case LGRP_CONFIG_MEM_ADD:
 645                 lgrp_mem_init((int)resource, where, B_FALSE);
 646                 atomic_inc_32(&lgrp_gen);
 646                 atomic_add_32(&lgrp_gen, 1);

 648                 break;
 649         case LGRP_CONFIG_MEM_DEL:
 650                 lgrp_mem_fini((int)resource, where, B_FALSE);
 651                 atomic_inc_32(&lgrp_gen);
 651                 atomic_add_32(&lgrp_gen, 1);

 653                 break;
 654         case LGRP_CONFIG_MEM_RENAME: {
 655                 lgrp_config_mem_rename_t *ren_arg =
 656                     (lgrp_config_mem_rename_t *)where;

 658                 lgrp_mem_rename((int)resource,
 659                     ren_arg->lmem_rename_from,
 660                     ren_arg->lmem_rename_to);
 661                 atomic_inc_32(&lgrp_gen);
 661                 atomic_add_32(&lgrp_gen, 1);

 663                 break;
 664         }
 665         case LGRP_CONFIG_GEN_UPDATE:
 666                 atomic_inc_32(&lgrp_gen);
 666                 atomic_add_32(&lgrp_gen, 1);

 668                 break;
 669         case LGRP_CONFIG_FLATTEN:
 670                 if (where == 0)
 671                         lgrp_topo_levels = (int)resource;
```

```
672                 else
673                         (void) lgrp_topo_flatten(resource,
674                             lgrp_table, lgrp_alloc_max, &changed);

676                 break;
677         /*
678          * Update any lgroups with old latency to new latency
679          */
680         case LGRP_CONFIG_LAT_CHANGE_ALL:
681                 lgrp_latency_change(LGRP_NULL_HANDLE, (u_longlong_t)resource,
682                     (u_longlong_t)where);

684                 break;
685         /*
686          * Update lgroup with specified lgroup platform handle to have
687          * new latency
688          */
689         case LGRP_CONFIG_LAT_CHANGE:
690                 lgrp_latency_change((lgrp_handle_t)resource, 0,
691                     (u_longlong_t)where);

693                 break;
694         case LGRP_CONFIG_NOP:

696                 break;
697         default:
698                 break;
699         }

701 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   68445 Mon Jul 28 07:44:50 2014
new/usr/src/uts/common/os/mmapobj.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 243 #define LIB_VA_SIZE     1024
 244 #define LIB_VA_MASK     (LIB_VA_SIZE - 1)
 245 #define LIB_VA_MUTEX_SHIFT      3

 247 #if (LIB_VA_SIZE & (LIB_VA_SIZE - 1))
 248 #error  "LIB_VA_SIZE is not a power of 2"
 249 #endif

 251 static struct lib_va *lib_va_hash[LIB_VA_SIZE];
 252 static kmutex_t lib_va_hash_mutex[LIB_VA_SIZE >> LIB_VA_MUTEX_SHIFT];

 254 #define LIB_VA_HASH_MUTEX(index)                                        \
 255         (&lib_va_hash_mutex[index >> LIB_VA_MUTEX_SHIFT])

 257 #define LIB_VA_HASH(nodeid)                                             \
 258         (((nodeid) ^ ((nodeid) << 7) ^ ((nodeid) << 13)) & LIB_VA_MASK)

 260 #define LIB_VA_MATCH_ID(arg1, arg2)                                     \
 261         ((arg1)->lv_nodeid == (arg2)->va_nodeid &&                      \
 262         (arg1)->lv_fsid == (arg2)->va_fsid)

 264 #define LIB_VA_MATCH_TIME(arg1, arg2)                                   \
 265         ((arg1)->lv_ctime.tv_sec == (arg2)->va_ctime.tv_sec &&          \
 266         (arg1)->lv_mtime.tv_sec == (arg2)->va_mtime.tv_sec &&           \
 267         (arg1)->lv_ctime.tv_nsec == (arg2)->va_ctime.tv_nsec &&         \
 268         (arg1)->lv_mtime.tv_nsec == (arg2)->va_mtime.tv_nsec)

 270 #define LIB_VA_MATCH(arg1, arg2)                                        \
 271         (LIB_VA_MATCH_ID(arg1, arg2) && LIB_VA_MATCH_TIME(arg1, arg2))

 273 /*
 274  * lib_va will be used for optimized allocation of address ranges for
 275  * libraries, such that subsequent mappings of the same library will attempt
 276  * to use the same VA as previous mappings of that library.
 277  * In order to map libraries at the same VA in many processes, we need to carve
 278  * out our own address space for them which is unique across many processes.
 279  * We use different arenas for 32 bit and 64 bit libraries.
 280  *
 281  * Since the 32 bit address space is relatively small, we limit the number of
 282  * libraries which try to use consistent virtual addresses to lib_threshold.
 283  * For 64 bit libraries there is no such limit since the address space is large.
 284  */
 285 static vmem_t *lib_va_32_arena;
 286 static vmem_t *lib_va_64_arena;
 287 uint_t lib_threshold = 20;      /* modifiable via /etc/system */

 289 static kmutex_t lib_va_init_mutex;      /* no need to initialize */

 291 /*
 292  * Number of 32 bit and 64 bit libraries in lib_va hash.
 293  */
 294 static uint_t libs_mapped_32 = 0;
 295 static uint_t libs_mapped_64 = 0;

 297 /*
 298  * Free up the resources associated with lvp as well as lvp itself.
 299  * We also decrement the number of libraries mapped via a lib_va
 300  * cached virtual address.
 301  */
```



```
 302 void
 303 lib_va_free(struct lib_va *lvp)
 304 {
 305         int is_64bit = lvp->lv_flags & LV_ELF64;
 306         ASSERT(lvp->lv_refcnt == 0);

 308         if (lvp->lv_base_va != NULL) {
 309                 vmem_xfree(is_64bit ? lib_va_64_arena : lib_va_32_arena,
 310                     lvp->lv_base_va, lvp->lv_len);
 311                 if (is_64bit) {
 312                         atomic_dec_32(&libs_mapped_64);
 312                         atomic_add_32(&libs_mapped_64, -1);
 313                 } else {
 314                         atomic_dec_32(&libs_mapped_32);
 314                         atomic_add_32(&libs_mapped_32, -1);
 315                 }
 316         }
 317         kmem_free(lvp, sizeof (struct lib_va));
 318 }
_____unchanged_portion_omitted_

 376 /*
 377  * Add a new entry to the lib_va hash.
 378  * Search the hash while holding the appropriate mutex to make sure that the
 379  * data is not already in the cache.  If we find data that is in the cache
 380  * already and has not been modified since last use, we return NULL.  If it
 381  * has been modified since last use, we will remove that entry from
 382  * the hash and it will be deleted once it's reference count reaches zero.
 383  * If there is no current entry in the hash we will add the new entry and
 384  * return it to the caller who is responsible for calling lib_va_release to
 385  * drop their reference count on it.
 386  *
 387  * lv_num_segs will be set to zero since the caller needs to add that
 388  * information to the data structure.
 389  */
 390 static struct lib_va *
 391 lib_va_add_hash(caddr_t base_va, ssize_t len, size_t align, vattr_t *vap)
 392 {
 393         struct lib_va *lvp;
 394         uint_t index;
 395         model_t model;
 396         struct lib_va **tmp;
 397         struct lib_va *del = NULL;

 399         model = get_udatamodel();
 400         index = LIB_VA_HASH(vap->va_nodeid);

 402         lvp = kmem_alloc(sizeof (struct lib_va), KM_SLEEP);

 404         mutex_enter(LIB_VA_HASH_MUTEX(index));

 406         /*
 407          * Make sure not adding same data a second time.
 408          * The hash chains should be relatively short and adding
 409          * is a relatively rare event, so it's worth the check.
 410          */
 411         tmp = &lib_va_hash[index];
 412         while (*tmp != NULL) {
 413                 if (LIB_VA_MATCH_ID(*tmp, vap)) {
 414                         if (LIB_VA_MATCH_TIME(*tmp, vap)) {
 415                                 mutex_exit(LIB_VA_HASH_MUTEX(index));
 416                                 kmem_free(lvp, sizeof (struct lib_va));
 417                                 return (NULL);
 418                         }

 420                         /*
```

```
421                          * We have the same nodeid and fsid but the file has
422                          * been modified since we last saw it.
423                          * Need to remove the old node and add this new
424                          * one.
425                          * Could probably use a callback mechanism to make
426                          * this cleaner.
427                          */
428                         ASSERT(del == NULL);
429                         del = *tmp;
430                         *tmp = del->lv_next;
431                         del->lv_next = NULL;

433                         /*
434                          * Check to see if we can free it.  If lv_refcnt
435                          * is greater than zero, than some other thread
436                          * has a reference to the one we want to delete
437                          * and we can not delete it.  All of this is done
438                          * under the lib_va_hash_mutex lock so it is atomic.
439                          */
440                         if (del->lv_refcnt) {
441                                 MOBJ_STAT_ADD(lib_va_add_delay_delete);
442                                 del->lv_flags |= LV_DEL;
443                                 del = NULL;
444                         }
445                         /* tmp is already advanced */
446                         continue;
447                 }
448                 tmp = &((*tmp)->lv_next);
449         }

451         lvp->lv_base_va = base_va;
452         lvp->lv_len = len;
453         lvp->lv_align = align;
454         lvp->lv_nodeid = vap->va_nodeid;
455         lvp->lv_fsid = vap->va_fsid;
456         lvp->lv_ctime.tv_sec = vap->va_ctime.tv_sec;
457         lvp->lv_ctime.tv_nsec = vap->va_ctime.tv_nsec;
458         lvp->lv_mtime.tv_sec = vap->va_mtime.tv_sec;
459         lvp->lv_mtime.tv_nsec = vap->va_mtime.tv_nsec;
460         lvp->lv_next = NULL;
461         lvp->lv_refcnt = 1;

463         /* Caller responsible for filling this and lv_mps out */
464         lvp->lv_num_segs = 0;

466         if (model == DATAMODEL_LP64) {
467                 lvp->lv_flags = LV_ELF64;
468         } else {
469                 ASSERT(model == DATAMODEL_ILP32);
470                 lvp->lv_flags = LV_ELF32;
471         }

473         if (base_va != NULL) {
474                 if (model == DATAMODEL_LP64) {
475                         atomic_inc_32(&libs_mapped_64);
475                         atomic_add_32(&libs_mapped_64, 1);
476                 } else {
477                         ASSERT(model == DATAMODEL_ILP32);
478                         atomic_inc_32(&libs_mapped_32);
478                         atomic_add_32(&libs_mapped_32, 1);
479                 }
480         }
481         ASSERT(*tmp == NULL);
482         *tmp = lvp;
483         mutex_exit(LIB_VA_HASH_MUTEX(index));
484         if (del) {
```



```
485                 ASSERT(del->lv_refcnt == 0);
486                 MOBJ_STAT_ADD(lib_va_add_delete);
487                 lib_va_free(del);
488         }
489         return (lvp);
490 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   45633 Mon Jul 28 07:44:50 2014
new/usr/src/uts/common/os/pool.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_


1300 /*
1301  * The meat of the bind operation.  The steps in pool_do_bind are:
1302  *
1303  * 1) Set PBWAIT in the p_poolflag of any process of interest, and add all
1304  *    such processes to an array.  For any interesting process that has
1305  *    threads inside the pool barrier set, increment a counter by the
1306  *    count of such threads.  Once PBWAIT is set on a process, that process
1307  *    will not disappear.
1308  *
1309  * 2) Wait for the counter from step 2 to drop to zero.  Any process which
1310  *    calls pool_barrier_exit() and notices that PBWAIT has been set on it
1311  *    will decrement that counter before going to sleep, and the process
1312  *    calling pool_barrier_exit() which does the final decrement will wake us.
1313  *
1314  * 3) For each interesting process, perform a calculation on it to see if
1315  *    the bind will actually succeed.  This uses the following three
1316  *    resource-set-specific functions:
1317  *
1318  *    - int set_bind_start(procs, pool)
1319  *
1320  *      Determine whether the given array of processes can be bound to the
1321  *      resource set associated with the given pool.  If it can, take and hold
1322  *      any locks necessary to ensure that the operation will succeed, and
1323  *      make any necessary reservations in the target resource set.  If it
1324  *      can't, return failure with no reservations made and no new locks held.
1325  *
1326  *    - void set_bind_abort(procs, pool)
1327  *
1328  *      set_bind_start() has completed successfully, but another resource set's
1329  *      set_bind_start() has failed, and we haven't begun the bind yet.  Undo
1330  *      any reservations made and drop any locks acquired by our
1331  *      set_bind_start().
1332  *
1333  *    - void set_bind_finish(void)
1334  *
1335  *      The bind has completed successfully.  The processes have been released,
1336  *      and the reservation acquired in set_bind_start() has been depleted as
1337  *      the processes have finished their bindings.  Drop any locks acquired by
1338  *      set_bind_start().
1339  *
1340  * 4) If we've decided that we can proceed with the bind, iterate through
1341  *    the list of interesting processes, grab the necessary locks (which
1342  *    may differ per resource set), perform the bind, and ASSERT that it
1343  *    succeeds.  Once a process has been rebound, it can be awakened.
1344  *
1345  * The operations from step 4 must be kept in sync with anything which might
1346  * cause the bind operations (e.g., cpupart_bind_thread()) to fail, and
1347  * are thus located in the same source files as the associated bind operations.
1348  */
1349 int
1350 pool_do_bind(pool_t *pool, idtype_t idtype, id_t id, int flags)
1351 {
1352         extern uint_t nproc;
1353         klwp_t *lwp = ttolwp(curthread);
1354         proc_t **pp, **procs;
1355         proc_t *prstart;
1356         int procs_count = 0;
1357         kproject_t *kpj;
```



```
1358         procset_t set;
1359         zone_t *zone;
1360         int procs_size;
1361         int rv = 0;
1362         proc_t *p;
1363         id_t cid = -1;

1365         ASSERT(pool_lock_held());

1367         if ((cid = pool_get_class(pool)) == POOL_CLASS_INVAL)
1368                 return (EINVAL);

1370         if (idtype == P_ZONEID) {
1371                 zone = zone_find_by_id(id);
1372                 if (zone == NULL)
1373                         return (ESRCH);
1374                 if (zone_status_get(zone) > ZONE_IS_RUNNING) {
1375                         zone_rele(zone);
1376                         return (EBUSY);
1377                 }
1378         }

1380         if (idtype == P_PROJID) {
1381                 kpj = project_hold_by_id(id, global_zone, PROJECT_HOLD_FIND);
1382                 if (kpj == NULL)
1383                         return (ESRCH);
1384                 mutex_enter(&kpj->kpj_poolbind);
1385         }

1387         if (idtype == P_PID) {
1388                 /*
1389                  * Fast-path for a single process case.
1390                  */
1391                 procs_size = 2; /* procs is NULL-terminated */
1392                 procs = kmem_zalloc(procs_size * sizeof (proc_t *), KM_SLEEP);
1393                 mutex_enter(&pidlock);
1394         } else {
1395                 /*
1396                  * We will need enough slots for proc_t pointers for as many as
1397                  * twice the number of currently running processes (assuming
1398                  * that each one could be in fork() creating a new child).
1399                  */
1400                 for (;;) {
1401                         procs_size = nproc * 2;
1402                         procs = kmem_zalloc(procs_size * sizeof (proc_t *),
1403                             KM_SLEEP);
1404                         mutex_enter(&pidlock);

1406                         if (nproc * 2 <= procs_size)
1407                                 break;
1408                         /*
1409                          * If nproc has changed, try again.
1410                          */
1411                         mutex_exit(&pidlock);
1412                         kmem_free(procs, procs_size * sizeof (proc_t *));
1413                 }
1414         }

1416         if (id == P_MYID)
1417                 id = getmyid(idtype);
1418         setprocset(&set, POP_AND, idtype, id, P_ALL, 0);

1420         /*
1421          * Do a first scan, and select target processes.
1422          */
1423         if (idtype == P_PID)
```

```
1424                 prstart = prfind(id);
1425         else
1426                 prstart = practive;
1427         for (p = prstart, pp = procs; p != NULL; p = p->p_next) {
1428                 mutex_enter(&p->p_lock);
1429                 /*
1430                  * Skip processes that don't match our (id, idtype) set or
1431                  * on the way of becoming zombies.  Skip kernel processes
1432                  * from the global zone.
1433                  */
1434                 if (procinset(p, &set) == 0 ||
1435                     p->p_poolflag & PEXITED ||
1436                     ((p->p_flag & SSYS) && INGLOBALZONE(p))) {
1437                         mutex_exit(&p->p_lock);
1438                         continue;
1439                 }
1440                 if (!INGLOBALZONE(p)) {
1441                         switch (idtype) {
1442                         case P_PID:
1443                         case P_TASKID:
1444                                 /*
1445                                  * Can't bind processes or tasks
1446                                  * in local zones to pools.
1447                                  */
1448                                 mutex_exit(&p->p_lock);
1449                                 mutex_exit(&pidlock);
1450                                 pool_bind_wakeall(procs);
1451                                 rv = EINVAL;
1452                                 goto out;
1453                         case P_PROJID:
1454                                 /*
1455                                  * Only projects in the global
1456                                  * zone can be rebound.
1457                                  */
1458                                 mutex_exit(&p->p_lock);
1459                                 continue;
1460                         case P_POOLID:
1461                                 /*
1462                                  * When rebinding pools, processes can be
1463                                  * in different zones.
1464                                  */
1465                                 break;
1466                         }
1467                 }

1469                 p->p_poolflag |= PBWAIT;
1470                 /*
1471                  * If some threads in this process are inside the pool
1472                  * barrier, add them to pool_barrier_count, as we have
1473                  * to wait for all of them to exit the barrier.
1474                  */
1475                 if (p->p_poolcnt > 0) {
1476                         mutex_enter(&pool_barrier_lock);
1477                         pool_barrier_count += p->p_poolcnt;
1478                         mutex_exit(&pool_barrier_lock);
1479                 }
1480                 ASSERT(pp < &procs[procs_size]);
1481                 *pp++ = p;
1482                 procs_count++;
1483                 mutex_exit(&p->p_lock);

1485                 /*
1486                  * We just found our process, so if we're only rebinding a
1487                  * single process then get out of this loop.
1488                  */
1489                 if (idtype == P_PID)
```



```
1490                         break;
1491         }
1492         *pp = NULL;     /* cap off the end of the array */
1493         mutex_exit(&pidlock);

1495         /*
1496          * Wait for relevant processes to stop before they try to enter the
1497          * barrier or at the exit from the barrier.  Make sure that we do
1498          * not get stopped here while we're holding pool_lock.  If we were
1499          * requested to stop, or got a signal then return EAGAIN to let the
1500          * library know that it needs to retry.
1501          */
1502         mutex_enter(&pool_barrier_lock);
1503         lwp->lwp_nostop++;
1504         while (pool_barrier_count > 0) {
1505                 (void) cv_wait_sig(&pool_barrier_cv, &pool_barrier_lock);
1506                 if (pool_barrier_count > 0) {
1507                         /*
1508                          * We either got a signal or were requested to
1509                          * stop by /proc.  Bail out with EAGAIN.  If we were
1510                          * requested to stop, we'll stop in post_syscall()
1511                          * on our way back to userland.
1512                          */
1513                         mutex_exit(&pool_barrier_lock);
1514                         pool_bind_wakeall(procs);
1515                         lwp->lwp_nostop--;
1516                         rv = EAGAIN;
1517                         goto out;
1518                 }
1519         }
1520         lwp->lwp_nostop--;
1521         mutex_exit(&pool_barrier_lock);

1523         if (idtype == P_PID) {
1524                 if ((p = *procs) == NULL)
1525                         goto skip;
1526                 mutex_enter(&p->p_lock);
1527                 /* Drop the process if it is exiting */
1528                 if (p->p_poolflag & PEXITED) {
1529                         mutex_exit(&p->p_lock);
1530                         pool_bind_wake(p);
1531                         procs_count--;
1532                 } else
1533                         mutex_exit(&p->p_lock);
1534                 goto skip;
1535         }

1537         /*
1538          * Do another run, and drop processes that were inside the barrier
1539          * in exit(), but when they have dropped to pool_barrier_exit
1540          * they have become of no interest to us.  Pick up child processes that
1541          * were created by fork() but didn't exist during our first scan.
1542          * Their parents are now stopped at pool_barrier_exit in cfork().
1543          */
1544         mutex_enter(&pidlock);
1545         for (pp = procs; (p = *pp) != NULL; pp++) {
1546                 mutex_enter(&p->p_lock);
1547                 if (p->p_poolflag & PEXITED) {
1548                         ASSERT(p->p_lwpcnt == 0);
1549                         mutex_exit(&p->p_lock);
1550                         pool_bind_wake(p);
1551                         /* flip w/last non-NULL slot */
1552                         *pp = procs[procs_count - 1];
1553                         procs[procs_count - 1] = NULL;
1554                         procs_count--;
1555                         pp--;                   /* try this slot again */
```

```
1556                    continue;
1557            } else
1558                    mutex_exit(&p->p_lock);
1559            /*
1560             * Look at the child and check if it should be rebound also.
1561             * We're holding pidlock, so it is safe to reference p_child.
1562             */
1563            if ((p = p->p_child) == NULL)
1564                    continue;

1566            mutex_enter(&p->p_lock);

1568            /*
1569             * Skip system processes and make sure that the child is in
1570             * the same task/project/pool/zone as the parent.
1571             */
1572            if ((!INGLOBALZONE(p) && idtype != P_ZONEID &&
1573                idtype != P_POOLID) || p->p_flag & SSYS) {
1574                    mutex_exit(&p->p_lock);
1575                    continue;
1576            }

1578            /*
1579             * If the child process has been already created by fork(), has
1580             * not exited, and has not been added to the list already,
1581             * then add it now.  We will hit this process again (since we
1582             * stick it at the end of the procs list) but it will ignored
1583             * because it will have the PBWAIT flag set.
1584             */
1585            if (procinset(p, &set) &&
1586                !(p->p_poolflag & PEXITED) &&
1587                !(p->p_poolflag & PBWAIT)) {
1588                    ASSERT(p->p_child == NULL); /* no child of a child */
1589                    procs[procs_count] = p;
1590                    procs[procs_count + 1] = NULL;
1591                    procs_count++;
1592                    p->p_poolflag |= PBWAIT;
1593            }
1594            mutex_exit(&p->p_lock);
1595    }
1596    mutex_exit(&pidlock);
1597 skip:
1598    /*
1599     * If there's no processes to rebind then return ESRCH, unless
1600     * we're associating a pool with new resource set, destroying it,
1601     * or binding a zone to a pool.
1602     */
1603    if (procs_count == 0) {
1604            if (idtype == P_POOLID || idtype == P_ZONEID)
1605                    rv = 0;
1606            else
1607                    rv = ESRCH;
1608            goto out;
1609    }

1611 #ifdef DEBUG
1612    /*
1613     * All processes in the array should have PBWAIT set, and none
1614     * should be in the critical section. Thus, although p_poolflag
1615     * and p_poolcnt are protected by p_lock, their ASSERTions below
1616     * should be stable without it. procinset(), however, ASSERTs that
1617     * the p_lock is held upon entry.
1618     */
1619    for (pp = procs; (p = *pp) != NULL; pp++) {
1620            int in_set;
```



```
1622            mutex_enter(&p->p_lock);
1623            in_set = procinset(p, &set);
1624            mutex_exit(&p->p_lock);

1626            ASSERT(in_set);
1627            ASSERT(p->p_poolflag & PBWAIT);
1628            ASSERT(p->p_poolcnt == 0);
1629    }
1630 #endif

1632    /*
1633     * Do the check if processor set rebinding is going to succeed or not.
1634     */
1635    if ((flags & POOL_BIND_PSET) &&
1636        (rv = pset_bind_start(procs, pool)) != 0) {
1637            pool_bind_wakeall(procs);
1638            goto out;
1639    }

1641    /*
1642     * At this point, all bind operations should succeed.
1643     */
1644    for (pp = procs; (p = *pp) != NULL; pp++) {
1645            if (flags & POOL_BIND_PSET) {
1646                    psetid_t psetid = pool->pool_pset->pset_id;
1647                    void *zonebuf;
1648                    void *projbuf;

1650                    /*
1651                     * Pre-allocate one buffer for FSS (per-project
1652                     * buffer for a new pset) in case if this is the
1653                     * first thread from its current project getting
1654                     * bound to this processor set.
1655                     */
1656                    projbuf = fss_allocbuf(FSS_ONE_BUF, FSS_ALLOC_PROJ);
1657                    zonebuf = fss_allocbuf(FSS_ONE_BUF, FSS_ALLOC_ZONE);

1659                    mutex_enter(&pidlock);
1660                    mutex_enter(&p->p_lock);
1661                    pool_pset_bind(p, psetid, projbuf, zonebuf);
1662                    mutex_exit(&p->p_lock);
1663                    mutex_exit(&pidlock);
1664                    /*
1665                     * Free buffers pre-allocated above if it
1666                     * wasn't actually used.
1667                     */
1668                    fss_freebuf(projbuf, FSS_ALLOC_PROJ);
1669                    fss_freebuf(zonebuf, FSS_ALLOC_ZONE);
1670            }
1671            /*
1672             * Now let's change the scheduling class of this
1673             * process if our target pool has it defined.
1674             */
1675            if (cid != POOL_CLASS_UNSET)
1676                    pool_change_class(p, cid);

1678            /*
1679             * It is safe to reference p_pool here without holding
1680             * p_lock because it cannot change underneath of us.
1681             * We're holding pool_lock here, so nobody else can be
1682             * moving this process between pools.  If process "p"
1683             * would be exiting, we're guaranteed that it would be blocked
1684             * at pool_barrier_enter() in exit().  Otherwise, it would've
1685             * been skipped by one of our scans of the practive list
1686             * as a process with PEXITED flag set.
1687             */
```

```
1688                 if (p->p_pool != pool) {
1689                         ASSERT(p->p_pool->pool_ref > 0);
1690                         atomic_dec_32(&p->p_pool->pool_ref);
1690                         atomic_add_32(&p->p_pool->pool_ref, -1);
1691                         p->p_pool = pool;
1692                         atomic_inc_32(&p->p_pool->pool_ref);
1692                         atomic_add_32(&p->p_pool->pool_ref, 1);
1693                 }
1694                 /*
1695                  * Okay, we've tortured this guy enough.
1696                  * Let this poor process go now.
1697                  */
1698                 pool_bind_wake(p);
1699         }
1700         if (flags & POOL_BIND_PSET)
1701                 pset_bind_finish();

1703 out:    switch (idtype) {
1704         case P_PROJID:
1705                 ASSERT(kpj != NULL);
1706                 mutex_exit(&kpj->kpj_poolbind);
1707                 project_rele(kpj);
1708                 break;
1709         case P_ZONEID:
1710                 if (rv == 0) {
1711                         mutex_enter(&cpu_lock);
1712                         zone_pool_set(zone, pool);
1713                         mutex_exit(&cpu_lock);
1714                 }
1715                 zone->zone_pool_mod = gethrtime();
1716                 zone_rele(zone);
1717                 break;
1718         }

1720         kmem_free(procs, procs_size * sizeof (proc_t *));
1721         ASSERT(pool_barrier_count == 0);
1722         return (rv);
1723 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    1709 Mon Jul 28 07:44:51 2014
new/usr/src/uts/common/os/refstr.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2003 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/systm.h>
#include <sys/param.h>
#include <sys/atomic.h>
#include <sys/kmem.h>
#include <sys/refstr.h>
#include <sys/refstr_impl.h>

refstr_t *
refstr_alloc(const char *str)
{
        refstr_t *rsp;
        size_t size = sizeof (rsp->rs_size) + sizeof (rsp->rs_refcnt) +
                strlen(str) + 1;

        ASSERT(size <= UINT32_MAX);
        rsp = kmem_alloc(size, KM_SLEEP);
        rsp->rs_size = (uint32_t)size;
        rsp->rs_refcnt = 1;
        (void) strcpy(rsp->rs_string, str);
        return (rsp);
}
```

_____***unchanged_portion_omitted_***

```

void
refstr_hold(refstr_t *rsp)
{
        atomic_inc_32(&rsp->rs_refcnt);
        atomic_add_32(&rsp->rs_refcnt, 1);
}

void
refstr_rele(refstr_t *rsp)
{
```



```
        if (atomic_dec_32_nv(&rsp->rs_refcnt) == 0)
        if (atomic_add_32_nv(&rsp->rs_refcnt, -1) == 0)
                kmem_free(rsp, (size_t)rsp->rs_size);
}
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    7940 Mon Jul 28 07:44:51 2014
new/usr/src/uts/common/os/sid.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

  79 void
  80 ksiddomain_hold(ksiddomain_t *kd)
  81 {
  82         atomic_inc_32(&kd->kd_ref);
  82         atomic_add_32(&kd->kd_ref, 1);
  83 }

  85 void
  86 ksiddomain_rele(ksiddomain_t *kd)
  87 {
  88         if (atomic_dec_32_nv(&kd->kd_ref) == 0) {
  88         if (atomic_add_32_nv(&kd->kd_ref, -1) == 0) {
  89                 /*
  90                  * The kd reference can only be incremented from 0 when
  91                  * the sid_lock is held; so we lock and then check need to
  92                  * check for 0 again.
  93                  */
  94                 mutex_enter(&sid_lock);
  95                 if (kd->kd_ref == 0) {
  96                         avl_remove(&sid_tree, kd);
  97                         kmem_free(kd->kd_name, kd->kd_len);
  98                         kmem_free(kd, sizeof (*kd));
  99                 }
 100                 mutex_exit(&sid_lock);
 101         }
 102 }

 104 void
 105 ksidlist_hold(ksidlist_t *ksl)
 106 {
 107         atomic_inc_32(&ksl->ksl_ref);
 107         atomic_add_32(&ksl->ksl_ref, 1);
 108 }

 110 void
 111 ksidlist_rele(ksidlist_t *ksl)
 112 {
 113         if (atomic_dec_32_nv(&ksl->ksl_ref) == 0) {
 113         if (atomic_add_32_nv(&ksl->ksl_ref, -1) == 0) {
 114                 int i;

 116                 for (i = 0; i < ksl->ksl_nsid; i++)
 117                         ksid_rele(&ksl->ksl_sids[i]);

 119                 kmem_free(ksl, KSIDLIST_MEM(ksl->ksl_nsid));
 120         }
 121 }
______unchanged_portion_omitted_

 261 void
 262 kcrsid_hold(credsid_t *kcr)
 263 {
 264         atomic_inc_32(&kcr->kr_ref);
 264         atomic_add_32(&kcr->kr_ref, 1);
 265 }

 267 void
 268 kcrsid_rele(credsid_t *kcr)
 269 {
```



```
 270         if (atomic_dec_32_nv(&kcr->kr_ref) == 0) {
 270         if (atomic_add_32_nv(&kcr->kr_ref, -1) == 0) {
 271                 ksid_index_t i;

 273                 for (i = 0; i < KSID_COUNT; i++)
 274                         ksid_rele(&kcr->kr_sidx[i]);

 276                 if (kcr->kr_sidlist != NULL)
 277                         ksidlist_rele(kcr->kr_sidlist);

 279                 kmem_free(kcr, sizeof (*kcr));
 280         }
 281 }
______unchanged_portion_omitted_
```

```
**********************************************************
  230046 Mon Jul 28 07:44:51 2014
new/usr/src/uts/common/os/strsubr.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

2642 perdm_t *
2643 hold_dm(struct streamtab *str, uint32_t qflag, uint32_t sqtype)
2644 {
2645         syncq_t *sq;
2646         perdm_t **pp;
2647         perdm_t *p;
2648         perdm_t *dmp;

2650         ASSERT(str != NULL);
2651         ASSERT(qflag & (QPERMOD | QMTOUTPERIM));

2653         rw_enter(&perdm_rwlock, RW_READER);
2654         for (p = perdm_list; p != NULL; p = p->dm_next) {
2655                 if (p->dm_str == str) { /* found one */
2656                         atomic_inc_32(&(p->dm_ref));
2656                         atomic_add_32(&(p->dm_ref), 1);
2657                         rw_exit(&perdm_rwlock);
2658                         return (p);
2659                 }
2660         }
2661         rw_exit(&perdm_rwlock);

2663         sq = new_syncq();
2664         if (qflag & QPERMOD) {
2665                 sq->sq_type = sqtype | SQ_PERMOD;
2666                 sq->sq_flags = sqtype & SQ_TYPES_IN_FLAGS;
2667         } else {
2668                 ASSERT(qflag & QMTOUTPERIM);
2669                 sq->sq_onext = sq->sq_oprev = sq;
2670         }

2672         dmp = kmem_alloc(sizeof (perdm_t), KM_SLEEP);
2673         dmp->dm_sq = sq;
2674         dmp->dm_str = str;
2675         dmp->dm_ref = 1;
2676         dmp->dm_next = NULL;

2678         rw_enter(&perdm_rwlock, RW_WRITER);
2679         for (pp = &perdm_list; (p = *pp) != NULL; pp = &(p->dm_next)) {
2680                 if (p->dm_str == str) { /* already present */
2681                         p->dm_ref++;
2682                         rw_exit(&perdm_rwlock);
2683                         free_syncq(sq);
2684                         kmem_free(dmp, sizeof (perdm_t));
2685                         return (p);
2686                 }
2687         }

2689         *pp = dmp;
2690         rw_exit(&perdm_rwlock);
2691         return (dmp);
2692 }
______unchanged_portion_omitted_
```

```
**********************************************************
  248841 Mon Jul 28 07:44:51 2014
new/usr/src/uts/common/os/sunddi.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

8070 /*
8071  * This procedure is provided as the general callback function when
8072  * umem_lockmemory calls as_add_callback for long term memory locking.
8073  * When as_unmap, as_setprot, or as_free encounter segments which have
8074  * locked memory, this callback will be invoked.
8075  */
8076 void
8077 umem_lock_undo(struct as *as, void *arg, uint_t event)
8078 {
8079         _NOTE(ARGUNUSED(as, event))
8080         struct ddi_umem_cookie *cp = (struct ddi_umem_cookie *)arg;

8082         /*
8083          * Call the cleanup function.  Decrement the cookie reference
8084          * count, if it goes to zero, return the memory for the cookie.
8085          * The i_ddi_umem_unlock for this cookie may or may not have been
8086          * called already.  It is the responsibility of the caller of
8087          * umem_lockmemory to handle the case of the cleanup routine
8088          * being called after a ddi_umem_unlock for the cookie
8089          * was called.
8090          */

8092         (*cp->callbacks.cbo_umem_lock_cleanup)((ddi_umem_cookie_t)cp);

8094         /* remove the cookie if reference goes to zero */
8095         if (atomic_dec_ulong_nv((ulong_t *)(&(cp->cook_refcnt))) == 0) {
8095         (atomic_add_long_nv((ulong_t *)(&(cp->cook_refcnt)), -1) == 0) {
8096                 kmem_free(cp, sizeof (struct ddi_umem_cookie));
8097         }
8098 }
______unchanged_portion_omitted_

8430 /*
8431  * Unlock the pages locked by ddi_umem_lock or umem_lockmemory and free
8432  * the cookie.  Called from i_ddi_umem_unlock_thread.
8433  */

8435 static void
8436 i_ddi_umem_unlock(struct ddi_umem_cookie *p)
8437 {
8438         uint_t  rc;

8440         /*
8441          * There is no way to determine whether a callback to
8442          * umem_lock_undo was registered via as_add_callback.
8443          * (i.e. umem_lockmemory was called with DDI_MEMLOCK_LONGTERM and
8444          * a valid callback function structure.)  as_delete_callback
8445          * is called to delete a possible registered callback.  If the
8446          * return from as_delete_callbacks is AS_CALLBACK_DELETED, it
8447          * indicates that there was a callback registered, and that is was
8448          * successfully deleted.  Thus, the cookie reference count
8449          * will never be decremented by umem_lock_undo.  Just return the
8450          * memory for the cookie, since both users of the cookie are done.
8451          * A return of AS_CALLBACK_NOTFOUND indicates a callback was
8452          * never registered.  A return of AS_CALLBACK_DELETE_DEFERRED
8453          * indicates that callback processing is taking place and, and
8454          * umem_lock_undo is, or will be, executing, and thus decrementing
8455          * the cookie reference count when it is complete.
8456          *
```



```
8457          * This needs to be done before as_pageunlock so that the
8458          * persistence of as is guaranteed because of the locked pages.
8459          *
8460          */
8461         rc = as_delete_callback(p->asp, p);


8464         /*
8465          * The proc->p_as will be stale if i_ddi_umem_unlock is called
8466          * after relvm is called so use p->asp.
8467          */
8468         as_pageunlock(p->asp, p->pparray, p->cvaddr, p->size, p->s_flags);

8470         /*
8471          * Now that we have unlocked the memory decrement the
8472          * *.max-locked-memory rctl
8473          */
8474         umem_decr_devlockmem(p);

8476         if (rc == AS_CALLBACK_DELETED) {
8477                 /* umem_lock_undo will not happen, return the cookie memory */
8478                 ASSERT(p->cook_refcnt == 2);
8479                 kmem_free(p, sizeof (struct ddi_umem_cookie));
8480         } else {
8481                 /*
8482                  * umem_undo_lock may happen if as_delete_callback returned
8483                  * AS_CALLBACK_DELETE_DEFERRED.  In that case, decrement the
8484                  * reference count, atomically, and return the cookie
8485                  * memory if the reference count goes to zero.  The only
8486                  * other value for rc is AS_CALLBACK_NOTFOUND.  In that
8487                  * case, just return the cookie memory.
8488                  */
8489                 if ((rc != AS_CALLBACK_DELETE_DEFERRED) ||
8490                     (atomic_dec_ulong_nv((ulong_t *)(&(p->cook_refcnt)))
8490                     (atomic_add_long_nv((ulong_t *)(&(p->cook_refcnt)), -1)
8491                     == 0)) {
8492                         kmem_free(p, sizeof (struct ddi_umem_cookie));
8493                 }
8494         }
8495 }
______unchanged_portion_omitted_
```

```
**********************************************************
   29574 Mon Jul 28 07:44:52 2014
new/usr/src/uts/common/os/task.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 353 /*
 354  * task_hold_by_id(), task_hold_by_id_zone()
 355  *
 356  * Overview
 357  *   task_hold_by_id() is used to take a reference on a task by its task id,
 358  *   supporting the various system call interfaces for obtaining resource data,
 359  *   delivering signals, and so forth.
 360  *
 361  * Return values
 362  *   Returns a pointer to the task_t with taskid_t id.  The task is returned
 363  *   with its hold count incremented by one.  Returns NULL if there
 364  *   is no task with the requested id.
 365  *
 366  * Caller's context
 367  *   Caller must not be holding task_hash_lock.  No restrictions on context.
 368  */
 369 task_t *
 370 task_hold_by_id_zone(taskid_t id, zoneid_t zoneid)
 371 {
 372         task_t *tk;

 374         mutex_enter(&task_hash_lock);
 375         if ((tk = task_find(id, zoneid)) != NULL)
 376                 atomic_inc_32(&tk->tk_hold_count);
 376                 atomic_add_32(&tk->tk_hold_count, 1);
 377         mutex_exit(&task_hash_lock);

 379         return (tk);
 380 }
______unchanged_portion_omitted_

 394 /*
 395  * void task_hold(task_t *)
 396  *
 397  * Overview
 398  *   task_hold() is used to take an additional reference to the given task.
 399  *
 400  * Return values
 401  *   None.
 402  *
 403  * Caller's context
 404  *   No restriction on context.
 405  */
 406 void
 407 task_hold(task_t *tk)
 408 {
 409         atomic_inc_32(&tk->tk_hold_count);
 409         atomic_add_32(&tk->tk_hold_count, 1);
 410 }
______unchanged_portion_omitted_
```

****
13921 Mon Jul 28 07:44:52 2014
new/usr/src/uts/common/os/tlabel.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
 118 /*
 119  * Put a hold on a label structure.
 120  */
 121 void
 122 label_hold(ts_label_t *lab)
 123 {
 124         atomic_inc_32(&lab->tsl_ref);
 124         atomic_add_32(&lab->tsl_ref, 1);
 125 }

 127 /*
 128  * Release previous hold on a label structure.  Free it if refcnt == 0.
 129  */
 130 void
 131 label_rele(ts_label_t *lab)
 132 {
 133         if (atomic_dec_32_nv(&lab->tsl_ref) == 0)
 133         if (atomic_add_32_nv(&lab->tsl_ref, -1) == 0)
 134                 kmem_cache_free(tslabel_cache, lab);
 135 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   54639 Mon Jul 28 07:44:52 2014
new/usr/src/uts/common/os/vmem.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1436 /*
1437  * Create an arena called name whose initial span is [base, base + size).
1438  * The arena's natural unit of currency is quantum, so vmem_alloc()
1439  * guarantees quantum-aligned results.  The arena may import new spans
1440  * by invoking afunc() on source, and may return those spans by invoking
1441  * ffunc() on source.  To make small allocations fast and scalable,
1442  * the arena offers high-performance caching for each integer multiple
1443  * of quantum up to qcache_max.
1444  */
1445 static vmem_t *
1446 vmem_create_common(const char *name, void *base, size_t size, size_t quantum,
1447         void *(*afunc)(vmem_t *, size_t, int),
1448         void (*ffunc)(vmem_t *, void *, size_t),
1449         vmem_t *source, size_t qcache_max, int vmflag)
1450 {
1451         int i;
1452         size_t nqcache;
1453         vmem_t *vmp, *cur, **vmpp;
1454         vmem_seg_t *vsp;
1455         vmem_freelist_t *vfp;
1456         uint32_t id = atomic_inc_32_nv(&vmem_id);
1456         uint32_t id = atomic_add_32_nv(&vmem_id, 1);

1458         if (vmem_vmem_arena != NULL) {
1459                 vmp = vmem_alloc(vmem_vmem_arena, sizeof (vmem_t),
1460                     vmflag & VM_KMFLAGS);
1461         } else {
1462                 ASSERT(id <= VMEM_INITIAL);
1463                 vmp = &vmem0[id - 1];
1464         }

1466         /* An identifier arena must inherit from another identifier arena */
1467         ASSERT(source == NULL || ((source->vm_cflags & VMC_IDENTIFIER) ==
1468             (vmflag & VMC_IDENTIFIER)));

1470         if (vmp == NULL)
1471                 return (NULL);
1472         bzero(vmp, sizeof (vmem_t));

1474         (void) snprintf(vmp->vm_name, VMEM_NAMELEN, "%s", name);
1475         mutex_init(&vmp->vm_lock, NULL, MUTEX_DEFAULT, NULL);
1476         cv_init(&vmp->vm_cv, NULL, CV_DEFAULT, NULL);
1477         vmp->vm_cflags = vmflag;
1478         vmflag &= VM_KMFLAGS;

1480         vmp->vm_quantum = quantum;
1481         vmp->vm_qshift = highbit(quantum) - 1;
1482         nqcache = MIN(qcache_max >> vmp->vm_qshift, VMEM_NQCACHE_MAX);

1484         for (i = 0; i <= VMEM_FREELISTS; i++) {
1485                 vfp = &vmp->vm_freelist[i];
1486                 vfp->vs_end = 1UL << i;
1487                 vfp->vs_knext = (vmem_seg_t *)(vfp + 1);
1488                 vfp->vs_kprev = (vmem_seg_t *)(vfp - 1);
1489         }

1491         vmp->vm_freelist[0].vs_kprev = NULL;
1492         vmp->vm_freelist[VMEM_FREELISTS].vs_knext = NULL;
1493         vmp->vm_freelist[VMEM_FREELISTS].vs_end = 0;
```



```
1494         vmp->vm_hash_table = vmp->vm_hash0;
1495         vmp->vm_hash_mask = VMEM_HASH_INITIAL - 1;
1496         vmp->vm_hash_shift = highbit(vmp->vm_hash_mask);

1498         vsp = &vmp->vm_seg0;
1499         vsp->vs_anext = vsp;
1500         vsp->vs_aprev = vsp;
1501         vsp->vs_knext = vsp;
1502         vsp->vs_kprev = vsp;
1503         vsp->vs_type = VMEM_SPAN;

1505         vsp = &vmp->vm_rotor;
1506         vsp->vs_type = VMEM_ROTOR;
1507         VMEM_INSERT(&vmp->vm_seg0, vsp, a);

1509         bcopy(&vmem_kstat_template, &vmp->vm_kstat, sizeof (vmem_kstat_t));

1511         vmp->vm_id = id;
1512         if (source != NULL)
1513                 vmp->vm_kstat.vk_source_id.value.ui32 = source->vm_id;
1514         vmp->vm_source = source;
1515         vmp->vm_source_alloc = afunc;
1516         vmp->vm_source_free = ffunc;

1518         /*
1519          * Some arenas (like vmem_metadata and kmem_metadata) cannot
1520          * use quantum caching to lower fragmentation.  Instead, we
1521          * increase their imports, giving a similar effect.
1522          */
1523         if (vmp->vm_cflags & VMC_NO_QCACHE) {
1524                 vmp->vm_min_import =
1525                     VMEM_QCACHE_SLABSIZE(nqcache << vmp->vm_qshift);
1526                 nqcache = 0;
1527         }

1529         if (nqcache != 0) {
1530                 ASSERT(!(vmflag & VM_NOSLEEP));
1531                 vmp->vm_qcache_max = nqcache << vmp->vm_qshift;
1532                 for (i = 0; i < nqcache; i++) {
1533                         char buf[VMEM_NAMELEN + 21];
1534                         (void) sprintf(buf, "%s_%lu", vmp->vm_name,
1535                             (i + 1) * quantum);
1536                         vmp->vm_qcache[i] = kmem_cache_create(buf,
1537                             (i + 1) * quantum, quantum, NULL, NULL, NULL,
1538                             NULL, vmp, KMC_QCACHE | KMC_NOTOUCH);
1539                 }
1540         }

1542         if ((vmp->vm_ksp = kstat_create("vmem", vmp->vm_id, vmp->vm_name,
1543             "vmem", KSTAT_TYPE_NAMED, sizeof (vmem_kstat_t) /
1544             sizeof (kstat_named_t), KSTAT_FLAG_VIRTUAL)) != NULL) {
1545                 vmp->vm_ksp->ks_data = &vmp->vm_kstat;
1546                 kstat_install(vmp->vm_ksp);
1547         }

1549         mutex_enter(&vmem_list_lock);
1550         vmpp = &vmem_list;
1551         while ((cur = *vmpp) != NULL)
1552                 vmpp = &cur->vm_next;
1553         *vmpp = vmp;
1554         mutex_exit(&vmem_list_lock);

1556         if (vmp->vm_cflags & VMC_POPULATOR) {
1557                 ASSERT(vmem_populators < VMEM_INITIAL);
1558                 vmem_populator[atomic_inc_32_nv(&vmem_populators) - 1] = vmp;
1558                 vmem_populator[atomic_add_32_nv(&vmem_populators, 1) - 1] = vmp;
```

```
1559                 mutex_enter(&vmp->vm_lock);
1560                 (void) vmem_populate(vmp, vmflag | VM_PANIC);
1561                 mutex_exit(&vmp->vm_lock);
1562         }

1564         if ((base || size) && vmem_add(vmp, base, size, vmflag) == NULL) {
1565                 vmem_destroy(vmp);
1566                 return (NULL);
1567         }

1569         return (vmp);
1570 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   62189 Mon Jul 28 07:44:53 2014
new/usr/src/uts/common/rpc/clnt_clts.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 242 static uint_t clts_rcstat_ndata =
 243         sizeof (clts_rcstat_tmpl) / sizeof (kstat_named_t);

 245 #define RCSTAT_INCR(s, x)                       \
 246         atomic_inc_64(&(s)->x.value.ui64)
 246         atomic_add_64(&(s)->x.value.ui64, 1)

 248 #define ptoh(p)         (&((p)->cku_client))
 249 #define htop(h)         ((struct cku_private *)((h)->cl_private))

 251 /*
 252  * Times to retry
 253  */
 254 #define SNDTRIES        4
 255 #define REFRESHES       2       /* authentication refreshes */

 257 /*
 258  * The following is used to determine the global default behavior for
 259  * CLTS when binding to a local port.
 260  *
 261  * If the value is set to 1 the default will be to select a reserved
 262  * (aka privileged) port, if the value is zero the default will be to
 263  * use non-reserved ports.  Users of kRPC may override this by using
 264  * CLNT_CONTROL() and CLSET_BINDRESVPORT.
 265  */
 266 static int clnt_clts_do_bindresvport = 1;

 268 #define BINDRESVPORT_RETRIES 5

 270 void
 271 clnt_clts_stats_init(zoneid_t zoneid, struct rpc_clts_client **statsp)
 272 {
 273         kstat_t *ksp;
 274         kstat_named_t *knp;

 276         knp = rpcstat_zone_init_common(zoneid, "unix", "rpc_clts_client",
 277             (const kstat_named_t *)&clts_rcstat_tmpl,
 278             sizeof (clts_rcstat_tmpl));
 279         /*
 280          * Backwards compatibility for old kstat clients
 281          */
 282         ksp = kstat_create_zone("unix", 0, "rpc_client", "rpc",
 283             KSTAT_TYPE_NAMED, clts_rcstat_ndata,
 284             KSTAT_FLAG_VIRTUAL | KSTAT_FLAG_WRITABLE, zoneid);
 285         if (ksp) {
 286                 ksp->ks_data = knp;
 287                 kstat_install(ksp);
 288         }
 289         *statsp = (struct rpc_clts_client *)knp;
 290 }
______unchanged_portion_omitted_
```

```
**********************************************************
  105307 Mon Jul 28 07:44:53 2014
new/usr/src/uts/common/rpc/clnt_cots.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 467 #define COTSRCSTAT_INCR(p, x)   \
 468         atomic_inc_64(&(p)->x.value.ui64)
 468         atomic_add_64(&(p)->x.value.ui64, 1)

 470 #define CLNT_MAX_CONNS  1       /* concurrent connections between clnt/srvr */
 471 int clnt_max_conns = CLNT_MAX_CONNS;

 473 #define CLNT_MIN_TIMEOUT        10      /* seconds to wait after we get a */
 474                                         /* connection reset */
 475 #define CLNT_MIN_CONNTIMEOUT    5       /* seconds to wait for a connection */


 478 int clnt_cots_min_tout = CLNT_MIN_TIMEOUT;
 479 int clnt_cots_min_conntout = CLNT_MIN_CONNTIMEOUT;

 481 /*
 482  * Limit the number of times we will attempt to receive a reply without
 483  * re-sending a response.
 484  */
 485 #define CLNT_MAXRECV_WITHOUT_RETRY      3
 486 uint_t clnt_cots_maxrecv        = CLNT_MAXRECV_WITHOUT_RETRY;

 488 uint_t *clnt_max_msg_sizep;
 489 void (*clnt_stop_idle)(queue_t *wq);

 491 #define ptoh(p)         (&((p)->cku_client))
 492 #define htop(h)         ((cku_private_t *)((h)->cl_private))

 494 /*
 495  * Times to retry
 496  */
 497 #define REFRESHES       2       /* authentication refreshes */

 499 /*
 500  * The following is used to determine the global default behavior for
 501  * COTS when binding to a local port.
 502  *
 503  * If the value is set to 1 the default will be to select a reserved
 504  * (aka privileged) port, if the value is zero the default will be to
 505  * use non-reserved ports.  Users of kRPC may override this by using
 506  * CLNT_CONTROL() and CLSET_BINDRESVPORT.
 507  */
 508 int clnt_cots_do_bindresvport = 1;

 510 static zone_key_t zone_cots_key;

 512 /*
 513  * Defaults TCP send and receive buffer size for RPC connections.
 514  * These values can be tuned by /etc/system.
 515  */
 516 int rpc_send_bufsz = 1024*1024;
 517 int rpc_recv_bufsz = 1024*1024;
 518 /*
 519  * To use system-wide default for TCP send and receive buffer size,
 520  * use /etc/system to set rpc_default_tcp_bufsz to 1:
 521  *
 522  * set rpcmod:rpc_default_tcp_bufsz=1
 523  */
 524 int rpc_default_tcp_bufsz = 0;
```



```
 526 /*
 527  * We need to do this after all kernel threads in the zone have exited.
 528  */
 529 /* ARGSUSED */
 530 static void
 531 clnt_zone_destroy(zoneid_t zoneid, void *unused)
 532 {
 533         struct cm_xprt **cmp;
 534         struct cm_xprt *cm_entry;
 535         struct cm_xprt *freelist = NULL;

 537         mutex_enter(&connmgr_lock);
 538         cmp = &cm_hd;
 539         while ((cm_entry = *cmp) != NULL) {
 540                 if (cm_entry->x_zoneid == zoneid) {
 541                         *cmp = cm_entry->x_next;
 542                         cm_entry->x_next = freelist;
 543                         freelist = cm_entry;
 544                 } else {
 545                         cmp = &cm_entry->x_next;
 546                 }
 547         }
 548         mutex_exit(&connmgr_lock);
 549         while ((cm_entry = freelist) != NULL) {
 550                 freelist = cm_entry->x_next;
 551                 connmgr_close(cm_entry);
 552         }
 553 }
______unchanged_portion_omitted_

2614 /*
2615  * Given an open stream, connect to the remote.  Returns true if connected,
2616  * false otherwise.
2617  */
2618 static bool_t
2619 connmgr_connect(
2620         struct cm_xprt          *cm_entry,
2621         queue_t                 *wq,
2622         struct netbuf           *addr,
2623         int                     addrfmly,
2624         calllist_t              *e,
2625         int                     *tidu_ptr,
2626         bool_t                  reconnect,
2627         const struct timeval    *waitp,
2628         bool_t                  nosignal,
2629         cred_t                  *cr)
2630 {
2631         mblk_t *mp;
2632         struct T_conn_req *tcr;
2633         struct T_info_ack *tinfo;
2634         int interrupted, error;
2635         int tidu_size, kstat_instance;

2637         /* if it's a reconnect, flush any lingering data messages */
2638         if (reconnect)
2639                 (void) putctl1(wq, M_FLUSH, FLUSHRW);

2641         /*
2642          * Note: if the receiver uses SCM_UCRED/getpeerucred the pid will
2643          * appear as -1.
2644          */
2645         mp = allocb_cred(sizeof (*tcr) + addr->len, cr, NOPID);
2646         if (mp == NULL) {
2647                 /*
2648                  * This is unfortunate, but we need to look up the stats for
```

```
2649                     * this zone to increment the "memory allocation failed"
2650                     * counter.  curproc->p_zone is safe since we're initiating a
2651                     * connection and not in some strange streams context.
2652                     */
2653                    struct rpcstat *rpcstat;

2655                    rpcstat = zone_getspecific(rpcstat_zone_key, rpc_zone());
2656                    ASSERT(rpcstat != NULL);

2658                    RPCLOG0(1, "connmgr_connect: cannot alloc mp for "
2659                        "sending conn request\n");
2660                    COTSRCSTAT_INCR(rpcstat->rpc_cots_client, rcnomem);
2661                    e->call_status = RPC_SYSTEMERROR;
2662                    e->call_reason = ENOSR;
2663                    return (FALSE);
2664            }

2666            /* Set TCP buffer size for RPC connections if needed */
2667            if (addrfmly == AF_INET || addrfmly == AF_INET6)
2668                    (void) connmgr_setbufsz(e, wq, cr);

2670            mp->b_datap->db_type = M_PROTO;
2671            tcr = (struct T_conn_req *)mp->b_rptr;
2672            bzero(tcr, sizeof (*tcr));
2673            tcr->PRIM_type = T_CONN_REQ;
2674            tcr->DEST_length = addr->len;
2675            tcr->DEST_offset = sizeof (struct T_conn_req);
2676            mp->b_wptr = mp->b_rptr + sizeof (*tcr);

2678            bcopy(addr->buf, mp->b_wptr, tcr->DEST_length);
2679            mp->b_wptr += tcr->DEST_length;

2681            RPCLOG(8, "connmgr_connect: sending conn request on queue "
2682                "%p", (void *)wq);
2683            RPCLOG(8, " call %p\n", (void *)wq);
2684            /*
2685             * We use the entry in the handle that is normally used for
2686             * waiting for RPC replies to wait for the connection accept.
2687             */
2688            if (clnt_dispatch_send(wq, mp, e, 0, 0) != RPC_SUCCESS) {
2689                    DTRACE_PROBE(krpc__e__connmgr__connect__cantsend);
2690                    freemsg(mp);
2691                    return (FALSE);
2692            }

2694            mutex_enter(&clnt_pending_lock);

2696            /*
2697             * We wait for the transport connection to be made, or an
2698             * indication that it could not be made.
2699             */
2700            interrupted = 0;

2702            /*
2703             * waitforack should have been called with T_OK_ACK, but the
2704             * present implementation needs to be passed T_INFO_ACK to
2705             * work correctly.
2706             */
2707            error = waitforack(e, T_INFO_ACK, waitp, nosignal);
2708            if (error == EINTR)
2709                    interrupted = 1;
2710            if (zone_status_get(curproc->p_zone) >= ZONE_IS_EMPTY) {
2711                    /*
2712                     * No time to lose; we essentially have been signaled to
2713                     * quit.
2714                     */
```



```
2715                    interrupted = 1;
2716            }
2717    #ifdef RPCDEBUG
2718            if (error == ETIME)
2719                    RPCLOG0(8, "connmgr_connect: giving up "
2720                        "on connection attempt; "
2721                        "clnt_dispatch notifyconn "
2722                        "diagnostic 'no one waiting for "
2723                        "connection' should not be "
2724                        "unexpected\n");
2725    #endif
2726            if (e->call_prev)
2727                    e->call_prev->call_next = e->call_next;
2728            else
2729                    clnt_pending = e->call_next;
2730            if (e->call_next)
2731                    e->call_next->call_prev = e->call_prev;
2732            mutex_exit(&clnt_pending_lock);

2734            if (e->call_status != RPC_SUCCESS || error != 0) {
2735                    if (interrupted)
2736                            e->call_status = RPC_INTR;
2737                    else if (error == ETIME)
2738                            e->call_status = RPC_TIMEDOUT;
2739                    else if (error == EPROTO) {
2740                            e->call_status = RPC_SYSTEMERROR;
2741                            e->call_reason = EPROTO;
2742                    }

2744                    RPCLOG(8, "connmgr_connect: can't connect, status: "
2745                        "%s\n", clnt_sperrno(e->call_status));

2747                    if (e->call_reply) {
2748                            freemsg(e->call_reply);
2749                            e->call_reply = NULL;
2750                    }

2752                    return (FALSE);
2753            }
2754            /*
2755             * The result of the "connection accept" is a T_info_ack
2756             * in the call_reply field.
2757             */
2758            ASSERT(e->call_reply != NULL);
2759            mp = e->call_reply;
2760            e->call_reply = NULL;
2761            tinfo = (struct T_info_ack *)mp->b_rptr;

2763            tidu_size = tinfo->TIDU_size;
2764            tidu_size -= (tidu_size % BYTES_PER_XDR_UNIT);
2765            if (tidu_size > COTS_DEFAULT_ALLOCSIZE || (tidu_size <= 0))
2766                    tidu_size = COTS_DEFAULT_ALLOCSIZE;
2767            *tidu_ptr = tidu_size;

2769            freemsg(mp);

2771            /*
2772             * Set up the pertinent options.  NODELAY is so the transport doesn't
2773             * buffer up RPC messages on either end.  This may not be valid for
2774             * all transports.  Failure to set this option is not cause to
2775             * bail out so we return success anyway.  Note that lack of NODELAY
2776             * or some other way to flush the message on both ends will cause
2777             * lots of retries and terrible performance.
2778             */
2779            if (addrfmly == AF_INET || addrfmly == AF_INET6) {
2780                    (void) connmgr_setopt(wq, IPPROTO_TCP, TCP_NODELAY, e, cr);
```

```
2781                 if (e->call_status == RPC_XPRTFAILED)
2782                         return (FALSE);
2783         }

2785         /*
2786          * Since we have a connection, we now need to figure out if
2787          * we need to create a kstat. If x_ksp is not NULL then we
2788          * are reusing a connection and so we do not need to create
2789          * another kstat -- lets just return.
2790          */
2791         if (cm_entry->x_ksp != NULL)
2792                 return (TRUE);

2794         /*
2795          * We need to increment rpc_kstat_instance atomically to prevent
2796          * two kstats being created with the same instance.
2797          */
2798         kstat_instance = atomic_inc_32_nv((uint32_t *)&rpc_kstat_instance);
2798         kstat_instance = atomic_add_32_nv((uint32_t *)&rpc_kstat_instance, 1);

2800         if ((cm_entry->x_ksp = kstat_create_zone("unix", kstat_instance,
2801             "rpc_cots_connections", "rpc", KSTAT_TYPE_NAMED,
2802             (uint_t)(sizeof (cm_kstat_xprt_t) / sizeof (kstat_named_t)),
2803             KSTAT_FLAG_VIRTUAL, cm_entry->x_zoneid)) == NULL) {
2804                 return (TRUE);
2805         }

2807         cm_entry->x_ksp->ks_lock = &connmgr_lock;
2808         cm_entry->x_ksp->ks_private = cm_entry;
2809         cm_entry->x_ksp->ks_data_size = ((INET6_ADDRSTRLEN * sizeof (char))
2810             + sizeof (cm_kstat_template));
2811         cm_entry->x_ksp->ks_data = kmem_alloc(cm_entry->x_ksp->ks_data_size,
2812             KM_SLEEP);
2813         bcopy(&cm_kstat_template, cm_entry->x_ksp->ks_data,
2814             cm_entry->x_ksp->ks_data_size);
2815         ((struct cm_kstat_xprt *)(cm_entry->x_ksp->ks_data))->
2816             x_server.value.str.addr.ptr =
2817             kmem_alloc(INET6_ADDRSTRLEN, KM_SLEEP);

2819         cm_entry->x_ksp->ks_update = conn_kstat_update;
2820         kstat_install(cm_entry->x_ksp);
2821         return (TRUE);
2822 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   26708 Mon Jul 28 07:44:53 2014
new/usr/src/uts/common/rpc/svc_clts.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 137 static uint_t clts_rsstat_ndata =
 138         sizeof (clts_rsstat_tmpl) / sizeof (kstat_named_t);

 140 #define CLONE2STATS(clone_xprt) \
 141         (struct rpc_clts_server *)(clone_xprt)->xp_master->xp_p2

 143 #define RSSTAT_INCR(stats, x)   \
 144         atomic_inc_64(&(stats)->x.value.ui64)
 144         atomic_add_64(&(stats)->x.value.ui64, 1)

 146 /*
 147  * Create a transport record.
 148  * The transport record, output buffer, and private data structure
 149  * are allocated.  The output buffer is serialized into using xdrmem.
 150  * There is one transport record per user process which implements a
 151  * set of services.
 152  */
 153 /* ARGSUSED */
 154 int
 155 svc_clts_kcreate(file_t *fp, uint_t sendsz, struct T_info_ack *tinfo,
 156     SVCMASTERXPRT **nxprt)
 157 {
 158         SVCMASTERXPRT *xprt;
 159         struct rpcstat *rpcstat;

 161         if (nxprt == NULL)
 162                 return (EINVAL);

 164         rpcstat = zone_getspecific(rpcstat_zone_key, curproc->p_zone);
 165         ASSERT(rpcstat != NULL);

 167         xprt = kmem_zalloc(sizeof (*xprt), KM_SLEEP);
 168         xprt->xp_lcladdr.buf = kmem_zalloc(sizeof (sin6_t), KM_SLEEP);
 169         xprt->xp_p2 = (caddr_t)rpcstat->rpc_clts_server;
 170         xprt->xp_ops = &svc_clts_op;
 171         xprt->xp_msg_size = tinfo->TSDU_size;

 173         xprt->xp_rtaddr.buf = NULL;
 174         xprt->xp_rtaddr.maxlen = tinfo->ADDR_size;
 175         xprt->xp_rtaddr.len = 0;

 177         *nxprt = xprt;

 179         return (0);
 180 }
______unchanged_portion_omitted_
```

```
**********************************************************
   26292 Mon Jul 28 07:44:53 2014
new/usr/src/uts/common/rpc/svc_cots.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 149 #define CLONE2STATS(clone_xprt) \
 150         ((struct cots_master_data *)(clone_xprt)->xp_master->xp_p2)->cmd_stats
 151 #define RSSTAT_INCR(s, x)       \
 152         atomic_inc_64(&(s)->x.value.ui64)
 152         atomic_add_64(&(s)->x.value.ui64, 1)

 154 /*
 155  * Pointer to a transport specific 'ready to receive' function in rpcmod
 156  * (set from rpcmod).
 157  */
 158 void    (*mir_start)(queue_t *);
 159 uint_t  *svc_max_msg_sizep;

 161 /*
 162  * the address size of the underlying transport can sometimes be
 163  * unknown (tinfo->ADDR_size == -1).  For this case, it is
 164  * necessary to figure out what the size is so the correct amount
 165  * of data is allocated.  This is an itterative process:
 166  *      1. take a good guess (use T_MINADDRSIZE)
 167  *      2. try it.
 168  *      3. if it works then everything is ok
 169  *      4. if the error is ENAMETOLONG, double the guess
 170  *      5. go back to step 2.
 171  */
 172 #define T_UNKNOWNADDRSIZE       (-1)
 173 #define T_MINADDRSIZE   32

 175 /*
 176  * Create a transport record.
 177  * The transport record, output buffer, and private data structure
 178  * are allocated.  The output buffer is serialized into using xdrmem.
 179  * There is one transport record per user process which implements a
 180  * set of services.
 181  */
 182 static kmutex_t cots_kcreate_lock;

 184 int
 185 svc_cots_kcreate(file_t *fp, uint_t max_msgsize, struct T_info_ack *tinfo,
 186     SVCMASTERXPRT **nxprt)
 187 {
 188         struct cots_master_data *cmd;
 189         int err, retval;
 190         SVCMASTERXPRT *xprt;
 191         struct rpcstat *rpcstat;
 192         struct T_addr_ack *ack_p;
 193         struct strioctl getaddr;

 195         if (nxprt == NULL)
 196                 return (EINVAL);

 198         rpcstat = zone_getspecific(rpcstat_zone_key, curproc->p_zone);
 199         ASSERT(rpcstat != NULL);

 201         xprt = kmem_zalloc(sizeof (SVCMASTERXPRT), KM_SLEEP);

 203         cmd = kmem_zalloc(sizeof (*cmd) + sizeof (*ack_p)
 204             + (2 * sizeof (sin6_t)), KM_SLEEP);

 206         ack_p = (struct T_addr_ack *)&cmd[1];
```



```
 208         if ((tinfo->TIDU_size > COTS_MAX_ALLOCSIZE) ||
 209             (tinfo->TIDU_size <= 0))
 210                 xprt->xp_msg_size = COTS_MAX_ALLOCSIZE;
 211         else {
 212                 xprt->xp_msg_size = tinfo->TIDU_size -
 213                     (tinfo->TIDU_size % BYTES_PER_XDR_UNIT);
 214         }

 216         xprt->xp_ops = &svc_cots_op;
 217         xprt->xp_p2 = (caddr_t)cmd;
 218         cmd->cmd_xprt_started = 0;
 219         cmd->cmd_stats = rpcstat->rpc_cots_server;

 221         getaddr.ic_cmd = TI_GETINFO;
 222         getaddr.ic_timout = -1;
 223         getaddr.ic_len = sizeof (*ack_p) + (2 * sizeof (sin6_t));
 224         getaddr.ic_dp = (char *)ack_p;
 225         ack_p->PRIM_type = T_ADDR_REQ;

 227         err = strioctl(fp->f_vnode, I_STR, (intptr_t)&getaddr,
 228             0, K_TO_K, CRED(), &retval);
 229         if (err) {
 230                 kmem_free(cmd, sizeof (*cmd) + sizeof (*ack_p) +
 231                     (2 * sizeof (sin6_t)));
 232                 kmem_free(xprt, sizeof (SVCMASTERXPRT));
 233                 return (err);
 234         }

 236         xprt->xp_rtaddr.maxlen = ack_p->REMADDR_length;
 237         xprt->xp_rtaddr.len = ack_p->REMADDR_length;
 238         cmd->cmd_src_addr = xprt->xp_rtaddr.buf =
 239             (char *)ack_p + ack_p->REMADDR_offset;

 241         xprt->xp_lcladdr.maxlen = ack_p->LOCADDR_length;
 242         xprt->xp_lcladdr.len = ack_p->LOCADDR_length;
 243         xprt->xp_lcladdr.buf = (char *)ack_p + ack_p->LOCADDR_offset;

 245         /*
 246          * If the current sanity check size in rpcmod is smaller
 247          * than the size needed for this xprt, then increase
 248          * the sanity check.
 249          */
 250         if (max_msgsize != 0 && svc_max_msg_sizep &&
 251             max_msgsize > *svc_max_msg_sizep) {

 253                 /* This check needs a lock */
 254                 mutex_enter(&cots_kcreate_lock);
 255                 if (svc_max_msg_sizep && max_msgsize > *svc_max_msg_sizep)
 256                         *svc_max_msg_sizep = max_msgsize;
 257                 mutex_exit(&cots_kcreate_lock);
 258         }

 260         *nxprt = xprt;

 262         return (0);
 263 }
______unchanged_portion_omitted_
```

```
**********************************************************
   36309 Mon Jul 28 07:44:53 2014
new/usr/src/uts/common/rpc/svc_rdma.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 182 kstat_named_t *rdmarsstat_ptr = (kstat_named_t *)&rdmarsstat;
 183 uint_t rdmarsstat_ndata = sizeof (rdmarsstat) / sizeof (kstat_named_t);

 185 #define RSSTAT_INCR(x)  atomic_inc_64(&rdmarsstat.x.value.ui64)
 185 #define RSSTAT_INCR(x)  atomic_add_64(&rdmarsstat.x.value.ui64, 1)
 186 /*
 187  * Create a transport record.
 188  * The transport record, output buffer, and private data structure
 189  * are allocated.  The output buffer is serialized into using xdrmem.
 190  * There is one transport record per user process which implements a
 191  * set of services.
 192  */
 193 /* ARGSUSED */
 194 int
 195 svc_rdma_kcreate(char *netid, SVC_CALLOUT_TABLE *sct, int id,
 196     rdma_xprt_group_t *started_xprts)
 197 {
 198         int error;
 199         SVCMASTERXPRT *xprt;
 200         struct rdma_data *rd;
 201         rdma_registry_t *rmod;
 202         rdma_xprt_record_t *xprt_rec;
 203         queue_t *q;
 204         /*
 205          * modload the RDMA plugins is not already done.
 206          */
 207         if (!rdma_modloaded) {
 208                 /*CONSTANTCONDITION*/
 209                 ASSERT(sizeof (struct clone_rdma_data) <= SVC_P2LEN);

 211                 mutex_enter(&rdma_modload_lock);
 212                 if (!rdma_modloaded) {
 213                         error = rdma_modload();
 214                 }
 215                 mutex_exit(&rdma_modload_lock);

 217                 if (error)
 218                         return (error);
 219         }

 221         /*
 222          * master_xprt_count is the count of master transport handles
 223          * that were successfully created and are ready to recieve for
 224          * RDMA based access.
 225          */
 226         error = 0;
 227         xprt_rec = NULL;
 228         rw_enter(&rdma_lock, RW_READER);
 229         if (rdma_mod_head == NULL) {
 230                 started_xprts->rtg_count = 0;
 231                 rw_exit(&rdma_lock);
 232                 if (rdma_dev_available)
 233                         return (EPROTONOSUPPORT);
 234                 else
 235                         return (ENODEV);
 236         }

 238         /*
 239          * If we have reached here, then atleast one RDMA plugin has loaded.
```



```
 240          * Create a master_xprt, make it start listenining on the device,
 241          * if an error is generated, record it, we might need to shut
 242          * the master_xprt.
 243          * SVC_START() calls svc_rdma_kstart which calls plugin binding
 244          * routines.
 245          */
 246         for (rmod = rdma_mod_head; rmod != NULL; rmod = rmod->r_next) {

 248                 /*
 249                  * One SVCMASTERXPRT per RDMA plugin.
 250                  */
 251                 xprt = kmem_zalloc(sizeof (*xprt), KM_SLEEP);
 252                 xprt->xp_ops = &rdma_svc_ops;
 253                 xprt->xp_sct = sct;
 254                 xprt->xp_type = T_RDMA;
 255                 mutex_init(&xprt->xp_req_lock, NULL, MUTEX_DEFAULT, NULL);
 256                 mutex_init(&xprt->xp_thread_lock, NULL, MUTEX_DEFAULT, NULL);
 257                 xprt->xp_req_head = (mblk_t *)0;
 258                 xprt->xp_req_tail = (mblk_t *)0;
 259                 xprt->xp_threads = 0;
 260                 xprt->xp_detached_threads = 0;

 262                 rd = kmem_zalloc(sizeof (*rd), KM_SLEEP);
 263                 xprt->xp_p2 = (caddr_t)rd;
 264                 rd->rd_xprt = xprt;
 265                 rd->r_mod = rmod->r_mod;

 267                 q = &rd->rd_data.q;
 268                 xprt->xp_wq = q;
 269                 q->q_ptr = &rd->rd_xprt;
 270                 xprt->xp_netid = NULL;

 272                 /*
 273                  * Each of the plugins will have their own Service ID
 274                  * to listener specific mapping, like port number for VI
 275                  * and service name for IB.
 276                  */
 277                 rd->rd_data.svcid = id;
 278                 error = svc_xprt_register(xprt, id);
 279                 if (error) {
 280                         DTRACE_PROBE(krpc__e__svcrdma__xprt__reg);
 281                         goto cleanup;
 282                 }

 284                 SVC_START(xprt);
 285                 if (!rd->rd_data.active) {
 286                         svc_xprt_unregister(xprt);
 287                         error = rd->rd_data.err_code;
 288                         goto cleanup;
 289                 }

 291                 /*
 292                  * This is set only when there is atleast one or more
 293                  * transports successfully created. We insert the pointer
 294                  * to the created RDMA master xprt into a separately maintained
 295                  * list. This way we can easily reference it later to cleanup,
 296                  * when NFS kRPC service pool is going away/unregistered.
 297                  */
 298                 started_xprts->rtg_count ++;
 299                 xprt_rec = kmem_alloc(sizeof (*xprt_rec), KM_SLEEP);
 300                 xprt_rec->rtr_xprt_ptr = xprt;
 301                 xprt_rec->rtr_next = started_xprts->rtg_listhead;
 302                 started_xprts->rtg_listhead = xprt_rec;
 303                 continue;
 304 cleanup:
 305                 SVC_DESTROY(xprt);
```

```
306                 if (error == RDMA_FAILED)
307                         error = EPROTONOSUPPORT;
308         }

310         rw_exit(&rdma_lock);

312         /*
313          * Don't return any error even if a single plugin was started
314          * successfully.
315          */
316         if (started_xprts->rtg_count == 0)
317                 return (error);
318         return (0);
319 }
```

_____***unchanged_portion_omitted_***

****
11702 Mon Jul 28 07:44:54 2014
new/usr/src/uts/common/sys/aggr_impl.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
 236 #define AGGR_GRP_REFHOLD(grp) {                 \
 237         atomic_inc_32(&(grp)->lg_refs); \
 237         atomic_add_32(&(grp)->lg_refs, 1);      \
 238         ASSERT((grp)->lg_refs != 0);            \
 239 }

 241 #define AGGR_GRP_REFRELE(grp) {                                 \
 242         ASSERT((grp)->lg_refs != 0);                            \
 243         membar_exit();                                          \
 244         if (atomic_dec_32_nv(&(grp)->lg_refs) == 0)             \
 244         if (atomic_add_32_nv(&(grp)->lg_refs, -1) == 0)         \
 245                 aggr_grp_free(grp);                             \
 246 }

 248 #define AGGR_PORT_REFHOLD(port) {               \
 249         atomic_inc_32(&(port)->lp_refs);        \
 249         atomic_add_32(&(port)->lp_refs, 1);     \
 250         ASSERT((port)->lp_refs != 0);           \
 251 }

 253 #define AGGR_PORT_REFRELE(port) {                               \
 254         ASSERT((port)->lp_refs != 0);                           \
 255         membar_exit();                                          \
 256         if (atomic_dec_32_nv(&(port)->lp_refs) == 0)    \
 256         if (atomic_add_32_nv(&(port)->lp_refs, -1) == 0)        \
 257                 aggr_port_free(port);                           \
 258 }
```

_____***unchanged_portion_omitted_***

****
53836 Mon Jul 28 07:44:54 2014
new/usr/src/uts/common/sys/crypto/impl.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
 393 /*
 394  * If a component has a reference to a kcf_policy_desc_t,
 395  * it REFHOLD()s. A new policy descriptor which is referenced only
 396  * by the policy table has a reference count of one.
 397  */
 398 #define KCF_POLICY_REFHOLD(desc) {              \
 399         atomic_inc_32(&(desc)->pd_refcnt);      \
 399         atomic_add_32(&(desc)->pd_refcnt, 1);   \
 400         ASSERT((desc)->pd_refcnt != 0);         \
 401 }

 403 /*
 404  * Releases a reference to a policy descriptor. When the last
 405  * reference is released, the descriptor is freed.
 406  */
 407 #define KCF_POLICY_REFRELE(desc) {                              \
 408         ASSERT((desc)->pd_refcnt != 0);                         \
 409         membar_exit();                                          \
 410         if (atomic_dec_32_nv(&(desc)->pd_refcnt) == 0)  \
 410         if (atomic_add_32_nv(&(desc)->pd_refcnt, -1) == 0)      \
 411                 kcf_policy_free_desc(desc);                     \
 412 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   15940 Mon Jul 28 07:44:54 2014
new/usr/src/uts/common/sys/crypto/sched_impl.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 103 /* Must be different from KM_SLEEP and KM_NOSLEEP */
 104 #define KCF_HOLD_PROV   0x1000

 106 #define IS_FG_SUPPORTED(mdesc, fg)              \
 107         (((mdesc)->pm_mech_info.cm_func_group_mask & (fg)) != 0)

 109 #define IS_PROVIDER_TRIED(pd, tlist)            \
 110         (tlist != NULL && is_in_triedlist(pd, tlist))

 112 #define IS_RECOVERABLE(error)                   \
 113         (error == CRYPTO_BUFFER_TOO_BIG ||      \
 114         error == CRYPTO_BUSY ||                 \
 115         error == CRYPTO_DEVICE_ERROR ||         \
 116         error == CRYPTO_DEVICE_MEMORY ||        \
 117         error == CRYPTO_KEY_SIZE_RANGE ||       \
 118         error == CRYPTO_NO_PERMISSION)

 120 #define KCF_ATOMIC_INCR(x)      atomic_inc_32(&(x))
 121 #define KCF_ATOMIC_DECR(x)      atomic_dec_32(&(x))
 120 #define KCF_ATOMIC_INCR(x)      atomic_add_32(&(x), 1)
 121 #define KCF_ATOMIC_DECR(x)      atomic_add_32(&(x), -1)

 123 /*
 124  * Node structure for synchronous requests.
 125  */
 126 typedef struct kcf_sreq_node {
 127         /* Should always be the first field in this structure */
 128         kcf_call_type_t         sn_type;
 129         /*
 130          * sn_cv and sr_lock are used to wait for the
 131          * operation to complete. sn_lock also protects
 132          * the sn_state field.
 133          */
 134         kcondvar_t              sn_cv;
 135         kmutex_t                sn_lock;
 136         kcf_req_status_t        sn_state;

 138         /*
 139          * Return value from the operation. This will be
 140          * one of the CRYPTO_* errors defined in common.h.
 141          */
 142         int                     sn_rv;

 144         /*
 145          * parameters to call the SPI with. This can be
 146          * a pointer as we know the caller context/stack stays.
 147          */
 148         struct kcf_req_params   *sn_params;

 150         /* Internal context for this request */
 151         struct kcf_context      *sn_context;

 153         /* Provider handling this request */
 154         kcf_provider_desc_t     *sn_provider;

 156         kcf_prov_cpu_t          *sn_mp;
 157 } kcf_sreq_node_t;
______unchanged_portion_omitted_
```



```
 212 #define KCF_AREQ_REFHOLD(areq) {                \
 213         atomic_inc_32(&(areq)->an_refcnt);      \
 213         atomic_add_32(&(areq)->an_refcnt, 1);   \
 214         ASSERT((areq)->an_refcnt != 0);         \
 215 }

 217 #define KCF_AREQ_REFRELE(areq) {                                \
 218         ASSERT((areq)->an_refcnt != 0);                         \
 219         membar_exit();                                          \
 220         if (atomic_dec_32_nv(&(areq)->an_refcnt) == 0)  \
 220         if (atomic_add_32_nv(&(areq)->an_refcnt, -1) == 0)      \
 221                 kcf_free_req(areq);                             \
 222 }
______unchanged_portion_omitted_

 311 /*
 312  * Bump up the reference count on the framework private context. A
 313  * global context or a request that references this structure should
 314  * do a hold.
 315  */
 316 #define KCF_CONTEXT_REFHOLD(ictx) {             \
 317         atomic_inc_32(&(ictx)->kc_refcnt);      \
 317         atomic_add_32(&(ictx)->kc_refcnt, 1);   \
 318         ASSERT((ictx)->kc_refcnt != 0);         \
 319 }

 321 /*
 322  * Decrement the reference count on the framework private context.
 323  * When the last reference is released, the framework private
 324  * context structure is freed along with the global context.
 325  */
 326 #define KCF_CONTEXT_REFRELE(ictx) {                             \
 327         ASSERT((ictx)->kc_refcnt != 0);                         \
 328         membar_exit();                                          \
 329         if (atomic_dec_32_nv(&(ictx)->kc_refcnt) == 0)  \
 329         if (atomic_add_32_nv(&(ictx)->kc_refcnt, -1) == 0)      \
 330                 kcf_free_context(ictx);                         \
 331 }
______unchanged_portion_omitted_
```

****
12082 Mon Jul 28 07:44:54 2014
new/usr/src/uts/common/sys/ib/clients/rdsv3/rdsv3_impl.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
348 /* rdsv3_impl.c */
349 void rdsv3_trans_init();
350 boolean_t rdsv3_capable_interface(struct lifreq *lifrp);
351 int rdsv3_do_ip_ioctl(ksocket_t so4, void **ipaddrs, int *size, int *nifs);
352 int rdsv3_do_ip_ioctl_old(ksocket_t so4, void **ipaddrs, int *size, int *nifs);
353 boolean_t rdsv3_isloopback(ipaddr_t addr);
354 void rdsv3_cancel_delayed_work(rdsv3_delayed_work_t *dwp);
355 void rdsv3_flush_workqueue(rdsv3_workqueue_struct_t *wq);
356 void rdsv3_queue_work(rdsv3_workqueue_struct_t *wq, rdsv3_work_t *wp);
357 void rdsv3_queue_delayed_work(rdsv3_workqueue_struct_t *wq,
358     rdsv3_delayed_work_t *dwp, uint_t delay);
359 struct rsock *rdsv3_sk_alloc();
360 void rdsv3_sock_init_data(struct rsock *sk);
361 void rdsv3_sock_exit_data(struct rsock *sk);
362 void rdsv3_destroy_task_workqueue(rdsv3_workqueue_struct_t *wq);
363 rdsv3_workqueue_struct_t *rdsv3_create_task_workqueue(char *name);
364 int rdsv3_conn_constructor(void *buf, void *arg, int kmflags);
365 void rdsv3_conn_destructor(void *buf, void *arg);
366 int rdsv3_conn_compare(const void *conn1, const void *conn2);
367 void rdsv3_loop_init();
368 int rdsv3_mr_compare(const void *mr1, const void *mr2);
369 int rdsv3_put_cmsg(struct nmsghdr *msg, int level, int type, size_t size,
370     void *payload);
371 int rdsv3_verify_bind_address(ipaddr_t addr);
372 uint16_t rdsv3_ip_fast_csum(void *buffer, size_t length);
373 uint_t rdsv3_ib_dma_map_sg(struct ib_device *dev, struct rdsv3_scatterlist
374         *scat, uint_t num);
375 void rdsv3_ib_dma_unmap_sg(ib_device_t *dev, struct rdsv3_scatterlist *scat,
376     uint_t num);
377 static inline void
378 rdsv3_sk_sock_hold(struct rsock *sk)
379 {
380         atomic_inc_32(&sk->sk_refcount);
380         atomic_add_32(&sk->sk_refcount, 1);
381 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   13740 Mon Jul 28 07:44:54 2014
new/usr/src/uts/common/syscall/corectl.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
  23  * Use is subject to license terms.
  24  */

  26 #pragma ident   "%Z%%M% %I%     %E% SMI"

  26 #include <sys/proc.h>
  27 #include <sys/systm.h>
  28 #include <sys/param.h>
  29 #include <sys/atomic.h>
  30 #include <sys/kmem.h>
  31 #include <sys/sysmacros.h>
  32 #include <sys/procset.h>
  33 #include <sys/corectl.h>
  34 #include <sys/zone.h>
  35 #include <sys/cmn_err.h>
  36 #include <sys/policy.h>

  38 /*
  39  * Core File Settings
  40  * ------------------
  41  *
  42  * A process's core file path and content live in separate reference-counted
  43  * structures. The corectl_content_t structure is fairly straightforward --
  44  * the only subtlety is that we only really _need_ the mutex on architectures
  45  * on which 64-bit memory operations are not atomic. The corectl_path_t
  46  * structure is slightly trickier in that it contains a refstr_t rather than
  47  * just a char * string. This is to allow consumers of the data in that
  48  * structure (the core dumping sub-system for example) to safely use the
  49  * string without holding any locks on it in light of updates.
  50  *
  51  * At system and zone boot, init_core() sets init(1M)'s core file path and
  52  * content to the same value as the fields core_default_path and
  53  * core_default_content respectively (for the global zone). All subsequent
  54  * children of init(1M) reference those same settings. During boot coreadm(1M)
  55  * is invoked with the -u option to update the system settings from
  56  * /etc/coreadm.conf. This has the effect of also changing the values in
  57  * core_default_path and core_default_content which updates the core file
  58  * settings for all processes in the zone.  Each zone has different default
  59  * settings; when processes enter a non-global zone, their core file path and
```



```
  60  * content are set to the zone's default path and content.
  61  *
  62  * Processes that have their core file settings explicitly overridden using
  63  * coreadm(1M) no longer reference core_default_path or core_default_content
  64  * so subsequent changes to the default will not affect them.
  65  */

  67 zone_key_t      core_zone_key;

  69 static int set_proc_info(pid_t pid, const char *path, core_content_t content);

  71 static corectl_content_t *
  72 corectl_content_alloc(core_content_t cc)
  73 {
  74         corectl_content_t *ccp;

  76         ccp = kmem_zalloc(sizeof (corectl_content_t), KM_SLEEP);
  77         ccp->ccc_content = cc;
  78         ccp->ccc_refcnt = 1;

  80         return (ccp);
  81 }
______unchanged_portion_omitted_

 103 void
 104 corectl_content_hold(corectl_content_t *ccp)
 105 {
 106         atomic_inc_32(&ccp->ccc_refcnt);
 108         atomic_add_32(&ccp->ccc_refcnt, 1);
 107 }

 109 void
 110 corectl_content_rele(corectl_content_t *ccp)
 111 {
 112         if (atomic_dec_32_nv(&ccp->ccc_refcnt) == 0)
 114         if (atomic_add_32_nv(&ccp->ccc_refcnt, -1) == 0)
 113                 kmem_free(ccp, sizeof (corectl_content_t));
 114 }
______unchanged_portion_omitted_

 152 void
 153 corectl_path_hold(corectl_path_t *ccp)
 154 {
 155         atomic_inc_32(&ccp->ccp_refcnt);
 157         atomic_add_32(&ccp->ccp_refcnt, 1);
 156 }

 158 void
 159 corectl_path_rele(corectl_path_t *ccp)
 160 {
 161         if (atomic_dec_32_nv(&ccp->ccp_refcnt) == 0) {
 163         if (atomic_add_32_nv(&ccp->ccp_refcnt, -1) == 0) {
 162                 refstr_rele(ccp->ccp_path);
 163                 kmem_free(ccp, sizeof (corectl_path_t));
 164         }
 165 }
______unchanged_portion_omitted_
```

```
**********************************************************
   88823 Mon Jul 28 07:44:55 2014
new/usr/src/uts/common/syscall/lwp_sobj.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 153 /*
 154  * Delete mappings from the lwpchan cache for pages that are being
 155  * unmapped by as_unmap().  Given a range of addresses, "start" to "end",
 156  * all mappings within the range are deleted from the lwpchan cache.
 157  */
 158 void
 159 lwpchan_delete_mapping(proc_t *p, caddr_t start, caddr_t end)
 160 {
 161         lwpchan_data_t *lcp;
 162         lwpchan_hashbucket_t *hashbucket;
 163         lwpchan_hashbucket_t *endbucket;
 164         lwpchan_entry_t *ent;
 165         lwpchan_entry_t **prev;
 166         caddr_t addr;

 168         mutex_enter(&p->p_lcp_lock);
 169         lcp = p->p_lcp;
 170         hashbucket = lcp->lwpchan_cache;
 171         endbucket = hashbucket + lcp->lwpchan_size;
 172         for (; hashbucket < endbucket; hashbucket++) {
 173                 if (hashbucket->lwpchan_chain == NULL)
 174                         continue;
 175                 mutex_enter(&hashbucket->lwpchan_lock);
 176                 prev = &hashbucket->lwpchan_chain;
 177                 /* check entire chain */
 178                 while ((ent = *prev) != NULL) {
 179                         addr = ent->lwpchan_addr;
 180                         if (start <= addr && addr < end) {
 181                                 *prev = ent->lwpchan_next;
 182                                 /*
 183                                  * We do this only for the obsolete type
 184                                  * USYNC_PROCESS_ROBUST.  Otherwise robust
 185                                  * locks do not draw ELOCKUNMAPPED or
 186                                  * EOWNERDEAD due to being unmapped.
 187                                  */
 188                                 if (ent->lwpchan_pool == LWPCHAN_MPPOOL &&
 189                                     (ent->lwpchan_type & USYNC_PROCESS_ROBUST))
 190                                         lwp_mutex_cleanup(ent, LOCK_UNMAPPED);
 191                                 /*
 192                                  * If there is a user-level robust lock
 193                                  * registration, mark it as invalid.
 194                                  */
 195                                 if ((addr = ent->lwpchan_uaddr) != NULL)
 196                                         lwp_mutex_unregister(addr);
 197                                 kmem_free(ent, sizeof (*ent));
 198                                 atomic_dec_32(&lcp->lwpchan_entries);
 198                                 atomic_add_32(&lcp->lwpchan_entries, -1);
 199                         } else {
 200                                 prev = &ent->lwpchan_next;
 201                         }
 202                 }
 203                 mutex_exit(&hashbucket->lwpchan_lock);
 204         }
 205         mutex_exit(&p->p_lcp_lock);
 206 }
______unchanged_portion_omitted_

 402 /*
 403  * Return the cached lwpchan mapping if cached, otherwise insert
```



```
 404  * a virtual address to lwpchan mapping into the cache.
 405  */
 406 static int
 407 lwpchan_get_mapping(struct as *as, caddr_t addr, caddr_t uaddr,
 408         int type, lwpchan_t *lwpchan, int pool)
 409 {
 410         proc_t *p = curproc;
 411         lwpchan_data_t *lcp;
 412         lwpchan_hashbucket_t *hashbucket;
 413         lwpchan_entry_t *ent;
 414         memid_t memid;
 415         uint_t count;
 416         uint_t bits;

 418 top:
 419         /* initialize the lwpchan cache, if necesary */
 420         if ((lcp = p->p_lcp) == NULL) {
 421                 lwpchan_alloc_cache(p, LWPCHAN_INITIAL_BITS);
 422                 goto top;
 423         }
 424         hashbucket = lwpchan_bucket(lcp, (uintptr_t)addr);
 425         mutex_enter(&hashbucket->lwpchan_lock);
 426         if (lcp != p->p_lcp) {
 427                 /* someone resized the lwpchan cache; start over */
 428                 mutex_exit(&hashbucket->lwpchan_lock);
 429                 goto top;
 430         }
 431         if (lwpchan_cache_mapping(addr, type, pool, lwpchan, hashbucket) == 0) {
 432                 /* it's in the cache */
 433                 mutex_exit(&hashbucket->lwpchan_lock);
 434                 return (1);
 435         }
 436         mutex_exit(&hashbucket->lwpchan_lock);
 437         if (as_getmemid(as, addr, &memid) != 0)
 438                 return (0);
 439         lwpchan->lc_wchan0 = (caddr_t)(uintptr_t)memid.val[0];
 440         lwpchan->lc_wchan = (caddr_t)(uintptr_t)memid.val[1];
 441         ent = kmem_alloc(sizeof (lwpchan_entry_t), KM_SLEEP);
 442         mutex_enter(&hashbucket->lwpchan_lock);
 443         if (lcp != p->p_lcp) {
 444                 /* someone resized the lwpchan cache; start over */
 445                 mutex_exit(&hashbucket->lwpchan_lock);
 446                 kmem_free(ent, sizeof (*ent));
 447                 goto top;
 448         }
 449         count = lwpchan_cache_mapping(addr, type, pool, lwpchan, hashbucket);
 450         if (count == 0) {
 451                 /* someone else added this entry to the cache */
 452                 mutex_exit(&hashbucket->lwpchan_lock);
 453                 kmem_free(ent, sizeof (*ent));
 454                 return (1);
 455         }
 456         if (count > lcp->lwpchan_bits + 2 && /* larger table, longer chains */
 457             (bits = lcp->lwpchan_bits) < LWPCHAN_MAX_BITS) {
 458                 /* hash chain too long; reallocate the hash table */
 459                 mutex_exit(&hashbucket->lwpchan_lock);
 460                 kmem_free(ent, sizeof (*ent));
 461                 lwpchan_alloc_cache(p, bits + 1);
 462                 goto top;
 463         }
 464         ent->lwpchan_addr = addr;
 465         ent->lwpchan_uaddr = uaddr;
 466         ent->lwpchan_type = (uint16_t)type;
 467         ent->lwpchan_pool = (uint16_t)pool;
 468         ent->lwpchan_lwpchan = *lwpchan;
 469         ent->lwpchan_next = hashbucket->lwpchan_chain;
```

```
 470         hashbucket->lwpchan_chain = ent;
 471         atomic_inc_32(&lcp->lwpchan_entries);
 471         atomic_add_32(&lcp->lwpchan_entries, 1);
 472         mutex_exit(&hashbucket->lwpchan_lock);
 473         return (1);
 474 }
______unchanged_portion_omitted_
```

```
**********************************************************
   36111 Mon Jul 28 07:44:55 2014
new/usr/src/uts/common/vm/page_retire.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 218 static kstat_t  *page_retire_ksp = NULL;

 220 #define PR_INCR_KSTAT(stat)     \
 221         atomic_inc_64(&(page_retire_kstat.stat.value.ui64))
 221         atomic_add_64(&(page_retire_kstat.stat.value.ui64), 1)
 222 #define PR_DECR_KSTAT(stat)     \
 223         atomic_dec_64(&(page_retire_kstat.stat.value.ui64))
 223         atomic_add_64(&(page_retire_kstat.stat.value.ui64), -1)

 225 #define PR_KSTAT_RETIRED_CE     (page_retire_kstat.pr_mce.value.ui64)
 226 #define PR_KSTAT_RETIRED_FMA    (page_retire_kstat.pr_fma.value.ui64)
 227 #define PR_KSTAT_RETIRED_NOTUE  (PR_KSTAT_RETIRED_CE + PR_KSTAT_RETIRED_FMA)
 228 #define PR_KSTAT_PENDING        (page_retire_kstat.pr_pending.value.ui64)
 229 #define PR_KSTAT_PENDING_KAS    (page_retire_kstat.pr_pending_kas.value.ui64)
 230 #define PR_KSTAT_EQFAIL         (page_retire_kstat.pr_enqueue_fail.value.ui64)
 231 #define PR_KSTAT_DQFAIL         (page_retire_kstat.pr_dequeue_fail.value.ui64)

 233 /*
 234  * page retire kstats to list all retired pages
 235  */
 236 static int pr_list_kstat_update(kstat_t *ksp, int rw);
 237 static int pr_list_kstat_snapshot(kstat_t *ksp, void *buf, int rw);
 238 kmutex_t pr_list_kstat_mutex;

 240 /*
 241  * Limit the number of multiple CE page retires.
 242  * The default is 0.1% of physmem, or 1 in 1000 pages. This is set in
 243  * basis points, where 100 basis points equals one percent.
 244  */
 245 #define MCE_BPT 10
 246 uint64_t        max_pages_retired_bps = MCE_BPT;
 247 #define PAGE_RETIRE_LIMIT       ((physmem * max_pages_retired_bps) / 10000)

 249 /*
 250  * Control over the verbosity of page retirement.
 251  *
 252  * When set to zero (the default), no messages will be printed.
 253  * When set to one, summary messages will be printed.
 254  * When set > one, all messages will be printed.
 255  *
 256  * A value of one will trigger detailed messages for retirement operations,
 257  * and is intended as a platform tunable for processors where FMA's DE does
 258  * not run (e.g., spitfire). Values > one are intended for debugging only.
 259  */
 260 int page_retire_messages = 0;

 262 /*
 263  * Control whether or not we return scrubbed UE pages to service.
 264  * By default we do not since FMA wants to run its diagnostics first
 265  * and then ask us to unretire the page if it passes. Non-FMA platforms
 266  * may set this to zero so we will only retire recidivist pages. It should
 267  * not be changed by the user.
 268  */
 269 int page_retire_first_ue = 1;

 271 /*
 272  * Master enable for page retire. This prevents a CE or UE early in boot
 273  * from trying to retire a page before page_retire_init() has finished
 274  * setting things up. This is internal only and is not a tunable!
```



```
 275  */
 276 static int pr_enable = 0;

 278 static void (*memscrub_notify_func)(uint64_t);

 280 #ifdef  DEBUG
 281 struct page_retire_debug {
 282         int prd_dup1;
 283         int prd_dup2;
 284         int prd_qdup;
 285         int prd_noaction;
 286         int prd_queued;
 287         int prd_notqueued;
 288         int prd_dequeue;
 289         int prd_top;
 290         int prd_locked;
 291         int prd_reloc;
 292         int prd_relocfail;
 293         int prd_mod;
 294         int prd_mod_late;
 295         int prd_kern;
 296         int prd_free;
 297         int prd_noreclaim;
 298         int prd_hashout;
 299         int prd_fma;
 300         int prd_uescrubbed;
 301         int prd_uenotscrubbed;
 302         int prd_mce;
 303         int prd_prlocked;
 304         int prd_prnotlocked;
 305         int prd_prretired;
 306         int prd_ulocked;
 307         int prd_unotretired;
 308         int prd_udestroy;
 309         int prd_uhashout;
 310         int prd_uunretired;
 311         int prd_unotlocked;
 312         int prd_checkhit;
 313         int prd_checkmiss_pend;
 314         int prd_checkmiss_noerr;
 315         int prd_tctop;
 316         int prd_tclocked;
 317         int prd_hunt;
 318         int prd_dohunt;
 319         int prd_earlyhunt;
 320         int prd_latehunt;
 321         int prd_nofreedemote;
 322         int prd_nodemote;
 323         int prd_demoted;
 324 } pr_debug;
______unchanged_portion_omitted_
```

****
45490 Mon Jul 28 07:44:55 2014
new/usr/src/uts/common/vm/seg_kmem.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____unchanged_portion_omitted_

```
1244 /*
1245  * This function is called to import new spans into the vmem arenas like
1246  * kmem_default_arena and kmem_oversize_arena. It first tries to import
1247  * spans from large page arena - kmem_lp_arena. In order to do this it might
1248  * have to "upgrade the requested size" to kmem_lp_arena quantum. If
1249  * it was not able to satisfy the upgraded request it then calls regular
1250  * segkmem_alloc() that satisfies the request by importing from "*vmp" arena
1251  */
1252 /*ARGSUSED*/
1253 void *
1254 segkmem_alloc_lp(vmem_t *vmp, size_t *sizep, size_t align, int vmflag)
1255 {
1256         size_t size;
1257         kthread_t *t = curthread;
1258         segkmem_lpcb_t *lpcb = &segkmem_lpcb;

1260         ASSERT(sizep != NULL);

1262         size = *sizep;

1264         if (lpcb->lp_uselp && !(t->t_flag & T_PANIC) &&
1265             !(vmflag & SEGKMEM_SHARELOCKED)) {

1267                 size_t kmemlp_qnt = segkmem_kmemlp_quantum;
1268                 size_t asize = P2ROUNDUP(size, kmemlp_qnt);
1269                 void  *addr = NULL;
1270                 ulong_t *lpthrtp = &lpcb->lp_throttle;
1271                 ulong_t lpthrt = *lpthrtp;
1272                 int     dowakeup = 0;
1273                 int     doalloc = 1;

1275                 ASSERT(kmem_lp_arena != NULL);
1276                 ASSERT(asize >= size);

1278                 if (lpthrt != 0) {
1279                         /* try to update the throttle value */
1280                         lpthrt = atomic_inc_ulong_nv(lpthrtp);
1280                         lpthrt = atomic_add_long_nv(lpthrtp, 1);
1281                         if (lpthrt >= segkmem_lpthrottle_max) {
1282                                 lpthrt = atomic_cas_ulong(lpthrtp, lpthrt,
1283                                     segkmem_lpthrottle_max / 4);
1284                         }

1286                         /*
1287                          * when we get above throttle start do an exponential
1288                          * backoff at trying large pages and reaping
1289                          */
1290                         if (lpthrt > segkmem_lpthrottle_start &&
1291                             (lpthrt & (lpthrt - 1))) {
1292                                 lpcb->allocs_throttled++;
1293                                 lpthrt--;
1294                                 if ((lpthrt & (lpthrt - 1)) == 0)
1295                                         kmem_reap();
1296                                 return (segkmem_alloc(vmp, size, vmflag));
1297                         }
1298                 }

1300                 if (!(vmflag & VM_NOSLEEP) &&
1301                     segkmem_heaplp_quantum >= (8 * kmemlp_qnt) &&
```



```
1302                     vmem_size(kmem_lp_arena, VMEM_FREE) <= kmemlp_qnt &&
1303                     asize < (segkmem_heaplp_quantum - kmemlp_qnt)) {

1305                         /*
1306                          * we are low on free memory in kmem_lp_arena
1307                          * we let only one guy to allocate heap_lp
1308                          * quantum size chunk that everybody is going to
1309                          * share
1310                          */
1311                         mutex_enter(&lpcb->lp_lock);

1313                         if (lpcb->lp_wait) {

1315                                 /* we are not the first one - wait */
1316                                 cv_wait(&lpcb->lp_cv, &lpcb->lp_lock);
1317                                 if (vmem_size(kmem_lp_arena, VMEM_FREE) <
1318                                     kmemlp_qnt)  {
1319                                         doalloc = 0;
1320                                 }
1321                         } else if (vmem_size(kmem_lp_arena, VMEM_FREE) <=
1322                             kmemlp_qnt) {

1324                                 /*
1325                                  * we are the first one, make sure we import
1326                                  * a large page
1327                                  */
1328                                 if (asize == kmemlp_qnt)
1329                                         asize += kmemlp_qnt;
1330                                 dowakeup = 1;
1331                                 lpcb->lp_wait = 1;
1332                         }

1334                         mutex_exit(&lpcb->lp_lock);
1335                 }

1337                 /*
1338                  * VM_ABORT flag prevents sleeps in vmem_xalloc when
1339                  * large pages are not available. In that case this allocation
1340                  * attempt will fail and we will retry allocation with small
1341                  * pages. We also do not want to panic if this allocation fails
1342                  * because we are going to retry.
1343                  */
1344                 if (doalloc) {
1345                         addr = vmem_alloc(kmem_lp_arena, asize,
1346                             (vmflag | VM_ABORT) & ~VM_PANIC);

1348                         if (dowakeup) {
1349                                 mutex_enter(&lpcb->lp_lock);
1350                                 ASSERT(lpcb->lp_wait != 0);
1351                                 lpcb->lp_wait = 0;
1352                                 cv_broadcast(&lpcb->lp_cv);
1353                                 mutex_exit(&lpcb->lp_lock);
1354                         }
1355                 }

1357                 if (addr != NULL) {
1358                         *sizep = asize;
1359                         *lpthrtp = 0;
1360                         return (addr);
1361                 }

1363                 if (vmflag & VM_NOSLEEP)
1364                         lpcb->nosleep_allocs_failed++;
1365                 else
1366                         lpcb->sleep_allocs_failed++;
1367                 lpcb->alloc_bytes_failed += size;
```

```
1369                 /* if large page throttling is not started yet do it */
1370                 if (segkmem_use_lpthrottle && lpthrt == 0) {
1371                         lpthrt = atomic_cas_ulong(lpthrtp, lpthrt, 1);
1372                 }
1373         }
1374         return (segkmem_alloc(vmp, size, vmflag));
1375 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   37170 Mon Jul 28 07:44:55 2014
new/usr/src/uts/common/vm/seg_kp.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 631 /*
 632  * Free the entire resource. segkp_unlock gets called with the start of the
 633  * mapped portion of the resource. The length is the size of the mapped
 634  * portion
 635  */
 636 static void
 637 segkp_release_internal(struct seg *seg, struct segkp_data *kpd, size_t len)
 638 {
 639         caddr_t         va;
 640         long            i;
 641         long            redzone;
 642         size_t          np;
 643         page_t          *pp;
 644         struct vnode    *vp;
 645         anoff_t         off;
 646         struct anon     *ap;
 647         pgcnt_t         segkpindex;

 649         ASSERT(kpd != NULL);
 650         ASSERT((kpd->kp_flags & KPD_HASAMP) == 0 || kpd->kp_cookie == -1);
 651         np = btop(len);

 653         /* Remove from active hash list */
 654         if (kpd->kp_cookie == -1) {
 655                 mutex_enter(&segkp_lock);
 656                 segkp_delete(seg, kpd);
 657                 mutex_exit(&segkp_lock);
 658         }

 660         /*
 661          * Precompute redzone page index.
 662          */
 663         redzone = -1;
 664         if (kpd->kp_flags & KPD_HASREDZONE)
 665                 redzone = KPD_REDZONE(kpd);


 668         va = kpd->kp_base;

 670         hat_unload(seg->s_as->a_hat, va, (np << PAGESHIFT),
 671             ((kpd->kp_flags & KPD_LOCKED) ? HAT_UNLOAD_UNLOCK : HAT_UNLOAD));
 672         /*
 673          * Free up those anon resources that are quiescent.
 674          */
 675         if (segkp_fromheap)
 676                 segkpindex = btop((uintptr_t)(va - kvseg.s_base));
 677         for (i = 0; i < np; i++, va += PAGESIZE) {

 679                 /*
 680                  * Clear the bit for this page from the bitmap.
 681                  */
 682                 if (segkp_fromheap) {
 683                         BT_ATOMIC_CLEAR(segkp_bitmap, segkpindex);
 684                         segkpindex++;
 685                 }

 687                 if (i == redzone)
 688                         continue;
 689                 if (kpd->kp_anon) {
```



```
 690                         /*
 691                          * Free up anon resources and destroy the
 692                          * associated pages.
 693                          *
 694                          * Release the lock if there is one. Have to get the
 695                          * page to do this, unfortunately.
 696                          */
 697                         if (kpd->kp_flags & KPD_LOCKED) {
 698                                 ap = anon_get_ptr(kpd->kp_anon,
 699                                     kpd->kp_anon_idx + i);
 700                                 swap_xlate(ap, &vp, &off);
 701                                 /* Find the shared-locked page. */
 702                                 pp = page_find(vp, (u_offset_t)off);
 703                                 if (pp == NULL) {
 704                                         panic("segkp_release: "
 705                                             "kp_anon: no page to unlock ");
 706                                         /*NOTREACHED*/
 707                                 }
 708                                 if (PP_ISRAF(pp))
 709                                         PP_CLRRAF(pp);

 711                                 page_unlock(pp);
 712                         }
 713                         if ((kpd->kp_flags & KPD_HASAMP) == 0) {
 714                                 anon_free(kpd->kp_anon, kpd->kp_anon_idx + i,
 715                                     PAGESIZE);
 716                                 anon_unresv_zone(PAGESIZE, NULL);
 717                                 atomic_dec_ulong(&anon_segkp_pages_resv);
 717                                 atomic_add_long(&anon_segkp_pages_resv,
 718                                     -1);
 718                         }
 719                         TRACE_5(TR_FAC_VM,
 720                             TR_ANON_SEGKP, "anon segkp:%p %p %lu %u %u",
 721                             kpd, va, PAGESIZE, 0, 0);
 722                 } else {
 723                         if (kpd->kp_flags & KPD_LOCKED) {
 724                                 pp = page_find(&kvp, (u_offset_t)(uintptr_t)va);
 725                                 if (pp == NULL) {
 726                                         panic("segkp_release: "
 727                                             "no page to unlock");
 728                                         /*NOTREACHED*/
 729                                 }
 730                                 if (PP_ISRAF(pp))
 731                                         PP_CLRRAF(pp);
 732                                 /*
 733                                  * We should just upgrade the lock here
 734                                  * but there is no upgrade that waits.
 735                                  */
 736                                 page_unlock(pp);
 737                         }
 738                         pp = page_lookup(&kvp, (u_offset_t)(uintptr_t)va,
 739                             SE_EXCL);
 740                         if (pp != NULL)
 741                                 page_destroy(pp, 0);
 742                 }
 743         }

 745         /* If locked, release physical memory reservation */
 746         if (kpd->kp_flags & KPD_LOCKED) {
 747                 pgcnt_t pages = btop(SEGKP_MAPLEN(kpd->kp_len, kpd->kp_flags));
 748                 if ((kpd->kp_flags & KPD_NO_ANON) == 0)
 749                         atomic_add_long(&anon_segkp_pages_locked, -pages);
 750                 page_unresv(pages);
 751         }

 753         vmem_free(SEGKP_VMEM(seg), kpd->kp_base, kpd->kp_len);
```

```
754         kmem_free(kpd, sizeof (struct segkp_data));
755 }

757 /*
758  * segkp_map_red() will check the current frame pointer against the
759  * stack base.  If the amount of stack remaining is questionable
760  * (less than red_minavail), then segkp_map_red() will map in the redzone
761  * and return 1.  Otherwise, it will return 0.  segkp_map_red() can
762  * _only_ be called when:
763  *
764  *   - it is safe to sleep on page_create_va().
765  *   - the caller is non-swappable.
766  *
767  * It is up to the caller to remember whether segkp_map_red() successfully
768  * mapped the redzone, and, if so, to call segkp_unmap_red() at a later
769  * time.  Note that the caller must _remain_ non-swappable until after
770  * calling segkp_unmap_red().
771  *
772  * Currently, this routine is only called from pagefault() (which necessarily
773  * satisfies the above conditions).
774  */
775 #if defined(STACK_GROWTH_DOWN)
776 int
777 segkp_map_red(void)
778 {
779         uintptr_t fp = STACK_BIAS + (uintptr_t)getfp();
780 #ifndef _LP64
781         caddr_t stkbase;
782 #endif

784         ASSERT(curthread->t_schedflag & TS_DONT_SWAP);

786         /*
787          * Optimize for the common case where we simply return.
788          */
789         if ((curthread->t_red_pp == NULL) &&
790             (fp - (uintptr_t)curthread->t_stkbase >= red_minavail))
791                 return (0);

793 #if defined(_LP64)
794         /*
795          * XXX  We probably need something better than this.
796          */
797         panic("kernel stack overflow");
798         /*NOTREACHED*/
799 #else /* _LP64 */
800         if (curthread->t_red_pp == NULL) {
801                 page_t *red_pp;
802                 struct seg kseg;

804                 caddr_t red_va = (caddr_t)
805                     (((uintptr_t)curthread->t_stkbase & (uintptr_t)PAGEMASK) -
806                     PAGESIZE);

808                 ASSERT(page_exists(&kvp, (u_offset_t)(uintptr_t)red_va) ==
809                     NULL);

811                 /*
812                  * Allocate the physical for the red page.
813                  */
814                 /*
815                  * No PG_NORELOC here to avoid waits. Unlikely to get
816                  * a relocate happening in the short time the page exists
817                  * and it will be OK anyway.
818                  */
```



```
820                 kseg.s_as = &kas;
821                 red_pp = page_create_va(&kvp, (u_offset_t)(uintptr_t)red_va,
822                     PAGESIZE, PG_WAIT | PG_EXCL, &kseg, red_va);
823                 ASSERT(red_pp != NULL);

825                 /*
826                  * So we now have a page to jam into the redzone...
827                  */
828                 page_io_unlock(red_pp);

830                 hat_memload(kas.a_hat, red_va, red_pp,
831                     (PROT_READ|PROT_WRITE), HAT_LOAD_LOCK);
832                 page_downgrade(red_pp);

834                 /*
835                  * The page is left SE_SHARED locked so we can hold on to
836                  * the page_t pointer.
837                  */
838                 curthread->t_red_pp = red_pp;

840                 atomic_inc_32(&red_nmapped);
841                 atomic_add_32(&red_nmapped, 1);
841                 while (fp - (uintptr_t)curthread->t_stkbase < red_closest) {
842                         (void) atomic_cas_32(&red_closest, red_closest,
843                             (uint32_t)(fp - (uintptr_t)curthread->t_stkbase));
844                 }
845                 return (1);
846         }

848         stkbase = (caddr_t)(((uintptr_t)curthread->t_stkbase &
849             (uintptr_t)PAGEMASK) - PAGESIZE);

851         atomic_inc_32(&red_ndoubles);
852         atomic_add_32(&red_ndoubles, 1);

853         if (fp - (uintptr_t)stkbase < RED_DEEP_THRESHOLD) {
854                 /*
855                  * Oh boy.  We're already deep within the mapped-in
856                  * redzone page, and the caller is trying to prepare
857                  * for a deep stack run.  We're running without a
858                  * redzone right now:  if the caller plows off the
859                  * end of the stack, it'll plow another thread or
860                  * LWP structure.  That situation could result in
861                  * a very hard-to-debug panic, so, in the spirit of
862                  * recording the name of one's killer in one's own
863                  * blood, we're going to record hrestime and the calling
864                  * thread.
865                  */
866                 red_deep_hires = hrestime.tv_nsec;
867                 red_deep_thread = curthread;
868         }

870         /*
871          * If this is a DEBUG kernel, and we've run too deep for comfort, toss.
872          */
873         ASSERT(fp - (uintptr_t)stkbase >= RED_DEEP_THRESHOLD);
874         return (0);
875 #endif /* _LP64 */
876 }
```

*unchanged_portion_omitted_*

```
1422 #include <sys/mem_config.h>

1424 /*ARGSUSED*/
1425 static void
1426 segkp_mem_config_post_add(void *arg, pgcnt_t delta_pages)
```

```
1427 {}

1429 /*
1430  * During memory delete, turn off caches so that pages are not held.
1431  * A better solution may be to unlock the pages while they are
1432  * in the cache so that they may be collected naturally.
1433  */

1435 /*ARGSUSED*/
1436 static int
1437 segkp_mem_config_pre_del(void *arg, pgcnt_t delta_pages)
1438 {
1439         atomic_inc_32(&segkp_indel);
1440         atomic_add_32(&segkp_indel, 1);
1440         segkp_cache_free();
1441         return (0);
1442 }

1444 /*ARGSUSED*/
1445 static void
1446 segkp_mem_config_post_del(void *arg, pgcnt_t delta_pages, int cancelled)
1447 {
1448         atomic_dec_32(&segkp_indel);
1449         atomic_add_32(&segkp_indel, -1);
1449 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   83927 Mon Jul 28 07:44:56 2014
new/usr/src/uts/common/vm/seg_spt.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 810 /*
 811  * DISM only.
 812  * Return locked pages over a given range.
 813  *
 814  * We will cache all DISM locked pages and save the pplist for the
 815  * entire segment in the ppa field of the underlying DISM segment structure.
 816  * Later, during a call to segspt_reclaim() we will use this ppa array
 817  * to page_unlock() all of the pages and then we will free this ppa list.
 818  */
 819 /*ARGSUSED*/
 820 static int
 821 segspt_dismpagelock(struct seg *seg, caddr_t addr, size_t len,
 822     struct page ***ppp, enum lock_type type, enum seg_rw rw)
 823 {
 824         struct  shm_data *shmd = (struct shm_data *)seg->s_data;
 825         struct  seg     *sptseg = shmd->shm_sptseg;
 826         struct  spt_data *sptd = sptseg->s_data;
 827         pgcnt_t pg_idx, npages, tot_npages, npgs;
 828         struct  page **pplist, **pl, **ppa, *pp;
 829         struct  anon_map *amp;
 830         spgcnt_t        an_idx;
 831         int     ret = ENOTSUP;
 832         uint_t  pl_built = 0;
 833         struct  anon *ap;
 834         struct  vnode *vp;
 835         u_offset_t off;
 836         pgcnt_t claim_availrmem = 0;
 837         uint_t  szc;

 839         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
 840         ASSERT(type == L_PAGELOCK || type == L_PAGEUNLOCK);

 842         /*
 843          * We want to lock/unlock the entire ISM segment. Therefore,
 844          * we will be using the underlying sptseg and it's base address
 845          * and length for the caching arguments.
 846          */
 847         ASSERT(sptseg);
 848         ASSERT(sptd);

 850         pg_idx = seg_page(seg, addr);
 851         npages = btopr(len);

 853         /*
 854          * check if the request is larger than number of pages covered
 855          * by amp
 856          */
 857         if (pg_idx + npages > btopr(sptd->spt_amp->size)) {
 858                 *ppp = NULL;
 859                 return (ENOTSUP);
 860         }

 862         if (type == L_PAGEUNLOCK) {
 863                 ASSERT(sptd->spt_ppa != NULL);

 865                 seg_pinactive(seg, NULL, seg->s_base, sptd->spt_amp->size,
 866                     sptd->spt_ppa, S_WRITE, SEGP_FORCE_WIRED, segspt_reclaim);

 868                 /*
```



```
 869                  * If someone is blocked while unmapping, we purge
 870                  * segment page cache and thus reclaim pplist synchronously
 871                  * without waiting for seg_pasync_thread. This speeds up
 872                  * unmapping in cases where munmap(2) is called, while
 873                  * raw async i/o is still in progress or where a thread
 874                  * exits on data fault in a multithreaded application.
 875                  */
 876                 if ((sptd->spt_flags & DISM_PPA_CHANGED) ||
 877                     (AS_ISUNMAPWAIT(seg->s_as) &&
 878                     shmd->shm_softlockcnt > 0)) {
 879                         segspt_purge(seg);
 880                 }
 881                 return (0);
 882         }

 884         /* The L_PAGELOCK case ... */

 886         if (sptd->spt_flags & DISM_PPA_CHANGED) {
 887                 segspt_purge(seg);
 888                 /*
 889                  * for DISM ppa needs to be rebuild since
 890                  * number of locked pages could be changed
 891                  */
 892                 *ppp = NULL;
 893                 return (ENOTSUP);
 894         }

 896         /*
 897          * First try to find pages in segment page cache, without
 898          * holding the segment lock.
 899          */
 900         pplist = seg_plookup(seg, NULL, seg->s_base, sptd->spt_amp->size,
 901             S_WRITE, SEGP_FORCE_WIRED);
 902         if (pplist != NULL) {
 903                 ASSERT(sptd->spt_ppa != NULL);
 904                 ASSERT(sptd->spt_ppa == pplist);
 905                 ppa = sptd->spt_ppa;
 906                 for (an_idx = pg_idx; an_idx < pg_idx + npages; ) {
 907                         if (ppa[an_idx] == NULL) {
 908                                 seg_pinactive(seg, NULL, seg->s_base,
 909                                     sptd->spt_amp->size, ppa,
 910                                     S_WRITE, SEGP_FORCE_WIRED, segspt_reclaim);
 911                                 *ppp = NULL;
 912                                 return (ENOTSUP);
 913                         }
 914                         if ((szc = ppa[an_idx]->p_szc) != 0) {
 915                                 npgs = page_get_pagecnt(szc);
 916                                 an_idx = P2ROUNDUP(an_idx + 1, npgs);
 917                         } else {
 918                                 an_idx++;
 919                         }
 920                 }
 921                 /*
 922                  * Since we cache the entire DISM segment, we want to
 923                  * set ppp to point to the first slot that corresponds
 924                  * to the requested addr, i.e. pg_idx.
 925                  */
 926                 *ppp = &(sptd->spt_ppa[pg_idx]);
 927                 return (0);
 928         }

 930         mutex_enter(&sptd->spt_lock);
 931         /*
 932          * try to find pages in segment page cache with mutex
 933          */
 934         pplist = seg_plookup(seg, NULL, seg->s_base, sptd->spt_amp->size,
```

```
935             S_WRITE, SEGP_FORCE_WIRED);
936         if (pplist != NULL) {
937                 ASSERT(sptd->spt_ppa != NULL);
938                 ASSERT(sptd->spt_ppa == pplist);
939                 ppa = sptd->spt_ppa;
940                 for (an_idx = pg_idx; an_idx < pg_idx + npages; ) {
941                         if (ppa[an_idx] == NULL) {
942                                 mutex_exit(&sptd->spt_lock);
943                                 seg_pinactive(seg, NULL, seg->s_base,
944                                     sptd->spt_amp->size, ppa,
945                                     S_WRITE, SEGP_FORCE_WIRED, segspt_reclaim);
946                                 *ppp = NULL;
947                                 return (ENOTSUP);
948                         }
949                         if ((szc = ppa[an_idx]->p_szc) != 0) {
950                                 npgs = page_get_pagecnt(szc);
951                                 an_idx = P2ROUNDUP(an_idx + 1, npgs);
952                         } else {
953                                 an_idx++;
954                         }
955                 }
956                 /*
957                  * Since we cache the entire DISM segment, we want to
958                  * set ppp to point to the first slot that corresponds
959                  * to the requested addr, i.e. pg_idx.
960                  */
961                 mutex_exit(&sptd->spt_lock);
962                 *ppp = &(sptd->spt_ppa[pg_idx]);
963                 return (0);
964         }
965         if (seg_pinsert_check(seg, NULL, seg->s_base, sptd->spt_amp->size,
966             SEGP_FORCE_WIRED) == SEGP_FAIL) {
967                 mutex_exit(&sptd->spt_lock);
968                 *ppp = NULL;
969                 return (ENOTSUP);
970         }

972         /*
973          * No need to worry about protections because DISM pages are always rw.
974          */
975         pl = pplist = NULL;
976         amp = sptd->spt_amp;

978         /*
979          * Do we need to build the ppa array?
980          */
981         if (sptd->spt_ppa == NULL) {
982                 pgcnt_t lpg_cnt = 0;

984                 pl_built = 1;
985                 tot_npages = btopr(sptd->spt_amp->size);

987                 ASSERT(sptd->spt_pcachecnt == 0);
988                 pplist = kmem_zalloc(sizeof (page_t *) * tot_npages, KM_SLEEP);
989                 pl = pplist;

991                 ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
992                 for (an_idx = 0; an_idx < tot_npages; ) {
993                         ap = anon_get_ptr(amp->ahp, an_idx);
994                         /*
995                          * Cache only mlocked pages. For large pages
996                          * if one (constituent) page is mlocked
997                          * all pages for that large page
998                          * are cached also. This is for quick
999                          * lookups of ppa array;
1000                         */
```



```
1001                        if ((ap != NULL) && (lpg_cnt != 0 ||
1002                            (sptd->spt_ppa_lckcnt[an_idx] != 0))) {

1004                                swap_xlate(ap, &vp, &off);
1005                                pp = page_lookup(vp, off, SE_SHARED);
1006                                ASSERT(pp != NULL);
1007                                if (lpg_cnt == 0) {
1008                                        lpg_cnt++;
1009                                        /*
1010                                         * For a small page, we are done --
1011                                         * lpg_count is reset to 0 below.
1012                                         *
1013                                         * For a large page, we are guaranteed
1014                                         * to find the anon structures of all
1015                                         * constituent pages and a non-zero
1016                                         * lpg_cnt ensures that we don't test
1017                                         * for mlock for these. We are done
1018                                         * when lpg_count reaches (npgs + 1).
1019                                         * If we are not the first constituent
1020                                         * page, restart at the first one.
1021                                         */
1022                                        npgs = page_get_pagecnt(pp->p_szc);
1023                                        if (!IS_P2ALIGNED(an_idx, npgs)) {
1024                                                an_idx = P2ALIGN(an_idx, npgs);
1025                                                page_unlock(pp);
1026                                                continue;
1027                                        }
1028                                }
1029                                if (++lpg_cnt > npgs)
1030                                        lpg_cnt = 0;

1032                                /*
1033                                 * availrmem is decremented only
1034                                 * for unlocked pages
1035                                 */
1036                                if (sptd->spt_ppa_lckcnt[an_idx] == 0)
1037                                        claim_availrmem++;
1038                                pplist[an_idx] = pp;
1039                        }
1040                        an_idx++;
1041                }
1042                ANON_LOCK_EXIT(&amp->a_rwlock);

1044                if (claim_availrmem) {
1045                        mutex_enter(&freemem_lock);
1046                        if (availrmem < tune.t_minarmem + claim_availrmem) {
1047                                mutex_exit(&freemem_lock);
1048                                ret = ENOTSUP;
1049                                claim_availrmem = 0;
1050                                goto insert_fail;
1051                        } else {
1052                                availrmem -= claim_availrmem;
1053                        }
1054                        mutex_exit(&freemem_lock);
1055                }

1057                sptd->spt_ppa = pl;
1058        } else {
1059                /*
1060                 * We already have a valid ppa[].
1061                 */
1062                pl = sptd->spt_ppa;
1063        }

1065        ASSERT(pl != NULL);
```

```
1067         ret = seg_pinsert(seg, NULL, seg->s_base, sptd->spt_amp->size,
1068             sptd->spt_amp->size, pl, S_WRITE, SEGP_FORCE_WIRED,
1069             segspt_reclaim);
1070         if (ret == SEGP_FAIL) {
1071                 /*
1072                  * seg_pinsert failed. We return
1073                  * ENOTSUP, so that the as_pagelock() code will
1074                  * then try the slower F_SOFTLOCK path.
1075                  */
1076                 if (pl_built) {
1077                         /*
1078                          * No one else has referenced the ppa[].
1079                          * We created it and we need to destroy it.
1080                          */
1081                         sptd->spt_ppa = NULL;
1082                 }
1083                 ret = ENOTSUP;
1084                 goto insert_fail;
1085         }

1087         /*
1088          * In either case, we increment softlockcnt on the 'real' segment.
1089          */
1090         sptd->spt_pcachecnt++;
1091         atomic_inc_ulong((ulong_t *)(&(shmd->shm_softlockcnt)));
1091         atomic_add_long((ulong_t *)(&(shmd->shm_softlockcnt)), 1);

1093         ppa = sptd->spt_ppa;
1094         for (an_idx = pg_idx; an_idx < pg_idx + npages; ) {
1095                 if (ppa[an_idx] == NULL) {
1096                         mutex_exit(&sptd->spt_lock);
1097                         seg_pinactive(seg, NULL, seg->s_base,
1098                             sptd->spt_amp->size,
1099                             pl, S_WRITE, SEGP_FORCE_WIRED, segspt_reclaim);
1100                         *ppp = NULL;
1101                         return (ENOTSUP);
1102                 }
1103                 if ((szc = ppa[an_idx]->p_szc) != 0) {
1104                         npgs = page_get_pagecnt(szc);
1105                         an_idx = P2ROUNDUP(an_idx + 1, npgs);
1106                 } else {
1107                         an_idx++;
1108                 }
1109         }
1110         /*
1111          * We can now drop the sptd->spt_lock since the ppa[]
1112          * exists and he have incremented pacachecnt.
1113          */
1114         mutex_exit(&sptd->spt_lock);

1116         /*
1117          * Since we cache the entire segment, we want to
1118          * set ppp to point to the first slot that corresponds
1119          * to the requested addr, i.e. pg_idx.
1120          */
1121         *ppp = &(sptd->spt_ppa[pg_idx]);
1122         return (0);

1124 insert_fail:
1125         /*
1126          * We will only reach this code if we tried and failed.
1127          *
1128          * And we can drop the lock on the dummy seg, once we've failed
1129          * to set up a new ppa[].
1130          */
1131         mutex_exit(&sptd->spt_lock);
```



```
1133         if (pl_built) {
1134                 if (claim_availrmem) {
1135                         mutex_enter(&freemem_lock);
1136                         availrmem += claim_availrmem;
1137                         mutex_exit(&freemem_lock);
1138                 }

1140                 /*
1141                  * We created pl and we need to destroy it.
1142                  */
1143                 pplist = pl;
1144                 for (an_idx = 0; an_idx < tot_npages; an_idx++) {
1145                         if (pplist[an_idx] != NULL)
1146                                 page_unlock(pplist[an_idx]);
1147                 }
1148                 kmem_free(pl, sizeof (page_t *) * tot_npages);
1149         }

1151         if (shmd->shm_softlockcnt <= 0) {
1152                 if (AS_ISUNMAPWAIT(seg->s_as)) {
1153                         mutex_enter(&seg->s_as->a_contents);
1154                         if (AS_ISUNMAPWAIT(seg->s_as)) {
1155                                 AS_CLRUNMAPWAIT(seg->s_as);
1156                                 cv_broadcast(&seg->s_as->a_cv);
1157                         }
1158                         mutex_exit(&seg->s_as->a_contents);
1159                 }
1160         }
1161         *ppp = NULL;
1162         return (ret);
1163 }



1167 /*
1168  * return locked pages over a given range.
1169  *
1170  * We will cache the entire ISM segment and save the pplist for the
1171  * entire segment in the ppa field of the underlying ISM segment structure.
1172  * Later, during a call to segspt_reclaim() we will use this ppa array
1173  * to page_unlock() all of the pages and then we will free this ppa list.
1174  */
1175 /*ARGSUSED*/
1176 static int
1177 segspt_shmpagelock(struct seg *seg, caddr_t addr, size_t len,
1178     struct page ***ppp, enum lock_type type, enum seg_rw rw)
1179 {
1180         struct shm_data *shmd = (struct shm_data *)seg->s_data;
1181         struct seg      *sptseg = shmd->shm_sptseg;
1182         struct spt_data *sptd = sptseg->s_data;
1183         pgcnt_t np, page_index, npages;
1184         caddr_t a, spt_base;
1185         struct page **pplist, **pl, *pp;
1186         struct anon_map *amp;
1187         ulong_t anon_index;
1188         int ret = ENOTSUP;
1189         uint_t  pl_built = 0;
1190         struct anon *ap;
1191         struct vnode *vp;
1192         u_offset_t off;

1194         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
1195         ASSERT(type == L_PAGELOCK || type == L_PAGEUNLOCK);
```

```
1198         /*
1199          * We want to lock/unlock the entire ISM segment. Therefore,
1200          * we will be using the underlying sptseg and it's base address
1201          * and length for the caching arguments.
1202          */
1203         ASSERT(sptseg);
1204         ASSERT(sptd);

1206         if (sptd->spt_flags & SHM_PAGEABLE) {
1207                 return (segspt_dismpagelock(seg, addr, len, ppp, type, rw));
1208         }

1210         page_index = seg_page(seg, addr);
1211         npages = btopr(len);

1213         /*
1214          * check if the request is larger than number of pages covered
1215          * by amp
1216          */
1217         if (page_index + npages > btopr(sptd->spt_amp->size)) {
1218                 *ppp = NULL;
1219                 return (ENOTSUP);
1220         }

1222         if (type == L_PAGEUNLOCK) {

1224                 ASSERT(sptd->spt_ppa != NULL);

1226                 seg_pinactive(seg, NULL, seg->s_base, sptd->spt_amp->size,
1227                     sptd->spt_ppa, S_WRITE, SEGP_FORCE_WIRED, segspt_reclaim);

1229                 /*
1230                  * If someone is blocked while unmapping, we purge
1231                  * segment page cache and thus reclaim pplist synchronously
1232                  * without waiting for seg_pasync_thread. This speeds up
1233                  * unmapping in cases where munmap(2) is called, while
1234                  * raw async i/o is still in progress or where a thread
1235                  * exits on data fault in a multithreaded application.
1236                  */
1237                 if (AS_ISUNMAPWAIT(seg->s_as) && (shmd->shm_softlockcnt > 0)) {
1238                         segspt_purge(seg);
1239                 }
1240                 return (0);
1241         }

1243         /* The L_PAGELOCK case... */

1245         /*
1246          * First try to find pages in segment page cache, without
1247          * holding the segment lock.
1248          */
1249         pplist = seg_plookup(seg, NULL, seg->s_base, sptd->spt_amp->size,
1250             S_WRITE, SEGP_FORCE_WIRED);
1251         if (pplist != NULL) {
1252                 ASSERT(sptd->spt_ppa == pplist);
1253                 ASSERT(sptd->spt_ppa[page_index]);
1254                 /*
1255                  * Since we cache the entire ISM segment, we want to
1256                  * set ppp to point to the first slot that corresponds
1257                  * to the requested addr, i.e. page_index.
1258                  */
1259                 *ppp = &(sptd->spt_ppa[page_index]);
1260                 return (0);
1261         }

1263         mutex_enter(&sptd->spt_lock);
```



```
1265         /*
1266          * try to find pages in segment page cache
1267          */
1268         pplist = seg_plookup(seg, NULL, seg->s_base, sptd->spt_amp->size,
1269             S_WRITE, SEGP_FORCE_WIRED);
1270         if (pplist != NULL) {
1271                 ASSERT(sptd->spt_ppa == pplist);
1272                 /*
1273                  * Since we cache the entire segment, we want to
1274                  * set ppp to point to the first slot that corresponds
1275                  * to the requested addr, i.e. page_index.
1276                  */
1277                 mutex_exit(&sptd->spt_lock);
1278                 *ppp = &(sptd->spt_ppa[page_index]);
1279                 return (0);
1280         }

1282         if (seg_pinsert_check(seg, NULL, seg->s_base, sptd->spt_amp->size,
1283             SEGP_FORCE_WIRED) == SEGP_FAIL) {
1284                 mutex_exit(&sptd->spt_lock);
1285                 *ppp = NULL;
1286                 return (ENOTSUP);
1287         }

1289         /*
1290          * No need to worry about protections because ISM pages
1291          * are always rw.
1292          */
1293         pl = pplist = NULL;

1295         /*
1296          * Do we need to build the ppa array?
1297          */
1298         if (sptd->spt_ppa == NULL) {
1299                 ASSERT(sptd->spt_ppa == pplist);

1301                 spt_base = sptseg->s_base;
1302                 pl_built = 1;

1304                 /*
1305                  * availrmem is decremented once during anon_swap_adjust()
1306                  * and is incremented during the anon_unresv(), which is
1307                  * called from shm_rm_amp() when the segment is destroyed.
1308                  */
1309                 amp = sptd->spt_amp;
1310                 ASSERT(amp != NULL);

1312                 /* pcachecnt is protected by sptd->spt_lock */
1313                 ASSERT(sptd->spt_pcachecnt == 0);
1314                 pplist = kmem_zalloc(sizeof (page_t *)
1315                     * btopr(sptd->spt_amp->size), KM_SLEEP);
1316                 pl = pplist;

1318                 anon_index = seg_page(sptseg, spt_base);

1320                 ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
1321                 for (a = spt_base; a < (spt_base + sptd->spt_amp->size);
1322                     a += PAGESIZE, anon_index++, pplist++) {
1323                         ap = anon_get_ptr(amp->ahp, anon_index);
1324                         ASSERT(ap != NULL);
1325                         swap_xlate(ap, &vp, &off);
1326                         pp = page_lookup(vp, off, SE_SHARED);
1327                         ASSERT(pp != NULL);
1328                         *pplist = pp;
1329                 }
```

```
1330                 ANON_LOCK_EXIT(&amp->a_rwlock);

1332                 if (a < (spt_base + sptd->spt_amp->size)) {
1333                         ret = ENOTSUP;
1334                         goto insert_fail;
1335                 }
1336                 sptd->spt_ppa = pl;
1337         } else {
1338                 /*
1339                  * We already have a valid ppa[].
1340                  */
1341                 pl = sptd->spt_ppa;
1342         }

1344         ASSERT(pl != NULL);

1346         ret = seg_pinsert(seg, NULL, seg->s_base, sptd->spt_amp->size,
1347             sptd->spt_amp->size, pl, S_WRITE, SEGP_FORCE_WIRED,
1348             segspt_reclaim);
1349         if (ret == SEGP_FAIL) {
1350                 /*
1351                  * seg_pinsert failed. We return
1352                  * ENOTSUP, so that the as_pagelock() code will
1353                  * then try the slower F_SOFTLOCK path.
1354                  */
1355                 if (pl_built) {
1356                         /*
1357                          * No one else has referenced the ppa[].
1358                          * We created it and we need to destroy it.
1359                          */
1360                         sptd->spt_ppa = NULL;
1361                 }
1362                 ret = ENOTSUP;
1363                 goto insert_fail;
1364         }

1366         /*
1367          * In either case, we increment softlockcnt on the 'real' segment.
1368          */
1369         sptd->spt_pcachecnt++;
1370         atomic_inc_ulong((ulong_t *)(&(shmd->shm_softlockcnt)));
1370         atomic_add_long((ulong_t *)(&(shmd->shm_softlockcnt)), 1);

1372         /*
1373          * We can now drop the sptd->spt_lock since the ppa[]
1374          * exists and he have incremented pacachecnt.
1375          */
1376         mutex_exit(&sptd->spt_lock);

1378         /*
1379          * Since we cache the entire segment, we want to
1380          * set ppp to point to the first slot that corresponds
1381          * to the requested addr, i.e. page_index.
1382          */
1383         *ppp = &(sptd->spt_ppa[page_index]);
1384         return (0);

1386 insert_fail:
1387         /*
1388          * We will only reach this code if we tried and failed.
1389          *
1390          * And we can drop the lock on the dummy seg, once we've failed
1391          * to set up a new ppa[].
1392          */
1393         mutex_exit(&sptd->spt_lock);
```



```
1395         if (pl_built) {
1396                 /*
1397                  * We created pl and we need to destroy it.
1398                  */
1399                 pplist = pl;
1400                 np = (((uintptr_t)(a - spt_base)) >> PAGESHIFT);
1401                 while (np) {
1402                         page_unlock(*pplist);
1403                         np--;
1404                         pplist++;
1405                 }
1406                 kmem_free(pl, sizeof (page_t *) * btopr(sptd->spt_amp->size));
1407         }
1408         if (shmd->shm_softlockcnt <= 0) {
1409                 if (AS_ISUNMAPWAIT(seg->s_as)) {
1410                         mutex_enter(&seg->s_as->a_contents);
1411                         if (AS_ISUNMAPWAIT(seg->s_as)) {
1412                                 AS_CLRUNMAPWAIT(seg->s_as);
1413                                 cv_broadcast(&seg->s_as->a_cv);
1414                         }
1415                         mutex_exit(&seg->s_as->a_contents);
1416                 }
1417         }
1418         *ppp = NULL;
1419         return (ret);
1420 }
```

_____***unchanged_portion_omitted_***

```
1431 static int
1432 segspt_reclaim(void *ptag, caddr_t addr, size_t len, struct page **pplist,
1433         enum seg_rw rw, int async)
1434 {
1435         struct seg *seg = (struct seg *)ptag;
1436         struct  shm_data *shmd = (struct shm_data *)seg->s_data;
1437         struct  seg     *sptseg;
1438         struct  spt_data *sptd;
1439         pgcnt_t npages, i, free_availrmem = 0;
1440         int     done = 0;

1442 #ifdef lint
1443         addr = addr;
1444 #endif
1445         sptseg = shmd->shm_sptseg;
1446         sptd = sptseg->s_data;
1447         npages = (len >> PAGESHIFT);
1448         ASSERT(npages);
1449         ASSERT(sptd->spt_pcachecnt != 0);
1450         ASSERT(sptd->spt_ppa == pplist);
1451         ASSERT(npages == btopr(sptd->spt_amp->size));
1452         ASSERT(async || AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

1454         /*
1455          * Acquire the lock on the dummy seg and destroy the
1456          * ppa array IF this is the last pcachecnt.
1457          */
1458         mutex_enter(&sptd->spt_lock);
1459         if (--sptd->spt_pcachecnt == 0) {
1460                 for (i = 0; i < npages; i++) {
1461                         if (pplist[i] == NULL) {
1462                                 continue;
1463                         }
1464                         if (rw == S_WRITE) {
1465                                 hat_setrefmod(pplist[i]);
1466                         } else {
1467                                 hat_setref(pplist[i]);
1468                         }
```

```
1469                        if ((sptd->spt_flags & SHM_PAGEABLE) &&
1470                            (sptd->spt_ppa_lckcnt[i] == 0))
1471                                free_availrmem++;
1472                        page_unlock(pplist[i]);
1473                }
1474                if ((sptd->spt_flags & SHM_PAGEABLE) && free_availrmem) {
1475                        mutex_enter(&freemem_lock);
1476                        availrmem += free_availrmem;
1477                        mutex_exit(&freemem_lock);
1478                }
1479                /*
1480                 * Since we want to cach/uncache the entire ISM segment,
1481                 * we will track the pplist in a segspt specific field
1482                 * ppa, that is initialized at the time we add an entry to
1483                 * the cache.
1484                 */
1485                ASSERT(sptd->spt_pcachecnt == 0);
1486                kmem_free(pplist, sizeof (page_t *) * npages);
1487                sptd->spt_ppa = NULL;
1488                sptd->spt_flags &= ~DISM_PPA_CHANGED;
1489                sptd->spt_gen++;
1490                cv_broadcast(&sptd->spt_cv);
1491                done = 1;
1492        }
1493        mutex_exit(&sptd->spt_lock);

1495        /*
1496         * If we are pcache async thread or called via seg_ppurge_wiredpp() we
1497         * may not hold AS lock (in this case async argument is not 0). This
1498         * means if softlockcnt drops to 0 after the decrement below address
1499         * space may get freed. We can't allow it since after softlock
1500         * derement to 0 we still need to access as structure for possible
1501         * wakeup of unmap waiters. To prevent the disappearance of as we take
1502         * this segment's shm_segfree_syncmtx. segspt_shmfree() also takes
1503         * this mutex as a barrier to make sure this routine completes before
1504         * segment is freed.
1505         *
1506         * The second complication we have to deal with in async case is a
1507         * possibility of missed wake up of unmap wait thread. When we don't
1508         * hold as lock here we may take a_contents lock before unmap wait
1509         * thread that was first to see softlockcnt was still not 0. As a
1510         * result we'll fail to wake up an unmap wait thread. To avoid this
1511         * race we set nounmapwait flag in as structure if we drop softlockcnt
1512         * to 0 if async is not 0.  unmapwait thread
1513         * will not block if this flag is set.
1514         */
1515        if (async)
1516                mutex_enter(&shmd->shm_segfree_syncmtx);

1518        /*
1519         * Now decrement softlockcnt.
1520         */
1521        ASSERT(shmd->shm_softlockcnt > 0);
1522        atomic_dec_ulong((ulong_t *)(&(shmd->shm_softlockcnt)));
1522        atomic_add_long((ulong_t *)(&(shmd->shm_softlockcnt)), -1);

1524        if (shmd->shm_softlockcnt <= 0) {
1525                if (async || AS_ISUNMAPWAIT(seg->s_as)) {
1526                        mutex_enter(&seg->s_as->a_contents);
1527                        if (async)
1528                                AS_SETNOUNMAPWAIT(seg->s_as);
1529                        if (AS_ISUNMAPWAIT(seg->s_as)) {
1530                                AS_CLRUNMAPWAIT(seg->s_as);
1531                                cv_broadcast(&seg->s_as->a_cv);
1532                        }
1533                        mutex_exit(&seg->s_as->a_contents);
```



```
1534                }
1535        }

1537        if (async)
1538                mutex_exit(&shmd->shm_segfree_syncmtx);

1540        return (done);
1541 }
```

*unchanged_portion_omitted_*

```
2859 /*
2860  * We need to wait for pending IO to complete to a DISM segment in order for
2861  * pages to get kicked out of the seg_pcache.  120 seconds should be more
2862  * than enough time to wait.
2863  */
2864 static clock_t spt_pcache_wait = 120;

2866 /*ARGSUSED*/
2867 static int
2868 segspt_shmadvise(struct seg *seg, caddr_t addr, size_t len, uint_t behav)
2869 {
2870        struct shm_data *shmd = (struct shm_data *)seg->s_data;
2871        struct spt_data *sptd = (struct spt_data *)shmd->shm_sptseg->s_data;
2872        struct anon_map *amp;
2873        pgcnt_t pg_idx;
2874        ushort_t gen;
2875        clock_t end_lbolt;
2876        int writer;
2877        page_t **ppa;

2879        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

2881        if (behav == MADV_FREE) {
2882                if ((sptd->spt_flags & SHM_PAGEABLE) == 0)
2883                        return (0);

2885                amp = sptd->spt_amp;
2886                pg_idx = seg_page(seg, addr);

2888                mutex_enter(&sptd->spt_lock);
2889                if ((ppa = sptd->spt_ppa) == NULL) {
2890                        mutex_exit(&sptd->spt_lock);
2891                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
2892                        anon_disclaim(amp, pg_idx, len);
2893                        ANON_LOCK_EXIT(&amp->a_rwlock);
2894                        return (0);
2895                }

2897                sptd->spt_flags |= DISM_PPA_CHANGED;
2898                gen = sptd->spt_gen;

2900                mutex_exit(&sptd->spt_lock);

2902                /*
2903                 * Purge all DISM cached pages
2904                 */
2905                seg_ppurge_wiredpp(ppa);

2907                /*
2908                 * Drop the AS_LOCK so that other threads can grab it
2909                 * in the as_pageunlock path and hopefully get the segment
2910                 * kicked out of the seg_pcache.  We bump the shm_softlockcnt
2911                 * to keep this segment resident.
2912                 */
2913                writer = AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock);
2914                atomic_inc_ulong((ulong_t *)(&(shmd->shm_softlockcnt)));
```

```
2914                atomic_add_long((ulong_t *)(&(shmd->shm_softlockcnt)), 1);
2915                AS_LOCK_EXIT(seg->s_as, &seg->s_as->a_lock);

2917                mutex_enter(&sptd->spt_lock);

2919                end_lbolt = ddi_get_lbolt() + (hz * spt_pcache_wait);

2921                /*
2922                 * Try to wait for pages to get kicked out of the seg_pcache.
2923                 */
2924                while (sptd->spt_gen == gen &&
2925                    (sptd->spt_flags & DISM_PPA_CHANGED) &&
2926                    ddi_get_lbolt() < end_lbolt) {
2927                        if (!cv_timedwait_sig(&sptd->spt_cv,
2928                            &sptd->spt_lock, end_lbolt)) {
2929                                break;
2930                        }
2931                }

2933                mutex_exit(&sptd->spt_lock);

2935                /* Regrab the AS_LOCK and release our hold on the segment */
2936                AS_LOCK_ENTER(seg->s_as, &seg->s_as->a_lock,
2937                    writer ? RW_WRITER : RW_READER);
2938                atomic_dec_ulong((ulong_t *)(&(shmd->shm_softlockcnt)));
2938                atomic_add_long((ulong_t *)(&(shmd->shm_softlockcnt)), -1);
2939                if (shmd->shm_softlockcnt <= 0) {
2940                        if (AS_ISUNMAPWAIT(seg->s_as)) {
2941                                mutex_enter(&seg->s_as->a_contents);
2942                                if (AS_ISUNMAPWAIT(seg->s_as)) {
2943                                        AS_CLRUNMAPWAIT(seg->s_as);
2944                                        cv_broadcast(&seg->s_as->a_cv);
2945                                }
2946                                mutex_exit(&seg->s_as->a_contents);
2947                        }
2948                }

2950                ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
2951                anon_disclaim(amp, pg_idx, len);
2952                ANON_LOCK_EXIT(&amp->a_rwlock);
2953        } else if (lgrp_optimizations() && (behav == MADV_ACCESS_LWP ||
2954            behav == MADV_ACCESS_MANY || behav == MADV_ACCESS_DEFAULT)) {
2955                int                     already_set;
2956                ulong_t                 anon_index;
2957                lgrp_mem_policy_t       policy;
2958                caddr_t                 shm_addr;
2959                size_t                  share_size;
2960                size_t                  size;
2961                struct seg              *sptseg = shmd->shm_sptseg;
2962                caddr_t                 sptseg_addr;

2964                /*
2965                 * Align address and length to page size of underlying segment
2966                 */
2967                share_size = page_get_pagesize(shmd->shm_sptseg->s_szc);
2968                shm_addr = (caddr_t)P2ALIGN((uintptr_t)(addr), share_size);
2969                size = P2ROUNDUP((uintptr_t)(((addr + len) - shm_addr)),
2970                    share_size);

2972                amp = shmd->shm_amp;
2973                anon_index = seg_page(seg, shm_addr);

2975                /*
2976                 * And now we may have to adjust size downward if we have
2977                 * exceeded the realsize of the segment or initial anon
2978                 * allocations.
```



```
2979                 */
2980                sptseg_addr = sptseg->s_base + ptob(anon_index);
2981                if ((sptseg_addr + size) >
2982                    (sptseg->s_base + sptd->spt_realsize))
2983                        size = (sptseg->s_base + sptd->spt_realsize) -
2984                            sptseg_addr;

2986                /*
2987                 * Set memory allocation policy for this segment
2988                 */
2989                policy = lgrp_madv_to_policy(behav, len, MAP_SHARED);
2990                already_set = lgrp_shm_policy_set(policy, amp, anon_index,
2991                    NULL, 0, len);

2993                /*
2994                 * If random memory allocation policy set already,
2995                 * don't bother reapplying it.
2996                 */
2997                if (already_set && !LGRP_MEM_POLICY_REAPPLICABLE(policy))
2998                        return (0);

3000                /*
3001                 * Mark any existing pages in the given range for
3002                 * migration, flushing the I/O page cache, and using
3003                 * underlying segment to calculate anon index and get
3004                 * anonmap and vnode pointer from
3005                 */
3006                if (shmd->shm_softlockcnt > 0)
3007                        segspt_purge(seg);

3009                page_mark_migrate(seg, shm_addr, size, amp, 0, NULL, 0, 0);
3010        }

3012        return (0);
3013 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  280582 Mon Jul 28 07:44:56 2014
new/usr/src/uts/common/vm/seg_vn.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

2604 static int stealcow = 1;

2606 /*
2607  * Workaround for viking chip bug.  See bug id 1220902.
2608  * To fix this down in pagefault() would require importing so
2609  * much as and segvn code as to be unmaintainable.
2610  */
2611 int enable_mbit_wa = 0;

2613 /*
2614  * Handles all the dirty work of getting the right
2615  * anonymous pages and loading up the translations.
2616  * This routine is called only from segvn_fault()
2617  * when looping over the range of addresses requested.
2618  *
2619  * The basic algorithm here is:
2620  *      If this is an anon_zero case
2621  *              Call anon_zero to allocate page
2622  *              Load up translation
2623  *              Return
2624  *      endif
2625  *      If this is an anon page
2626  *              Use anon_getpage to get the page
2627  *      else
2628  *              Find page in pl[] list passed in
2629  *      endif
2630  *      If not a cow
2631  *              Load up the translation to the page
2632  *              return
2633  *      endif
2634  *      Call anon_private to handle cow
2635  *      Load up (writable) translation to new page
2636  */
2637 static faultcode_t
2638 segvn_faultpage(
2639         struct hat *hat,                /* the hat to use for mapping */
2640         struct seg *seg,                /* seg_vn of interest */
2641         caddr_t addr,                   /* address in as */
2642         u_offset_t off,                 /* offset in vp */
2643         struct vpage *vpage,            /* pointer to vpage for vp, off */
2644         page_t *pl[],                   /* object source page pointer */
2645         uint_t vpprot,                  /* access allowed to object pages */
2646         enum fault_type type,           /* type of fault */
2647         enum seg_rw rw,                 /* type of access at fault */
2648         int brkcow)                     /* we may need to break cow */
2649 {
2650         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
2651         page_t *pp, **ppp;
2652         uint_t pageflags = 0;
2653         page_t *anon_pl[1 + 1];
2654         page_t *opp = NULL;             /* original page */
2655         uint_t prot;
2656         int err;
2657         int cow;
2658         int claim;
2659         int steal = 0;
2660         ulong_t anon_index;
2661         struct anon *ap, *oldap;
2662         struct anon_map *amp;
```



```
2663         int hat_flag = (type == F_SOFTLOCK) ? HAT_LOAD_LOCK : HAT_LOAD;
2664         int anon_lock = 0;
2665         anon_sync_obj_t cookie;

2667         if (svd->flags & MAP_TEXT) {
2668                 hat_flag |= HAT_LOAD_TEXT;
2669         }

2671         ASSERT(SEGVN_READ_HELD(seg->s_as, &svd->lock));
2672         ASSERT(seg->s_szc == 0);
2673         ASSERT(svd->tr_state != SEGVN_TR_INIT);

2675         /*
2676          * Initialize protection value for this page.
2677          * If we have per page protection values check it now.
2678          */
2679         if (svd->pageprot) {
2680                 uint_t protchk;

2682                 switch (rw) {
2683                 case S_READ:
2684                         protchk = PROT_READ;
2685                         break;
2686                 case S_WRITE:
2687                         protchk = PROT_WRITE;
2688                         break;
2689                 case S_EXEC:
2690                         protchk = PROT_EXEC;
2691                         break;
2692                 case S_OTHER:
2693                 default:
2694                         protchk = PROT_READ | PROT_WRITE | PROT_EXEC;
2695                         break;
2696                 }

2698                 prot = VPP_PROT(vpage);
2699                 if ((prot & protchk) == 0)
2700                         return (FC_PROT);       /* illegal access type */
2701         } else {
2702                 prot = svd->prot;
2703         }

2705         if (type == F_SOFTLOCK) {
2706                 atomic_inc_ulong((ulong_t *)&svd->softlockcnt);
2706                 atomic_add_long((ulong_t *)&svd->softlockcnt, 1);
2707         }

2709         /*
2710          * Always acquire the anon array lock to prevent 2 threads from
2711          * allocating separate anon slots for the same "addr".
2712          */

2714         if ((amp = svd->amp) != NULL) {
2715                 ASSERT(RW_READ_HELD(&amp->a_rwlock));
2716                 anon_index = svd->anon_index + seg_page(seg, addr);
2717                 anon_array_enter(amp, anon_index, &cookie);
2718                 anon_lock = 1;
2719         }

2721         if (svd->vp == NULL && amp != NULL) {
2722                 if ((ap = anon_get_ptr(amp->ahp, anon_index)) == NULL) {
2723                         /*
2724                          * Allocate a (normally) writable anonymous page of
2725                          * zeroes. If no advance reservations, reserve now.
2726                          */
2727                         if (svd->flags & MAP_NORESERVE) {
```

```
2728                                    if (anon_resv_zone(ptob(1),
2729                                        seg->s_as->a_proc->p_zone)) {
2730                                            atomic_add_long(&svd->swresv, ptob(1));
2731                                            atomic_add_long(&seg->s_as->a_resvsize,
2732                                                ptob(1));
2733                                    } else {
2734                                            err = ENOMEM;
2735                                            goto out;
2736                                    }
2737                            }
2738                            if ((pp = anon_zero(seg, addr, &ap,
2739                                svd->cred)) == NULL) {
2740                                err = ENOMEM;
2741                                goto out;       /* out of swap space */
2742                            }
2743                            /*
2744                             * Re-acquire the anon_map lock and
2745                             * initialize the anon array entry.
2746                             */
2747                            (void) anon_set_ptr(amp->ahp, anon_index, ap,
2748                                ANON_SLEEP);

2750                            ASSERT(pp->p_szc == 0);

2752                            /*
2753                             * Handle pages that have been marked for migration
2754                             */
2755                            if (lgrp_optimizations())
2756                                    page_migrate(seg, addr, &pp, 1);

2758                            if (enable_mbit_wa) {
2759                                    if (rw == S_WRITE)
2760                                            hat_setmod(pp);
2761                                    else if (!hat_ismod(pp))
2762                                            prot &= ~PROT_WRITE;
2763                            }
2764                            /*
2765                             * If AS_PAGLCK is set in a_flags (via memcntl(2)
2766                             * with MC_LOCKAS, MCL_FUTURE) and this is a
2767                             * MAP_NORESERVE segment, we may need to
2768                             * permanently lock the page as it is being faulted
2769                             * for the first time. The following text applies
2770                             * only to MAP_NORESERVE segments:
2771                             *
2772                             * As per memcntl(2), if this segment was created
2773                             * after MCL_FUTURE was applied (a "future"
2774                             * segment), its pages must be locked.  If this
2775                             * segment existed at MCL_FUTURE application (a
2776                             * "past" segment), the interface is unclear.
2777                             *
2778                             * We decide to lock only if vpage is present:
2779                             *
2780                             * - "future" segments will have a vpage array (see
2781                             *    as_map), and so will be locked as required
2782                             *
2783                             * - "past" segments may not have a vpage array,
2784                             *    depending on whether events (such as
2785                             *    mprotect) have occurred. Locking if vpage
2786                             *    exists will preserve legacy behavior.  Not
2787                             *    locking if vpage is absent, will not break
2788                             *    the interface or legacy behavior.  Note that
2789                             *    allocating vpage here if it's absent requires
2790                             *    upgrading the segvn reader lock, the cost of
2791                             *    which does not seem worthwhile.
2792                             *
2793                             * Usually testing and setting VPP_ISPPLOCK and
```



```
2794                             * VPP_SETPPLOCK requires holding the segvn lock as
2795                             * writer, but in this case all readers are
2796                             * serializing on the anon array lock.
2797                             */
2798                            if (AS_ISPGLCK(seg->s_as) && vpage != NULL &&
2799                                (svd->flags & MAP_NORESERVE) &&
2800                                !VPP_ISPPLOCK(vpage)) {
2801                                    proc_t *p = seg->s_as->a_proc;
2802                                    ASSERT(svd->type == MAP_PRIVATE);
2803                                    mutex_enter(&p->p_lock);
2804                                    if (rctl_incr_locked_mem(p, NULL, PAGESIZE,
2805                                        1) == 0) {
2806                                            claim = VPP_PROT(vpage) & PROT_WRITE;
2807                                            if (page_pp_lock(pp, claim, 0)) {
2808                                                    VPP_SETPPLOCK(vpage);
2809                                            } else {
2810                                                    rctl_decr_locked_mem(p, NULL,
2811                                                        PAGESIZE, 1);
2812                                            }
2813                                    }
2814                                    mutex_exit(&p->p_lock);
2815                            }

2817                            ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
2818                            hat_memload(hat, addr, pp, prot, hat_flag);

2820                            if (!(hat_flag & HAT_LOAD_LOCK))
2821                                    page_unlock(pp);

2823                            anon_array_exit(&cookie);
2824                            return (0);
2825                    }
2826            }

2828            /*
2829             * Obtain the page structure via anon_getpage() if it is
2830             * a private copy of an object (the result of a previous
2831             * copy-on-write).
2832             */
2833            if (amp != NULL) {
2834                    if ((ap = anon_get_ptr(amp->ahp, anon_index)) != NULL) {
2835                            err = anon_getpage(&ap, &vpprot, anon_pl, PAGESIZE,
2836                                seg, addr, rw, svd->cred);
2837                            if (err)
2838                                    goto out;

2840                            if (svd->type == MAP_SHARED) {
2841                                    /*
2842                                     * If this is a shared mapping to an
2843                                     * anon_map, then ignore the write
2844                                     * permissions returned by anon_getpage().
2845                                     * They apply to the private mappings
2846                                     * of this anon_map.
2847                                     */
2848                                    vpprot |= PROT_WRITE;
2849                            }
2850                            opp = anon_pl[0];
2851                    }
2852            }

2854            /*
2855             * Search the pl[] list passed in if it is from the
2856             * original object (i.e., not a private copy).
2857             */
2858            if (opp == NULL) {
2859                    /*
```

```
2860                  * Find original page.  We must be bringing it in
2861                  * from the list in pl[].
2862                  */
2863                 for (ppp = pl; (opp = *ppp) != NULL; ppp++) {
2864                         if (opp == PAGE_HANDLED)
2865                                 continue;
2866                         ASSERT(opp->p_vnode == svd->vp); /* XXX */
2867                         if (opp->p_offset == off)
2868                                 break;
2869                 }
2870                 if (opp == NULL) {
2871                         panic("segvn_faultpage not found");
2872                         /*NOTREACHED*/
2873                 }
2874                 *ppp = PAGE_HANDLED;

2876         }

2878         ASSERT(PAGE_LOCKED(opp));

2880         TRACE_3(TR_FAC_VM, TR_SEGVN_FAULT,
2881             "segvn_fault:pp %p vp %p offset %llx", opp, NULL, 0);

2883         /*
2884          * The fault is treated as a copy-on-write fault if a
2885          * write occurs on a private segment and the object
2886          * page (i.e., mapping) is write protected.  We assume
2887          * that fatal protection checks have already been made.
2888          */

2890         if (brkcow) {
2891                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
2892                 cow = !(vpprot & PROT_WRITE);
2893         } else if (svd->tr_state == SEGVN_TR_ON) {
2894                 /*
2895                  * If we are doing text replication COW on first touch.
2896                  */
2897                 ASSERT(amp != NULL);
2898                 ASSERT(svd->vp != NULL);
2899                 ASSERT(rw != S_WRITE);
2900                 cow = (ap == NULL);
2901         } else {
2902                 cow = 0;
2903         }

2905         /*
2906          * If not a copy-on-write case load the translation
2907          * and return.
2908          */
2909         if (cow == 0) {

2911                 /*
2912                  * Handle pages that have been marked for migration
2913                  */
2914                 if (lgrp_optimizations())
2915                         page_migrate(seg, addr, &opp, 1);

2917                 if (IS_VMODSORT(opp->p_vnode) || enable_mbit_wa) {
2918                         if (rw == S_WRITE)
2919                                 hat_setmod(opp);
2920                         else if (rw != S_OTHER && !hat_ismod(opp))
2921                                 prot &= ~PROT_WRITE;
2922                 }

2924                 ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE ||
2925                     (!svd->pageprot && svd->prot == (prot & vpprot)));
```



```
2926                 ASSERT(amp == NULL ||
2927                     svd->rcookie == HAT_INVALID_REGION_COOKIE);
2928                 hat_memload_region(hat, addr, opp, prot & vpprot, hat_flag,
2929                     svd->rcookie);

2931                 if (!(hat_flag & HAT_LOAD_LOCK))
2932                         page_unlock(opp);

2934                 if (anon_lock) {
2935                         anon_array_exit(&cookie);
2936                 }
2937                 return (0);
2938         }

2940         ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);

2942         hat_setref(opp);

2944         ASSERT(amp != NULL && anon_lock);

2946         /*
2947          * Steal the page only if it isn't a private page
2948          * since stealing a private page is not worth the effort.
2949          */
2950         if ((ap = anon_get_ptr(amp->ahp, anon_index)) == NULL)
2951                 steal = 1;

2953         /*
2954          * Steal the original page if the following conditions are true:
2955          *
2956          * We are low on memory, the page is not private, page is not large,
2957          * not shared, not modified, not 'locked' or if we have it 'locked'
2958          * (i.e., p_cowcnt == 1 and p_lckcnt == 0, which also implies
2959          * that the page is not shared) and if it doesn't have any
2960          * translations. page_struct_lock isn't needed to look at p_cowcnt
2961          * and p_lckcnt because we first get exclusive lock on page.
2962          */
2963         (void) hat_pagesync(opp, HAT_SYNC_DONTZERO | HAT_SYNC_STOPON_MOD);

2965         if (stealcow && freemem < minfree && steal && opp->p_szc == 0 &&
2966             page_tryupgrade(opp) && !hat_ismod(opp) &&
2967             ((opp->p_lckcnt == 0 && opp->p_cowcnt == 0) ||
2968             (opp->p_lckcnt == 0 && opp->p_cowcnt == 1 &&
2969             vpage != NULL && VPP_ISPPLOCK(vpage)))) {
2970                 /*
2971                  * Check if this page has other translations
2972                  * after unloading our translation.
2973                  */
2974                 if (hat_page_is_mapped(opp)) {
2975                         ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
2976                         hat_unload(seg->s_as->a_hat, addr, PAGESIZE,
2977                             HAT_UNLOAD);
2978                 }

2980                 /*
2981                  * hat_unload() might sync back someone else's recent
2982                  * modification, so check again.
2983                  */
2984                 if (!hat_ismod(opp) && !hat_page_is_mapped(opp))
2985                         pageflags |= STEAL_PAGE;
2986         }

2988         /*
2989          * If we have a vpage pointer, see if it indicates that we have
2990          * ''locked'' the page we map -- if so, tell anon_private to
2991          * transfer the locking resource to the new page.
```

```
2992          *
2993          * See Statement at the beginning of segvn_lockop regarding
2994          * the way lockcnts/cowcnts are handled during COW.
2995          *
2996          */
2997         if (vpage != NULL && VPP_ISPPLOCK(vpage))
2998                 pageflags |= LOCK_PAGE;

3000         /*
3001          * Allocate a private page and perform the copy.
3002          * For MAP_NORESERVE reserve swap space now, unless this
3003          * is a cow fault on an existing anon page in which case
3004          * MAP_NORESERVE will have made advance reservations.
3005          */
3006         if ((svd->flags & MAP_NORESERVE) && (ap == NULL)) {
3007                 if (anon_resv_zone(ptob(1), seg->s_as->a_proc->p_zone)) {
3008                         atomic_add_long(&svd->swresv, ptob(1));
3009                         atomic_add_long(&seg->s_as->a_resvsize, ptob(1));
3010                 } else {
3011                         page_unlock(opp);
3012                         err = ENOMEM;
3013                         goto out;
3014                 }
3015         }
3016         oldap = ap;
3017         pp = anon_private(&ap, seg, addr, prot, opp, pageflags, svd->cred);
3018         if (pp == NULL) {
3019                 err = ENOMEM;   /* out of swap space */
3020                 goto out;
3021         }

3023         /*
3024          * If we copied away from an anonymous page, then
3025          * we are one step closer to freeing up an anon slot.
3026          *
3027          * NOTE:  The original anon slot must be released while
3028          * holding the "anon_map" lock.  This is necessary to prevent
3029          * other threads from obtaining a pointer to the anon slot
3030          * which may be freed if its "refcnt" is 1.
3031          */
3032         if (oldap != NULL)
3033                 anon_decref(oldap);

3035         (void) anon_set_ptr(amp->ahp, anon_index, ap, ANON_SLEEP);

3037         /*
3038          * Handle pages that have been marked for migration
3039          */
3040         if (lgrp_optimizations())
3041                 page_migrate(seg, addr, &pp, 1);

3043         ASSERT(pp->p_szc == 0);

3045         ASSERT(!IS_VMODSORT(pp->p_vnode));
3046         if (enable_mbit_wa) {
3047                 if (rw == S_WRITE)
3048                         hat_setmod(pp);
3049                 else if (!hat_ismod(pp))
3050                         prot &= ~PROT_WRITE;
3051         }

3053         ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
3054         hat_memload(hat, addr, pp, prot, hat_flag);

3056         if (!(hat_flag & HAT_LOAD_LOCK))
3057                 page_unlock(pp);
```



```
3059         ASSERT(anon_lock);
3060         anon_array_exit(&cookie);
3061         return (0);
3062 out:
3063         if (anon_lock)
3064                 anon_array_exit(&cookie);

3066         if (type == F_SOFTLOCK) {
3067                 atomic_dec_ulong((ulong_t *)&svd->softlockcnt);
3067                 atomic_add_long((ulong_t *)&svd->softlockcnt, -1);
3068         }
3069         return (FC_MAKE_ERR(err));
3070 }
______unchanged_portion_omitted_

8555 #ifdef DEBUG
8556 static uint32_t segvn_pglock_mtbf = 0;
8557 #endif

8559 #define PCACHE_SHWLIST          ((page_t *)-2)
8560 #define NOPCACHE_SHWLIST        ((page_t *)-1)

8562 /*
8563  * Lock/Unlock anon pages over a given range. Return shadow list. This routine
8564  * uses global segment pcache to cache shadow lists (i.e. pp arrays) of pages
8565  * to avoid the overhead of per page locking, unlocking for subsequent IOs to
8566  * the same parts of the segment. Currently shadow list creation is only
8567  * supported for pure anon segments. MAP_PRIVATE segment pcache entries are
8568  * tagged with segment pointer, starting virtual address and length. This
8569  * approach for MAP_SHARED segments may add many pcache entries for the same
8570  * set of pages and lead to long hash chains that decrease pcache lookup
8571  * performance. To avoid this issue for shared segments shared anon map and
8572  * starting anon index are used for pcache entry tagging. This allows all
8573  * segments to share pcache entries for the same anon range and reduces pcache
8574  * chain's length as well as memory overhead from duplicate shadow lists and
8575  * pcache entries.
8576  *
8577  * softlockcnt field in segvn_data structure counts the number of F_SOFTLOCK'd
8578  * pages via segvn_fault() and pagelock'd pages via this routine. But pagelock
8579  * part of softlockcnt accounting is done differently for private and shared
8580  * segments. In private segment case softlock is only incremented when a new
8581  * shadow list is created but not when an existing one is found via
8582  * seg_plookup(). pcache entries have reference count incremented/decremented
8583  * by each seg_plookup()/seg_pinactive() operation. Only entries that have 0
8584  * reference count can be purged (and purging is needed before segment can be
8585  * freed). When a private segment pcache entry is purged segvn_reclaim() will
8586  * decrement softlockcnt. Since in private segment case each of its pcache
8587  * entries only belongs to this segment we can expect that when
8588  * segvn_pagelock(L_PAGEUNLOCK) was called for all outstanding IOs in this
8589  * segment purge will succeed and softlockcnt will drop to 0. In shared
8590  * segment case reference count in pcache entry counts active locks from many
8591  * different segments so we can't expect segment purging to succeed even when
8592  * segvn_pagelock(L_PAGEUNLOCK) was called for all outstanding IOs in this
8593  * segment. To be able to determine when there're no pending pagelocks in
8594  * shared segment case we don't rely on purging to make softlockcnt drop to 0
8595  * but instead softlockcnt is incremented and decremented for every
8596  * segvn_pagelock(L_PAGELOCK/L_PAGEUNLOCK) call regardless if a new shadow
8597  * list was created or an existing one was found. When softlockcnt drops to 0
8598  * this segment no longer has any claims for pcached shadow lists and the
8599  * segment can be freed even if there're still active pcache entries
8600  * shared by this segment anon map. Shared segment pcache entries belong to
8601  * anon map and are typically removed when anon map is freed after all
8602  * processes destroy the segments that use this anon map.
8603  */
8604 static int
```

```
8605 segvn_pagelock(struct seg *seg, caddr_t addr, size_t len, struct page ***ppp,
8606     enum lock_type type, enum seg_rw rw)
8607 {
8608         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
8609         size_t np;
8610         pgcnt_t adjustpages;
8611         pgcnt_t npages;
8612         ulong_t anon_index;
8613         uint_t protchk = (rw == S_READ) ? PROT_READ : PROT_WRITE;
8614         uint_t error;
8615         struct anon_map *amp;
8616         pgcnt_t anpgcnt;
8617         struct page **pplist, **pl, *pp;
8618         caddr_t a;
8619         size_t page;
8620         caddr_t lpgaddr, lpgeaddr;
8621         anon_sync_obj_t cookie;
8622         int anlock;
8623         struct anon_map *pamp;
8624         caddr_t paddr;
8625         seg_preclaim_cbfunc_t preclaim_callback;
8626         size_t pgsz;
8627         int use_pcache;
8628         size_t wlen;
8629         uint_t pflags = 0;
8630         int sftlck_sbase = 0;
8631         int sftlck_send = 0;

8633 #ifdef DEBUG
8634         if (type == L_PAGELOCK && segvn_pglock_mtbf) {
8635                 hrtime_t ts = gethrtime();
8636                 if ((ts % segvn_pglock_mtbf) == 0) {
8637                         return (ENOTSUP);
8638                 }
8639                 if ((ts % segvn_pglock_mtbf) == 1) {
8640                         return (EFAULT);
8641                 }
8642         }
8643 #endif

8645         TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_START,
8646             "segvn_pagelock: start seg %p addr %p", seg, addr);

8648         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
8649         ASSERT(type == L_PAGELOCK || type == L_PAGEUNLOCK);

8651         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);

8653         /*
8654          * for now we only support pagelock to anon memory. We would have to
8655          * check protections for vnode objects and call into the vnode driver.
8656          * That's too much for a fast path. Let the fault entry point handle
8657          * it.
8658          */
8659         if (svd->vp != NULL) {
8660                 if (type == L_PAGELOCK) {
8661                         error = ENOTSUP;
8662                         goto out;
8663                 }
8664                 panic("segvn_pagelock(L_PAGEUNLOCK): vp != NULL");
8665         }
8666         if ((amp = svd->amp) == NULL) {
8667                 if (type == L_PAGELOCK) {
8668                         error = EFAULT;
8669                         goto out;
8670                 }
```



```
8671                 panic("segvn_pagelock(L_PAGEUNLOCK): amp == NULL");
8672         }
8673         if (rw != S_READ && rw != S_WRITE) {
8674                 if (type == L_PAGELOCK) {
8675                         error = ENOTSUP;
8676                         goto out;
8677                 }
8678                 panic("segvn_pagelock(L_PAGEUNLOCK): bad rw");
8679         }

8681         if (seg->s_szc != 0) {
8682                 /*
8683                  * We are adjusting the pagelock region to the large page size
8684                  * boundary because the unlocked part of a large page cannot
8685                  * be freed anyway unless all constituent pages of a large
8686                  * page are locked. Bigger regions reduce pcache chain length
8687                  * and improve lookup performance. The tradeoff is that the
8688                  * very first segvn_pagelock() call for a given page is more
8689                  * expensive if only 1 page_t is needed for IO. This is only
8690                  * an issue if pcache entry doesn't get reused by several
8691                  * subsequent calls. We optimize here for the case when pcache
8692                  * is heavily used by repeated IOs to the same address range.
8693                  *
8694                  * Note segment's page size cannot change while we are holding
8695                  * as lock.  And then it cannot change while softlockcnt is
8696                  * not 0. This will allow us to correctly recalculate large
8697                  * page size region for the matching pageunlock/reclaim call
8698                  * since as_pageunlock() caller must always match
8699                  * as_pagelock() call's addr and len.
8700                  *
8701                  * For pageunlock *ppp points to the pointer of page_t that
8702                  * corresponds to the real unadjusted start address. Similar
8703                  * for pagelock *ppp must point to the pointer of page_t that
8704                  * corresponds to the real unadjusted start address.
8705                  */
8706                 pgsz = page_get_pagesize(seg->s_szc);
8707                 CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
8708                 adjustpages = btop((uintptr_t)(addr - lpgaddr));
8709         } else if (len < segvn_pglock_comb_thrshld) {
8710                 lpgaddr = addr;
8711                 lpgeaddr = addr + len;
8712                 adjustpages = 0;
8713                 pgsz = PAGESIZE;
8714         } else {
8715                 /*
8716                  * Align the address range of large enough requests to allow
8717                  * combining of different shadow lists into 1 to reduce memory
8718                  * overhead from potentially overlapping large shadow lists
8719                  * (worst case is we have a 1MB IO into buffers with start
8720                  * addresses separated by 4K).  Alignment is only possible if
8721                  * padded chunks have sufficient access permissions. Note
8722                  * permissions won't change between L_PAGELOCK and
8723                  * L_PAGEUNLOCK calls since non 0 softlockcnt will force
8724                  * segvn_setprot() to wait until softlockcnt drops to 0. This
8725                  * allows us to determine in L_PAGEUNLOCK the same range we
8726                  * computed in L_PAGELOCK.
8727                  *
8728                  * If alignment is limited by segment ends set
8729                  * sftlck_sbase/sftlck_send flags. In L_PAGELOCK case when
8730                  * these flags are set bump softlockcnt_sbase/softlockcnt_send
8731                  * per segment counters. In L_PAGEUNLOCK case decrease
8732                  * softlockcnt_sbase/softlockcnt_send counters if
8733                  * sftlck_sbase/sftlck_send flags are set.  When
8734                  * softlockcnt_sbase/softlockcnt_send are non 0
8735                  * segvn_concat()/segvn_extend_prev()/segvn_extend_next()
8736                  * won't merge the segments. This restriction combined with
```

```
8737                 * restriction on segment unmapping and splitting for segments
8738                 * that have non 0 softlockcnt allows L_PAGEUNLOCK to
8739                 * correctly determine the same range that was previously
8740                 * locked by matching L_PAGELOCK.
8741                 */
8742                pflags = SEGP_PSHIFT | (segvn_pglock_comb_bshift << 16);
8743                pgsz = PAGESIZE;
8744                if (svd->type == MAP_PRIVATE) {
8745                        lpgaddr = (caddr_t)P2ALIGN((uintptr_t)addr,
8746                            segvn_pglock_comb_balign);
8747                        if (lpgaddr < seg->s_base) {
8748                                lpgaddr = seg->s_base;
8749                                sftlck_sbase = 1;
8750                        }
8751                } else {
8752                        ulong_t aix = svd->anon_index + seg_page(seg, addr);
8753                        ulong_t aaix = P2ALIGN(aix, segvn_pglock_comb_palign);
8754                        if (aaix < svd->anon_index) {
8755                                lpgaddr = seg->s_base;
8756                                sftlck_sbase = 1;
8757                        } else {
8758                                lpgaddr = addr - ptob(aix - aaix);
8759                                ASSERT(lpgaddr >= seg->s_base);
8760                        }
8761                }
8762                if (svd->pageprot && lpgaddr != addr) {
8763                        struct vpage *vp = &svd->vpage[seg_page(seg, lpgaddr)];
8764                        struct vpage *evp = &svd->vpage[seg_page(seg, addr)];
8765                        while (vp < evp) {
8766                                if ((VPP_PROT(vp) & protchk) == 0) {
8767                                        break;
8768                                }
8769                                vp++;
8770                        }
8771                        if (vp < evp) {
8772                                lpgaddr = addr;
8773                                pflags = 0;
8774                        }
8775                }
8776                lpgeaddr = addr + len;
8777                if (pflags) {
8778                        if (svd->type == MAP_PRIVATE) {
8779                                lpgeaddr = (caddr_t)P2ROUNDUP(
8780                                    (uintptr_t)lpgeaddr,
8781                                    segvn_pglock_comb_balign);
8782                        } else {
8783                                ulong_t aix = svd->anon_index +
8784                                    seg_page(seg, lpgeaddr);
8785                                ulong_t aaix = P2ROUNDUP(aix,
8786                                    segvn_pglock_comb_palign);
8787                                if (aaix < aix) {
8788                                        lpgeaddr = 0;
8789                                } else {
8790                                        lpgeaddr += ptob(aaix - aix);
8791                                }
8792                        }
8793                        if (lpgeaddr == 0 ||
8794                            lpgeaddr > seg->s_base + seg->s_size) {
8795                                lpgeaddr = seg->s_base + seg->s_size;
8796                                sftlck_send = 1;
8797                        }
8798                }
8799                if (svd->pageprot && lpgeaddr != addr + len) {
8800                        struct vpage *vp;
8801                        struct vpage *evp;
```



```
8803                        vp = &svd->vpage[seg_page(seg, addr + len)];
8804                        evp = &svd->vpage[seg_page(seg, lpgeaddr)];
8805
8806                        while (vp < evp) {
8807                                if ((VPP_PROT(vp) & protchk) == 0) {
8808                                        break;
8809                                }
8810                                vp++;
8811                        }
8812                        if (vp < evp) {
8813                                lpgeaddr = addr + len;
8814                        }
8815                }
8816                adjustpages = btop((uintptr_t)(addr - lpgaddr));
8817        }
8818
8819        /*
8820         * For MAP_SHARED segments we create pcache entries tagged by amp and
8821         * anon index so that we can share pcache entries with other segments
8822         * that map this amp.  For private segments pcache entries are tagged
8823         * with segment and virtual address.
8824         */
8825        if (svd->type == MAP_SHARED) {
8826                pamp = amp;
8827                paddr = (caddr_t)((lpgaddr - seg->s_base) +
8828                    ptob(svd->anon_index));
8829                preclaim_callback = shamp_reclaim;
8830        } else {
8831                pamp = NULL;
8832                paddr = lpgaddr;
8833                preclaim_callback = segvn_reclaim;
8834        }
8835
8836        if (type == L_PAGEUNLOCK) {
8837                VM_STAT_ADD(segvnvmstats.pagelock[0]);
8838
8839                /*
8840                 * update hat ref bits for /proc. We need to make sure
8841                 * that threads tracing the ref and mod bits of the
8842                 * address space get the right data.
8843                 * Note: page ref and mod bits are updated at reclaim time
8844                 */
8845                if (seg->s_as->a_vbits) {
8846                        for (a = addr; a < addr + len; a += PAGESIZE) {
8847                                if (rw == S_WRITE) {
8848                                        hat_setstat(seg->s_as, a,
8849                                            PAGESIZE, P_REF | P_MOD);
8850                                } else {
8851                                        hat_setstat(seg->s_as, a,
8852                                            PAGESIZE, P_REF);
8853                                }
8854                        }
8855                }
8856
8857                /*
8858                 * Check the shadow list entry after the last page used in
8859                 * this IO request. If it's NOPCACHE_SHWLIST the shadow list
8860                 * was not inserted into pcache and is not large page
8861                 * adjusted.  In this case call reclaim callback directly and
8862                 * don't adjust the shadow list start and size for large
8863                 * pages.
8864                 */
8865                npages = btop(len);
8866                if ((*ppp)[npages] == NOPCACHE_SHWLIST) {
8867                        void *ptag;
8868                        if (pamp != NULL) {
```

```
8869                                    ASSERT(svd->type == MAP_SHARED);
8870                                    ptag = (void *)pamp;
8871                                    paddr = (caddr_t)((addr - seg->s_base) +
8872                                        ptob(svd->anon_index));
8873                            } else {
8874                                    ptag = (void *)seg;
8875                                    paddr = addr;
8876                            }
8877                            (*preclaim_callback)(ptag, paddr, len, *ppp, rw, 0);
8878                    } else {
8879                            ASSERT((*ppp)[npages] == PCACHE_SHWLIST ||
8880                                IS_SWAPFSVP((*ppp)[npages]->p_vnode));
8881                            len = lpgeaddr - lpgaddr;
8882                            npages = btop(len);
8883                            seg_pinactive(seg, pamp, paddr, len,
8884                                *ppp - adjustpages, rw, pflags, preclaim_callback);
8885                    }

8887                    if (pamp != NULL) {
8888                            ASSERT(svd->type == MAP_SHARED);
8889                            ASSERT(svd->softlockcnt >= npages);
8890                            atomic_add_long((ulong_t *)&svd->softlockcnt, -npages);
8891                    }

8893                    if (sftlck_sbase) {
8894                            ASSERT(svd->softlockcnt_sbase > 0);
8895                            atomic_dec_ulong((ulong_t *)&svd->softlockcnt_sbase);
8895                            atomic_add_long((ulong_t *)&svd->softlockcnt_sbase, -1);
8896                    }
8897                    if (sftlck_send) {
8898                            ASSERT(svd->softlockcnt_send > 0);
8899                            atomic_dec_ulong((ulong_t *)&svd->softlockcnt_send);
8899                            atomic_add_long((ulong_t *)&svd->softlockcnt_send, -1);
8900                    }

8902                    /*
8903                     * If someone is blocked while unmapping, we purge
8904                     * segment page cache and thus reclaim pplist synchronously
8905                     * without waiting for seg_pasync_thread. This speeds up
8906                     * unmapping in cases where munmap(2) is called, while
8907                     * raw async i/o is still in progress or where a thread
8908                     * exits on data fault in a multithreaded application.
8909                     */
8910                    if (AS_ISUNMAPWAIT(seg->s_as)) {
8911                            if (svd->softlockcnt == 0) {
8912                                    mutex_enter(&seg->s_as->a_contents);
8913                                    if (AS_ISUNMAPWAIT(seg->s_as)) {
8914                                            AS_CLRUNMAPWAIT(seg->s_as);
8915                                            cv_broadcast(&seg->s_as->a_cv);
8916                                    }
8917                                    mutex_exit(&seg->s_as->a_contents);
8918                            } else if (pamp == NULL) {
8919                                    /*
8920                                     * softlockcnt is not 0 and this is a
8921                                     * MAP_PRIVATE segment. Try to purge its
8922                                     * pcache entries to reduce softlockcnt.
8923                                     * If it drops to 0 segvn_reclaim()
8924                                     * will wake up a thread waiting on
8925                                     * unmapwait flag.
8926                                     *
8927                                     * We don't purge MAP_SHARED segments with non
8928                                     * 0 softlockcnt since IO is still in progress
8929                                     * for such segments.
8930                                     */
8931                                    ASSERT(svd->type == MAP_PRIVATE);
8932                                    segvn_purge(seg);
```



```
8933                            }
8934                    }
8935                    SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
8936                    TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_UNLOCK_END,
8937                        "segvn_pagelock: unlock seg %p addr %p", seg, addr);
8938                    return (0);
8939            }

8941            /* The L_PAGELOCK case ... */

8943            VM_STAT_ADD(segvnvmstats.pagelock[1]);

8945            /*
8946             * For MAP_SHARED segments we have to check protections before
8947             * seg_plookup() since pcache entries may be shared by many segments
8948             * with potentially different page protections.
8949             */
8950            if (pamp != NULL) {
8951                    ASSERT(svd->type == MAP_SHARED);
8952                    if (svd->pageprot == 0) {
8953                            if ((svd->prot & protchk) == 0) {
8954                                    error = EACCES;
8955                                    goto out;
8956                            }
8957                    } else {
8958                            /*
8959                             * check page protections
8960                             */
8961                            caddr_t ea;

8963                            if (seg->s_szc) {
8964                                    a = lpgaddr;
8965                                    ea = lpgeaddr;
8966                            } else {
8967                                    a = addr;
8968                                    ea = addr + len;
8969                            }
8970                            for (; a < ea; a += pgsz) {
8971                                    struct vpage *vp;

8973                                    ASSERT(seg->s_szc == 0 ||
8974                                        sameprot(seg, a, pgsz));
8975                                    vp = &svd->vpage[seg_page(seg, a)];
8976                                    if ((VPP_PROT(vp) & protchk) == 0) {
8977                                            error = EACCES;
8978                                            goto out;
8979                                    }
8980                            }
8981                    }
8982            }

8984            /*
8985             * try to find pages in segment page cache
8986             */
8987            pplist = seg_plookup(seg, pamp, paddr, lpgeaddr - lpgaddr, rw, pflags);
8988            if (pplist != NULL) {
8989                    if (pamp != NULL) {
8990                            npages = btop((uintptr_t)(lpgeaddr - lpgaddr));
8991                            ASSERT(svd->type == MAP_SHARED);
8992                            atomic_add_long((ulong_t *)&svd->softlockcnt,
8993                                npages);
8994                    }
8995                    if (sftlck_sbase) {
8996                            atomic_inc_ulong((ulong_t *)&svd->softlockcnt_sbase);
8996                            atomic_add_long((ulong_t *)&svd->softlockcnt_sbase, 1);
8997                    }
```

```
8998                if (sftlck_send) {
8999                        atomic_inc_ulong((ulong_t *)&svd->softlockcnt_send);
8999                        atomic_add_long((ulong_t *)&svd->softlockcnt_send, 1);
9000                }
9001                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
9002                *ppp = pplist + adjustpages;
9003                TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_HIT_END,
9004                    "segvn_pagelock: cache hit seg %p addr %p", seg, addr);
9005                return (0);
9006        }

9008        /*
9009         * For MAP_SHARED segments we already verified above that segment
9010         * protections allow this pagelock operation.
9011         */
9012        if (pamp == NULL) {
9013                ASSERT(svd->type == MAP_PRIVATE);
9014                if (svd->pageprot == 0) {
9015                        if ((svd->prot & protchk) == 0) {
9016                                error = EACCES;
9017                                goto out;
9018                        }
9019                        if (svd->prot & PROT_WRITE) {
9020                                wlen = lpgeaddr - lpgaddr;
9021                        } else {
9022                                wlen = 0;
9023                                ASSERT(rw == S_READ);
9024                        }
9025                } else {
9026                        int wcont = 1;
9027                        /*
9028                         * check page protections
9029                         */
9030                        for (a = lpgaddr, wlen = 0; a < lpgeaddr; a += pgsz) {
9031                                struct vpage *vp;

9033                                ASSERT(seg->s_szc == 0 ||
9034                                    sameprot(seg, a, pgsz));
9035                                vp = &svd->vpage[seg_page(seg, a)];
9036                                if ((VPP_PROT(vp) & protchk) == 0) {
9037                                        error = EACCES;
9038                                        goto out;
9039                                }
9040                                if (wcont && (VPP_PROT(vp) & PROT_WRITE)) {
9041                                        wlen += pgsz;
9042                                } else {
9043                                        wcont = 0;
9044                                        ASSERT(rw == S_READ);
9045                                }
9046                        }
9047                }
9048                ASSERT(rw == S_READ || wlen == lpgeaddr - lpgaddr);
9049                ASSERT(rw == S_WRITE || wlen <= lpgeaddr - lpgaddr);
9050        }

9052        /*
9053         * Only build large page adjusted shadow list if we expect to insert
9054         * it into pcache. For large enough pages it's a big overhead to
9055         * create a shadow list of the entire large page. But this overhead
9056         * should be amortized over repeated pcache hits on subsequent reuse
9057         * of this shadow list (IO into any range within this shadow list will
9058         * find it in pcache since we large page align the request for pcache
9059         * lookups). pcache performance is improved with bigger shadow lists
9060         * as it reduces the time to pcache the entire big segment and reduces
9061         * pcache chain length.
9062         */
```



```
9063        if (seg_pinsert_check(seg, pamp, paddr,
9064            lpgeaddr - lpgaddr, pflags) == SEGP_SUCCESS) {
9065                addr = lpgaddr;
9066                len = lpgeaddr - lpgaddr;
9067                use_pcache = 1;
9068        } else {
9069                use_pcache = 0;
9070                /*
9071                 * Since this entry will not be inserted into the pcache, we
9072                 * will not do any adjustments to the starting address or
9073                 * size of the memory to be locked.
9074                 */
9075                adjustpages = 0;
9076        }
9077        npages = btop(len);

9079        pplist = kmem_alloc(sizeof (page_t *) * (npages + 1), KM_SLEEP);
9080        pl = pplist;
9081        *ppp = pplist + adjustpages;
9082        /*
9083         * If use_pcache is 0 this shadow list is not large page adjusted.
9084         * Record this info in the last entry of shadow array so that
9085         * L_PAGEUNLOCK can determine if it should large page adjust the
9086         * address range to find the real range that was locked.
9087         */
9088        pl[npages] = use_pcache ? PCACHE_SHWLIST : NOPCACHE_SHWLIST;

9090        page = seg_page(seg, addr);
9091        anon_index = svd->anon_index + page;

9093        anlock = 0;
9094        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
9095        ASSERT(amp->a_szc >= seg->s_szc);
9096        anpgcnt = page_get_pagecnt(amp->a_szc);
9097        for (a = addr; a < addr + len; a += PAGESIZE, anon_index++) {
9098                struct anon *ap;
9099                struct vnode *vp;
9100                u_offset_t off;

9102                /*
9103                 * Lock and unlock anon array only once per large page.
9104                 * anon_array_enter() locks the root anon slot according to
9105                 * a_szc which can't change while anon map is locked.  We lock
9106                 * anon the first time through this loop and each time we
9107                 * reach anon index that corresponds to a root of a large
9108                 * page.
9109                 */
9110                if (a == addr || P2PHASE(anon_index, anpgcnt) == 0) {
9111                        ASSERT(anlock == 0);
9112                        anon_array_enter(amp, anon_index, &cookie);
9113                        anlock = 1;
9114                }
9115                ap = anon_get_ptr(amp->ahp, anon_index);

9117                /*
9118                 * We must never use seg_pcache for COW pages
9119                 * because we might end up with original page still
9120                 * lying in seg_pcache even after private page is
9121                 * created. This leads to data corruption as
9122                 * aio_write refers to the page still in cache
9123                 * while all other accesses refer to the private
9124                 * page.
9125                 */
9126                if (ap == NULL || ap->an_refcnt != 1) {
9127                        struct vpage *vpage;
```

```
9129                        if (seg->s_szc) {
9130                                error = EFAULT;
9131                                break;
9132                        }
9133                        if (svd->vpage != NULL) {
9134                                vpage = &svd->vpage[seg_page(seg, a)];
9135                        } else {
9136                                vpage = NULL;
9137                        }
9138                        ASSERT(anlock);
9139                        anon_array_exit(&cookie);
9140                        anlock = 0;
9141                        pp = NULL;
9142                        error = segvn_faultpage(seg->s_as->a_hat, seg, a, 0,
9143                            vpage, &pp, 0, F_INVAL, rw, 1);
9144                        if (error) {
9145                                error = fc_decode(error);
9146                                break;
9147                        }
9148                        anon_array_enter(amp, anon_index, &cookie);
9149                        anlock = 1;
9150                        ap = anon_get_ptr(amp->ahp, anon_index);
9151                        if (ap == NULL || ap->an_refcnt != 1) {
9152                                error = EFAULT;
9153                                break;
9154                        }
9155                }
9156                swap_xlate(ap, &vp, &off);
9157                pp = page_lookup_nowait(vp, off, SE_SHARED);
9158                if (pp == NULL) {
9159                        error = EFAULT;
9160                        break;
9161                }
9162                if (ap->an_pvp != NULL) {
9163                        anon_swap_free(ap, pp);
9164                }
9165                /*
9166                 * Unlock anon if this is the last slot in a large page.
9167                 */
9168                if (P2PHASE(anon_index, anpgcnt) == anpgcnt - 1) {
9169                        ASSERT(anlock);
9170                        anon_array_exit(&cookie);
9171                        anlock = 0;
9172                }
9173                *pplist++ = pp;
9174        }
9175        if (anlock) {           /* Ensure the lock is dropped */
9176                anon_array_exit(&cookie);
9177        }
9178        ANON_LOCK_EXIT(&amp->a_rwlock);

9180        if (a >= addr + len) {
9181                atomic_add_long((ulong_t *)&svd->softlockcnt, npages);
9182                if (pamp != NULL) {
9183                        ASSERT(svd->type == MAP_SHARED);
9184                        atomic_add_long((ulong_t *)&pamp->a_softlockcnt,
9185                            npages);
9186                        wlen = len;
9187                }
9188                if (sftlck_sbase) {
9189                        atomic_inc_ulong((ulong_t *)&svd->softlockcnt_sbase);
9189                        atomic_add_long((ulong_t *)&svd->softlockcnt_sbase, 1);
9190                }
9191                if (sftlck_send) {
9192                        atomic_inc_ulong((ulong_t *)&svd->softlockcnt_send);
9192                        atomic_add_long((ulong_t *)&svd->softlockcnt_send, 1);
```



```
9193                }
9194                if (use_pcache) {
9195                        (void) seg_pinsert(seg, pamp, paddr, len, wlen, pl,
9196                            rw, pflags, preclaim_callback);
9197                }
9198                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
9199                TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_FILL_END,
9200                    "segvn_pagelock: cache fill seg %p addr %p", seg, addr);
9201                return (0);
9202        }

9204        pplist = pl;
9205        np = ((uintptr_t)(a - addr)) >> PAGESHIFT;
9206        while (np > (uint_t)0) {
9207                ASSERT(PAGE_LOCKED(*pplist));
9208                page_unlock(*pplist);
9209                np--;
9210                pplist++;
9211        }
9212        kmem_free(pl, sizeof (page_t *) * (npages + 1));
9213 out:
9214        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
9215        *ppp = NULL;
9216        TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEGVN_MISS_END,
9217            "segvn_pagelock: cache miss seg %p addr %p", seg, addr);
9218        return (error);
9219 }
```

unchanged_portion_omitted_

```
**********************************************************
  187506 Mon Jul 28 07:44:56 2014
new/usr/src/uts/common/vm/vm_page.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 319 #define MEMSEG_STAT_INCR(v) \
 320         atomic_inc_32(&memseg_stats.v)
 320         atomic_add_32(&memseg_stats.v, 1)
 321 #else
 322 #define MEMSEG_STAT_INCR(x)
 323 #endif

 325 struct memseg *memsegs;         /* list of memory segments */

 327 /*
 328  * /etc/system tunable to control large page allocation hueristic.
 329  *
 330  * Setting to LPAP_LOCAL will heavily prefer the local lgroup over remote lgroup
 331  * for large page allocation requests.  If a large page is not readily
 332  * avaliable on the local freelists we will go through additional effort
 333  * to create a large page, potentially moving smaller pages around to coalesce
 334  * larger pages in the local lgroup.
 335  * Default value of LPAP_DEFAULT will go to remote freelists if large pages
 336  * are not readily available in the local lgroup.
 337  */
 338 enum lpap {
 339         LPAP_DEFAULT,   /* default large page allocation policy */
 340         LPAP_LOCAL      /* local large page allocation policy */
 341 };
______unchanged_portion_omitted_
```

****
  116379 Mon Jul 28 07:44:57 2014
new/usr/src/uts/common/vm/vm_pagelist.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____unchanged_portion_omitted_

```
1470 #ifdef __sparc
1471 /*
1472  * This routine is only used by kcage_init during system startup.
1473  * It performs the function of page_list_sub/PP_SETNORELOC/page_list_add
1474  * without the overhead of taking locks and updating counters.
1475  */
1476 void
1477 page_list_noreloc_startup(page_t *pp)
1478 {
1479         page_t          **ppp;
1480         uint_t          bin;
1481         int             mnode;
1482         int             mtype;
1483         int             flags = 0;

1485         /*
1486          * If this is a large page on the freelist then
1487          * break it up into smaller pages.
1488          */
1489         if (pp->p_szc != 0)
1490                 page_boot_demote(pp);

1492         /*
1493          * Get list page is currently on.
1494          */
1495         bin = PP_2_BIN(pp);
1496         mnode = PP_2_MEM_NODE(pp);
1497         mtype = PP_2_MTYPE(pp);
1498         ASSERT(mtype == MTYPE_RELOC);
1499         ASSERT(pp->p_szc == 0);

1501         if (PP_ISAGED(pp)) {
1502                 ppp = &PAGE_FREELISTS(mnode, 0, bin, mtype);
1503                 flags |= PG_FREE_LIST;
1504         } else {
1505                 ppp = &PAGE_CACHELISTS(mnode, bin, mtype);
1506                 flags |= PG_CACHE_LIST;
1507         }

1509         ASSERT(*ppp != NULL);

1511         /*
1512          * Delete page from current list.
1513          */
1514         if (*ppp == pp)
1515                 *ppp = pp->p_next;              /* go to next page */
1516         if (*ppp == pp) {
1517                 *ppp = NULL;                    /* page list is gone */
1518         } else {
1519                 pp->p_prev->p_next = pp->p_next;
1520                 pp->p_next->p_prev = pp->p_prev;
1521         }

1523         /*
1524          * Decrement page counters
1525          */
1526         page_ctr_sub_internal(mnode, mtype, pp, flags);
```



```
1528         /*
1529          * Set no reloc for cage initted pages.
1530          */
1531         PP_SETNORELOC(pp);

1533         mtype = PP_2_MTYPE(pp);
1534         ASSERT(mtype == MTYPE_NORELOC);

1536         /*
1537          * Get new list for page.
1538          */
1539         if (PP_ISAGED(pp)) {
1540                 ppp = &PAGE_FREELISTS(mnode, 0, bin, mtype);
1541         } else {
1542                 ppp = &PAGE_CACHELISTS(mnode, bin, mtype);
1543         }

1545         /*
1546          * Insert page on new list.
1547          */
1548         if (*ppp == NULL) {
1549                 *ppp = pp;
1550                 pp->p_next = pp->p_prev = pp;
1551         } else {
1552                 pp->p_next = *ppp;
1553                 pp->p_prev = (*ppp)->p_prev;
1554                 (*ppp)->p_prev = pp;
1555                 pp->p_prev->p_next = pp;
1556         }

1558         /*
1559          * Increment page counters
1560          */
1561         page_ctr_add_internal(mnode, mtype, pp, flags);

1563         /*
1564          * Update cage freemem counter
1565          */
1566         atomic_inc_ulong(&kcage_freemem);
1566         atomic_add_long(&kcage_freemem, 1);
1567 }
_____unchanged_portion_omitted_
4120 #define REPL_STAT_INCR(v)       atomic_inc_32(&repl_page_stats.v)
4120 #define REPL_STAT_INCR(v)       atomic_add_32(&repl_page_stats.v, 1)
4121 #else /* REPL_PAGE_STATS */
4122 #define REPL_STAT_INCR(v)
4123 #endif /* REPL_PAGE_STATS */

4125 int     pgrppgcp;

4127 /*
4128  * The freemem accounting must be done by the caller.
4129  * First we try to get a replacement page of the same size as like_pp,
4130  * if that is not possible, then we just get a set of discontiguous
4131  * PAGESIZE pages.
4132  */
4133 page_t *
4134 page_get_replacement_page(page_t *orig_like_pp, struct lgrp *lgrp_target,
4135     uint_t pgrflags)
4136 {
4137         page_t          *like_pp;
4138         page_t          *pp, *pplist;
4139         page_t          *pl = NULL;
4140         ulong_t         bin;
4141         int             mnode, page_mnode;
4142         int             szc;
```

```
4143         spgcnt_t        npgs, pg_cnt;
4144         pfn_t           pfnum;
4145         int             mtype;
4146         int             flags = 0;
4147         lgrp_mnode_cookie_t     lgrp_cookie;
4148         lgrp_t          *lgrp;

4150         REPL_STAT_INCR(ngets);
4151         like_pp = orig_like_pp;
4152         ASSERT(PAGE_EXCL(like_pp));

4154         szc = like_pp->p_szc;
4155         npgs = page_get_pagecnt(szc);
4156         /*
4157          * Now we reset like_pp to the base page_t.
4158          * That way, we won't walk past the end of this 'szc' page.
4159          */
4160         pfnum = PFN_BASE(like_pp->p_pagenum, szc);
4161         like_pp = page_numtopp_nolock(pfnum);
4162         ASSERT(like_pp->p_szc == szc);

4164         if (PP_ISNORELOC(like_pp)) {
4165                 ASSERT(kcage_on);
4166                 REPL_STAT_INCR(ngets_noreloc);
4167                 flags = PGI_RELOCONLY;
4168         } else if (pgrflags & PGR_NORELOC) {
4169                 ASSERT(kcage_on);
4170                 REPL_STAT_INCR(npgr_noreloc);
4171                 flags = PG_NORELOC;
4172         }

4174         /*
4175          * Kernel pages must always be replaced with the same size
4176          * pages, since we cannot properly handle demotion of kernel
4177          * pages.
4178          */
4179         if (PP_ISKAS(like_pp))
4180                 pgrflags |= PGR_SAMESZC;

4182         /* LINTED */
4183         MTYPE_PGR_INIT(mtype, flags, like_pp, page_mnode, npgs);

4185         while (npgs) {
4186                 pplist = NULL;
4187                 for (;;) {
4188                         pg_cnt = page_get_pagecnt(szc);
4189                         bin = PP_2_BIN(like_pp);
4190                         ASSERT(like_pp->p_szc == orig_like_pp->p_szc);
4191                         ASSERT(pg_cnt <= npgs);

4193                         /*
4194                          * If an lgroup was specified, try to get the
4195                          * page from that lgroup.
4196                          * NOTE: Must be careful with code below because
4197                          *       lgroup may disappear and reappear since there
4198                          *       is no locking for lgroup here.
4199                          */
4200                         if (LGRP_EXISTS(lgrp_target)) {
4201                                 /*
4202                                  * Keep local variable for lgroup separate
4203                                  * from lgroup argument since this code should
4204                                  * only be exercised when lgroup argument
4205                                  * exists....
4206                                  */
4207                                 lgrp = lgrp_target;
```



```
4209                                 /* Try the lgroup's freelists first */
4210                                 LGRP_MNODE_COOKIE_INIT(lgrp_cookie, lgrp,
4211                                     LGRP_SRCH_LOCAL);
4212                                 while ((pplist == NULL) &&
4213                                     (mnode = lgrp_memnode_choose(&lgrp_cookie))
4214                                     != -1) {
4215                                         pplist =
4216                                             page_get_mnode_freelist(mnode, bin,
4217                                             mtype, szc, flags);
4218                                 }

4220                                 /*
4221                                  * Now try it's cachelists if this is a
4222                                  * small page. Don't need to do it for
4223                                  * larger ones since page_freelist_coalesce()
4224                                  * already failed.
4225                                  */
4226                                 if (pplist != NULL || szc != 0)
4227                                         break;

4229                                 /* Now try it's cachelists */
4230                                 LGRP_MNODE_COOKIE_INIT(lgrp_cookie, lgrp,
4231                                     LGRP_SRCH_LOCAL);

4233                                 while ((pplist == NULL) &&
4234                                     (mnode = lgrp_memnode_choose(&lgrp_cookie))
4235                                     != -1) {
4236                                         pplist =
4237                                             page_get_mnode_cachelist(bin, flags,
4238                                             mnode, mtype);
4239                                 }
4240                                 if (pplist != NULL) {
4241                                         page_hashout(pplist, NULL);
4242                                         PP_SETAGED(pplist);
4243                                         REPL_STAT_INCR(nhashout);
4244                                         break;
4245                                 }
4246                                 /* Done looking in this lgroup. Bail out. */
4247                                 break;
4248                         }

4250                         /*
4251                          * No lgroup was specified (or lgroup was removed by
4252                          * DR, so just try to get the page as close to
4253                          * like_pp's mnode as possible.
4254                          * First try the local freelist...
4255                          */
4256                         mnode = PP_2_MEM_NODE(like_pp);
4257                         pplist = page_get_mnode_freelist(mnode, bin,
4258                             mtype, szc, flags);
4259                         if (pplist != NULL)
4260                                 break;

4262                         REPL_STAT_INCR(nnofree);

4264                         /*
4265                          * ...then the local cachelist. Don't need to do it for
4266                          * larger pages cause page_freelist_coalesce() already
4267                          * failed there anyway.
4268                          */
4269                         if (szc == 0) {
4270                                 pplist = page_get_mnode_cachelist(bin, flags,
4271                                     mnode, mtype);
4272                                 if (pplist != NULL) {
4273                                         page_hashout(pplist, NULL);
4274                                         PP_SETAGED(pplist);
```

```
4275                                    REPL_STAT_INCR(nhashout);
4276                                    break;
4277                            }
4278                    }

4280                    /* Now try remote freelists */
4281                    page_mnode = mnode;
4282                    lgrp =
4283                        lgrp_hand_to_lgrp(MEM_NODE_2_LGRPHAND(page_mnode));
4284                    LGRP_MNODE_COOKIE_INIT(lgrp_cookie, lgrp,
4285                        LGRP_SRCH_HIER);
4286                    while (pplist == NULL &&
4287                        (mnode = lgrp_memnode_choose(&lgrp_cookie))
4288                        != -1) {
4289                            /*
4290                             * Skip local mnode.
4291                             */
4292                            if ((mnode == page_mnode) ||
4293                                (mem_node_config[mnode].exists == 0))
4294                                    continue;

4296                            pplist = page_get_mnode_freelist(mnode,
4297                                bin, mtype, szc, flags);
4298                    }

4300                    if (pplist != NULL)
4301                            break;


4304                    /* Now try remote cachelists */
4305                    LGRP_MNODE_COOKIE_INIT(lgrp_cookie, lgrp,
4306                        LGRP_SRCH_HIER);
4307                    while (pplist == NULL && szc == 0) {
4308                            mnode = lgrp_memnode_choose(&lgrp_cookie);
4309                            if (mnode == -1)
4310                                    break;
4311                            /*
4312                             * Skip local mnode.
4313                             */
4314                            if ((mnode == page_mnode) ||
4315                                (mem_node_config[mnode].exists == 0))
4316                                    continue;

4318                            pplist = page_get_mnode_cachelist(bin,
4319                                flags, mnode, mtype);

4321                            if (pplist != NULL) {
4322                                    page_hashout(pplist, NULL);
4323                                    PP_SETAGED(pplist);
4324                                    REPL_STAT_INCR(nhashout);
4325                                    break;
4326                            }
4327                    }

4329                    /*
4330                     * Break out of while loop under the following cases:
4331                     * - If we successfully got a page.
4332                     * - If pgrflags specified only returning a specific
4333                     *   page size and we could not find that page size.
4334                     * - If we could not satisfy the request with PAGESIZE
4335                     *   or larger pages.
4336                     */
4337                    if (pplist != NULL || szc == 0)
4338                            break;

4340                    if ((pgrflags & PGR_SAMESZC) || pgrppgcp) {
```



```
4341                                    /* try to find contig page */

4343                                    LGRP_MNODE_COOKIE_INIT(lgrp_cookie, lgrp,
4344                                        LGRP_SRCH_HIER);

4346                                    while ((pplist == NULL) &&
4347                                        (mnode =
4348                                        lgrp_memnode_choose(&lgrp_cookie))
4349                                        != -1) {
4350                                            pplist = page_get_contig_pages(
4351                                                mnode, bin, mtype, szc,
4352                                                flags | PGI_PGCPHIPRI);
4353                                    }
4354                                    break;
4355                            }

4357                            /*
4358                             * The correct thing to do here is try the next
4359                             * page size down using szc--. Due to a bug
4360                             * with the processing of HAT_RELOAD_SHARE
4361                             * where the sfmmu_ttecnt arrays of all
4362                             * hats sharing an ISM segment don't get updated,
4363                             * using intermediate size pages for relocation
4364                             * can lead to continuous page faults.
4365                             */
4366                            szc = 0;
4367                    }

4369                    if (pplist != NULL) {
4370                            DTRACE_PROBE4(page__get,
4371                                lgrp_t *, lgrp,
4372                                int, mnode,
4373                                ulong_t, bin,
4374                                uint_t, flags);

4376                            while (pplist != NULL && pg_cnt--) {
4377                                    ASSERT(pplist != NULL);
4378                                    pp = pplist;
4379                                    page_sub(&pplist, pp);
4380                                    PP_CLRFREE(pp);
4381                                    PP_CLRAGED(pp);
4382                                    page_list_concat(&pl, &pp);
4383                                    npgs--;
4384                                    like_pp = like_pp + 1;
4385                                    REPL_STAT_INCR(nnext_pp);
4386                            }
4387                            ASSERT(pg_cnt == 0);
4388                    } else {
4389                            break;
4390                    }
4391            }

4393            if (npgs) {
4394                    /*
4395                     * We were unable to allocate the necessary number
4396                     * of pages.
4397                     * We need to free up any pl.
4398                     */
4399                    REPL_STAT_INCR(nnopage);
4400                    page_free_replacement_page(pl);
4401                    return (NULL);
4402            } else {
4403                    return (pl);
4404            }
4405 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   50383 Mon Jul 28 07:44:57 2014
new/usr/src/uts/common/xen/io/xnb.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1267 static int
1268 xnb_txbuf_constructor(void *buf, void *arg, int kmflag)
1269 {
1270         _NOTE(ARGUNUSED(kmflag));
1271         xnb_txbuf_t *txp = buf;
1272         xnb_t *xnbp = arg;
1273         size_t len;
1274         ddi_dma_cookie_t dma_cookie;
1275         uint_t ncookies;

1277         txp->xt_free_rtn.free_func = xnb_txbuf_recycle;
1278         txp->xt_free_rtn.free_arg = (caddr_t)txp;
1279         txp->xt_xnbp = xnbp;
1280         txp->xt_next = NULL;

1282         if (ddi_dma_alloc_handle(xnbp->xnb_devinfo, &buf_dma_attr,
1283             0, 0, &txp->xt_dma_handle) != DDI_SUCCESS)
1284                 goto failure;

1286         if (ddi_dma_mem_alloc(txp->xt_dma_handle, PAGESIZE, &data_accattr,
1287             DDI_DMA_STREAMING, 0, 0, &txp->xt_buf, &len,
1288             &txp->xt_acc_handle) != DDI_SUCCESS)
1289                 goto failure_1;

1291         if (ddi_dma_addr_bind_handle(txp->xt_dma_handle, NULL, txp->xt_buf,
1292             len, DDI_DMA_RDWR | DDI_DMA_STREAMING, DDI_DMA_DONTWAIT, 0,
1293             &dma_cookie, &ncookies)
1294             != DDI_DMA_MAPPED)
1295                 goto failure_2;
1296         ASSERT(ncookies == 1);

1298         txp->xt_mfn = xnb_btop(dma_cookie.dmac_laddress);
1299         txp->xt_buflen = dma_cookie.dmac_size;

1301         DTRACE_PROBE(txbuf_allocated);

1303         atomic_inc_32(&xnbp->xnb_tx_buf_count);
1303         atomic_add_32(&xnbp->xnb_tx_buf_count, 1);
1304         xnbp->xnb_tx_buf_outstanding++;

1306         return (0);

1308 failure_2:
1309         ddi_dma_mem_free(&txp->xt_acc_handle);

1311 failure_1:
1312         ddi_dma_free_handle(&txp->xt_dma_handle);

1314 failure:

1316         return (-1);
1317 }

1319 static void
1320 xnb_txbuf_destructor(void *buf, void *arg)
1321 {
1322         xnb_txbuf_t *txp = buf;
1323         xnb_t *xnbp = arg;
```



```
1325         (void) ddi_dma_unbind_handle(txp->xt_dma_handle);
1326         ddi_dma_mem_free(&txp->xt_acc_handle);
1327         ddi_dma_free_handle(&txp->xt_dma_handle);

1329         atomic_dec_32(&xnbp->xnb_tx_buf_count);
1329         atomic_add_32(&xnbp->xnb_tx_buf_count, -1);
1330 }
______unchanged_portion_omitted_
```

****
66766 Mon Jul 28 07:44:57 2014
new/usr/src/uts/common/xen/io/xnf.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
 348 /*
 349  * Acquire a grant reference.
 350  */
 351 static grant_ref_t
 352 gref_get(xnf_t *xnfp)
 353 {
 354         grant_ref_t gref;

 356         mutex_enter(&xnfp->xnf_gref_lock);

 358         do {
 359                 gref = gnttab_claim_grant_reference(&xnfp->xnf_gref_head);

 361         } while ((gref == INVALID_GRANT_REF) &&
 362             (gnttab_alloc_grant_references(16, &xnfp->xnf_gref_head) == 0));

 364         mutex_exit(&xnfp->xnf_gref_lock);

 366         if (gref == INVALID_GRANT_REF) {
 367                 xnfp->xnf_stat_gref_failure++;
 368         } else {
 369                 atomic_inc_64(&xnfp->xnf_stat_gref_outstanding);
 369                 atomic_add_64(&xnfp->xnf_stat_gref_outstanding, 1);
 370                 if (xnfp->xnf_stat_gref_outstanding > xnfp->xnf_stat_gref_peak)
 371                         xnfp->xnf_stat_gref_peak =
 372                             xnfp->xnf_stat_gref_outstanding;
 373         }

 375         return (gref);
 376 }

 378 /*
 379  * Release a grant reference.
 380  */
 381 static void
 382 gref_put(xnf_t *xnfp, grant_ref_t gref)
 383 {
 384         ASSERT(gref != INVALID_GRANT_REF);

 386         mutex_enter(&xnfp->xnf_gref_lock);
 387         gnttab_release_grant_reference(&xnfp->xnf_gref_head, gref);
 388         mutex_exit(&xnfp->xnf_gref_lock);

 390         atomic_dec_64(&xnfp->xnf_stat_gref_outstanding);
 390         atomic_add_64(&xnfp->xnf_stat_gref_outstanding, -1);
 391 }
```

_____***unchanged_portion_omitted_***

```
2315 static int
2316 xnf_buf_constructor(void *buf, void *arg, int kmflag)
2317 {
2318         int (*ddiflags)(caddr_t) = DDI_DMA_SLEEP;
2319         xnf_buf_t *bdesc = buf;
2320         xnf_t *xnfp = arg;
2321         ddi_dma_cookie_t dma_cookie;
2322         uint_t ncookies;
2323         size_t len;

2325         if (kmflag & KM_NOSLEEP)
```



```
2326                 ddiflags = DDI_DMA_DONTWAIT;

2328         /* Allocate a DMA access handle for the buffer. */
2329         if (ddi_dma_alloc_handle(xnfp->xnf_devinfo, &buf_dma_attr,
2330             ddiflags, 0, &bdesc->dma_handle) != DDI_SUCCESS)
2331                 goto failure;

2333         /* Allocate DMA-able memory for buffer. */
2334         if (ddi_dma_mem_alloc(bdesc->dma_handle,
2335             PAGESIZE, &data_accattr, DDI_DMA_STREAMING, ddiflags, 0,
2336             &bdesc->buf, &len, &bdesc->acc_handle) != DDI_SUCCESS)
2337                 goto failure_1;

2339         /* Bind to virtual address of buffer to get physical address. */
2340         if (ddi_dma_addr_bind_handle(bdesc->dma_handle, NULL,
2341             bdesc->buf, len, DDI_DMA_RDWR | DDI_DMA_STREAMING,
2342             ddiflags, 0, &dma_cookie, &ncookies) != DDI_DMA_MAPPED)
2343                 goto failure_2;
2344         ASSERT(ncookies == 1);

2346         bdesc->free_rtn.free_func = xnf_buf_recycle;
2347         bdesc->free_rtn.free_arg = (caddr_t)bdesc;
2348         bdesc->xnfp = xnfp;
2349         bdesc->buf_phys = dma_cookie.dmac_laddress;
2350         bdesc->buf_mfn = pfn_to_mfn(xnf_btop(bdesc->buf_phys));
2351         bdesc->len = dma_cookie.dmac_size;
2352         bdesc->grant_ref = INVALID_GRANT_REF;
2353         bdesc->gen = xnfp->xnf_gen;

2355         atomic_inc_64(&xnfp->xnf_stat_buf_allocated);
2355         atomic_add_64(&xnfp->xnf_stat_buf_allocated, 1);

2357         return (0);

2359 failure_2:
2360         ddi_dma_mem_free(&bdesc->acc_handle);

2362 failure_1:
2363         ddi_dma_free_handle(&bdesc->dma_handle);

2365 failure:

2367         ASSERT(kmflag & KM_NOSLEEP); /* Cannot fail for KM_SLEEP. */
2368         return (-1);
2369 }

2371 static void
2372 xnf_buf_destructor(void *buf, void *arg)
2373 {
2374         xnf_buf_t *bdesc = buf;
2375         xnf_t *xnfp = arg;

2377         (void) ddi_dma_unbind_handle(bdesc->dma_handle);
2378         ddi_dma_mem_free(&bdesc->acc_handle);
2379         ddi_dma_free_handle(&bdesc->dma_handle);

2381         atomic_dec_64(&xnfp->xnf_stat_buf_allocated);
2381         atomic_add_64(&xnfp->xnf_stat_buf_allocated, -1);
2382 }

2384 static xnf_buf_t *
2385 xnf_buf_get(xnf_t *xnfp, int flags, boolean_t readonly)
2386 {
2387         grant_ref_t gref;
2388         xnf_buf_t *bufp;
```

```
2390         /*
2391          * Usually grant references are more scarce than memory, so we
2392          * attempt to acquire a grant reference first.
2393          */
2394         gref = gref_get(xnfp);
2395         if (gref == INVALID_GRANT_REF)
2396                 return (NULL);

2398         bufp = kmem_cache_alloc(xnfp->xnf_buf_cache, flags);
2399         if (bufp == NULL) {
2400                 gref_put(xnfp, gref);
2401                 return (NULL);
2402         }

2404         ASSERT(bufp->grant_ref == INVALID_GRANT_REF);

2406         bufp->grant_ref = gref;

2408         if (bufp->gen != xnfp->xnf_gen)
2409                 xnf_buf_refresh(bufp);

2411         gnttab_grant_foreign_access_ref(bufp->grant_ref,
2412             xvdi_get_oeid(bufp->xnfp->xnf_devinfo),
2413             bufp->buf_mfn, readonly ? 1 : 0);

2415         atomic_inc_64(&xnfp->xnf_stat_buf_outstanding);
2415         atomic_add_64(&xnfp->xnf_stat_buf_outstanding, 1);

2417         return (bufp);
2418 }

2420 static void
2421 xnf_buf_put(xnf_t *xnfp, xnf_buf_t *bufp, boolean_t readonly)
2422 {
2423         if (bufp->grant_ref != INVALID_GRANT_REF) {
2424                 (void) gnttab_end_foreign_access_ref(
2425                     bufp->grant_ref, readonly ? 1 : 0);
2426                 gref_put(xnfp, bufp->grant_ref);
2427                 bufp->grant_ref = INVALID_GRANT_REF;
2428         }

2430         kmem_cache_free(xnfp->xnf_buf_cache, bufp);

2432         atomic_dec_64(&xnfp->xnf_stat_buf_outstanding);
2432         atomic_add_64(&xnfp->xnf_stat_buf_outstanding, -1);
2433 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   25915 Mon Jul 28 07:44:58 2014
new/usr/src/uts/i86pc/io/psm/uppc.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 310 /*ARGSUSED3*/
 311 static int
 312 uppc_addspl(int irqno, int ipl, int min_ipl, int max_ipl)
 313 {
 314         struct standard_pic *pp;
 315         int i;
 316         int startidx;
 317         uchar_t vectmask;

 319         if (irqno <= MAX_ISA_IRQ)
 320                 atomic_inc_16(&uppc_irq_shared_table[irqno]);
 320                 atomic_add_16(&uppc_irq_shared_table[irqno], 1);

 322         if (ipl != min_ipl)
 323                 return (0);

 325         if (irqno > 7) {
 326                 vectmask = 1 << (irqno - 8);
 327                 startidx = (ipl << 1);
 328         } else {
 329                 vectmask = 1 << irqno;
 330                 startidx = (ipl << 1) + 1;
 331         }

 333         /*
 334          * mask intr same or above ipl
 335          * level MAXIPL has all intr off as init. default
 336          */
 337         pp = &pics0;
 338         for (i = startidx; i < (MAXIPL << 1); i += 2) {
 339                 if (pp->c_iplmask[i] & vectmask)
 340                         break;
 341                 pp->c_iplmask[i] |= vectmask;
 342         }

 344         /*
 345          * unmask intr below ipl
 346          */
 347         for (i = startidx-2; i >= 0; i -= 2) {
 348                 if (!(pp->c_iplmask[i] & vectmask))
 349                         break;
 350                 pp->c_iplmask[i] &= ~vectmask;
 351         }
 352         return (0);
 353 }

 355 static int
 356 uppc_delspl(int irqno, int ipl, int min_ipl, int max_ipl)
 357 {
 358         struct standard_pic *pp;
 359         int i;
 360         uchar_t vectmask;

 362         if (irqno <= MAX_ISA_IRQ)
 363                 atomic_dec_16(&uppc_irq_shared_table[irqno]);
 363                 atomic_add_16(&uppc_irq_shared_table[irqno], -1);

 365         /*
 366          * skip if we are not deleting the last handler
```



```
 367          * and the ipl is higher than minimum
 368          */
 369         if ((max_ipl != PSM_INVALID_IPL) && (ipl >= min_ipl))
 370                 return (0);

 372         if (irqno > 7) {
 373                 vectmask = 1 << (irqno - 8);
 374                 i = 0;
 375         } else {
 376                 vectmask = 1 << irqno;
 377                 i = 1;
 378         }

 380         pp = &pics0;

 382         /*
 383          * check any handlers left for this irqno
 384          */
 385         if (max_ipl != PSM_INVALID_IPL) {
 386                 /*
 387                  * unmasks all levels below the lowest priority
 388                  */
 389                 i += ((min_ipl - 1) << 1);
 390                 for (; i >= 0; i -= 2) {
 391                         if (!(pp->c_iplmask[i] & vectmask))
 392                                 break;
 393                         pp->c_iplmask[i] &= ~vectmask;
 394                 }
 395         } else {
 396                 /*
 397                  * set mask to all levels
 398                  */
 399                 for (; i < (MAXIPL << 1); i += 2) {
 400                         if (pp->c_iplmask[i] & vectmask)
 401                                 break;
 402                         pp->c_iplmask[i] |= vectmask;
 403                 }
 404         }
 405         return (0);
 406 }
______unchanged_portion_omitted_
```

```
**********************************************************
    7121 Mon Jul 28 07:44:58 2014
new/usr/src/uts/i86pc/os/memnode.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/systm.h>
#include <sys/sysmacros.h>
#include <sys/bootconf.h>
#include <sys/atomic.h>
#include <sys/lgrp.h>
#include <sys/memlist.h>
#include <sys/memnode.h>
#include <sys/platform_module.h>
#include <vm/vm_dep.h>

int     max_mem_nodes = 1;

struct mem_node_conf mem_node_config[MAX_MEM_NODES];
int mem_node_pfn_shift;
/*
 * num_memnodes should be updated atomically and always >=
 * the number of bits in memnodes_mask or the algorithm may fail.
 */
uint16_t num_memnodes;
mnodeset_t memnodes_mask; /* assumes 8*(sizeof(mnodeset_t)) >= MAX_MEM_NODES */

/*
 * If set, mem_node_physalign should be a power of two, and
 * should reflect the minimum address alignment of each node.
 */
uint64_t mem_node_physalign;

/*
 * Platform hooks we will need.
 */

#pragma weak plat_build_mem_nodes
#pragma weak plat_slice_add
#pragma weak plat_slice_del

/*
```



```
 * Adjust the memnode config after a DR operation.
 *
 * It is rather tricky to do these updates since we can't
 * protect the memnode structures with locks, so we must
 * be mindful of the order in which updates and reads to
 * these values can occur.
 */

void
mem_node_add_slice(pfn_t start, pfn_t end)
{
        int mnode;
        mnodeset_t newmask, oldmask;

        /*
         * DR will pass us the first pfn that is allocatable.
         * We need to round down to get the real start of
         * the slice.
         */
        if (mem_node_physalign) {
                start &= ~(btop(mem_node_physalign) - 1);
                end = roundup(end, btop(mem_node_physalign)) - 1;
        }

        mnode = PFN_2_MEM_NODE(start);
        ASSERT(mnode >= 0 && mnode < max_mem_nodes);

        if (atomic_cas_32((uint32_t *)&mem_node_config[mnode].exists, 0, 1)) {
                /*
                 * Add slice to existing node.
                 */
                if (start < mem_node_config[mnode].physbase)
                        mem_node_config[mnode].physbase = start;
                if (end > mem_node_config[mnode].physmax)
                        mem_node_config[mnode].physmax = end;
        } else {
                mem_node_config[mnode].physbase = start;
                mem_node_config[mnode].physmax = end;
                atomic_inc_16(&num_memnodes);
                atomic_add_16(&num_memnodes, 1);
                do {
                        oldmask = memnodes_mask;
                        newmask = memnodes_mask | (1ull << mnode);
                } while (atomic_cas_64(&memnodes_mask, oldmask, newmask) !=
                    oldmask);
        }

        /*
         * Inform the common lgrp framework about the new memory
         */
        lgrp_config(LGRP_CONFIG_MEM_ADD, mnode, MEM_NODE_2_LGRPHAND(mnode));
}

/*
 * Remove a PFN range from a memnode.  On some platforms,
 * the memnode will be created with physbase at the first
 * allocatable PFN, but later deleted with the MC slice
 * base address converted to a PFN, in which case we need
 * to assume physbase and up.
 */
void
mem_node_del_slice(pfn_t start, pfn_t end)
{
        int mnode;
        pgcnt_t delta_pgcnt, node_size;
        mnodeset_t omask, nmask;
```

```
128         if (mem_node_physalign) {
129                 start &= ~(btop(mem_node_physalign) - 1);
130                 end = roundup(end, btop(mem_node_physalign)) - 1;
131         }
132         mnode = PFN_2_MEM_NODE(start);

134         ASSERT(mnode >= 0 && mnode < max_mem_nodes);
135         ASSERT(mem_node_config[mnode].exists == 1);

137         delta_pgcnt = end - start;
138         node_size = mem_node_config[mnode].physmax -
139             mem_node_config[mnode].physbase;

141         if (node_size > delta_pgcnt) {
142                 /*
143                  * Subtract the slice from the memnode.
144                  */
145                 if (start <= mem_node_config[mnode].physbase)
146                         mem_node_config[mnode].physbase = end + 1;
147                 ASSERT(end <= mem_node_config[mnode].physmax);
148                 if (end == mem_node_config[mnode].physmax)
149                         mem_node_config[mnode].physmax = start - 1;
150         } else {
151                 /*
152                  * Let the common lgrp framework know this mnode is
153                  * leaving
154                  */
155                 lgrp_config(LGRP_CONFIG_MEM_DEL,
156                     mnode, MEM_NODE_2_LGRPHAND(mnode));

158                 /*
159                  * Delete the whole node.
160                  */
161                 ASSERT(MNODE_PGCNT(mnode) == 0);
162                 do {
163                         omask = memnodes_mask;
164                         nmask = omask & ~(1ull << mnode);
165                 } while (atomic_cas_64(&memnodes_mask, omask, nmask) != omask);
166                 atomic_dec_16(&num_memnodes);
166                 atomic_add_16(&num_memnodes, -1);
167                 mem_node_config[mnode].exists = 0;
168         }
169 }
```

_____***unchanged_portion_omitted_***

```
219 /*
220  * Allocate an unassigned memnode.
221  */
222 int
223 mem_node_alloc()
224 {
225         int mnode;
226         mnodeset_t newmask, oldmask;

228         /*
229          * Find an unused memnode.  Update it atomically to prevent
230          * a first time memnode creation race.
231          */
232         for (mnode = 0; mnode < max_mem_nodes; mnode++)
233                 if (atomic_cas_32((uint32_t *)&mem_node_config[mnode].exists,
234                     0, 1) == 0)
235                         break;

237         if (mnode >= max_mem_nodes)
238                 panic("Out of free memnodes\n");
```



```
240         mem_node_config[mnode].physbase = (pfn_t)-1l;
241         mem_node_config[mnode].physmax = 0;
242         atomic_inc_16(&num_memnodes);
242         atomic_add_16(&num_memnodes, 1);
243         do {
244                 oldmask = memnodes_mask;
245                 newmask = memnodes_mask | (1ull << mnode);
246         } while (atomic_cas_64(&memnodes_mask, oldmask, newmask) != oldmask);

248         return (mnode);
249 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   13629 Mon Jul 28 07:44:58 2014
new/usr/src/uts/i86pc/sys/rootnex.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
  23  */

  25 #ifndef _SYS_ROOTNEX_H
  26 #define _SYS_ROOTNEX_H

  28 /*
  29  * x86 root nexus implementation specific state
  30  */

  32 #include <sys/types.h>
  33 #include <sys/conf.h>
  34 #include <sys/modctl.h>
  35 #include <sys/sunddi.h>
  36 #include <sys/iommulib.h>
  37 #include <sys/sdt.h>

  39 #ifdef  __cplusplus
  40 extern "C" {
  41 #endif


  44 /* size of buffer used for ctlop reportdev */
  45 #define REPORTDEV_BUFSIZE       1024

  47 /* min and max interrupt vectors */
  48 #define VEC_MIN                 1
  49 #define VEC_MAX                 255

  51 /* atomic increment/decrement to keep track of outstanding binds, etc */
  52 #ifdef DEBUG
  53 #define ROOTNEX_DPROF_INC(addr)         atomic_inc_64(addr)
  54 #define ROOTNEX_DPROF_DEC(addr)         atomic_dec_64(addr)
  54 #define ROOTNEX_DPROF_DEC(addr)         atomic_add_64(addr, -1)
  55 #define ROOTNEX_DPROBE1(name, type1, arg1) \
  56         DTRACE_PROBE1(name, type1, arg1)
  57 #define ROOTNEX_DPROBE2(name, type1, arg1, type2, arg2) \
  58         DTRACE_PROBE2(name, type1, arg1, type2, arg2)
  59 #define ROOTNEX_DPROBE3(name, type1, arg1, type2, arg2, type3, arg3) \
  60         DTRACE_PROBE3(name, type1, arg1, type2, arg2, type3, arg3)
```



```
  61 #define ROOTNEX_DPROBE4(name, type1, arg1, type2, arg2, type3, arg3, \
  62     type4, arg4) \
  63         DTRACE_PROBE4(name, type1, arg1, type2, arg2, type3, arg3, type4, arg4)
  64 #else
  65 #define ROOTNEX_DPROF_INC(addr)
  66 #define ROOTNEX_DPROF_DEC(addr)
  67 #define ROOTNEX_DPROBE1(name, type1, arg1)
  68 #define ROOTNEX_DPROBE2(name, type1, arg1, type2, arg2)
  69 #define ROOTNEX_DPROBE3(name, type1, arg1, type2, arg2, type3, arg3)
  70 #define ROOTNEX_DPROBE4(name, type1, arg1, type2, arg2, type3, arg3, \
  71     type4, arg4)
  72 #endif

  74 /* set in dmac_type to signify that this cookie uses the copy buffer */
  75 #define ROOTNEX_USES_COPYBUF            0x80000000

  77 /*
  78  * integer or boolean property name and value. A few static rootnex properties
  79  * are created during rootnex attach from an array of rootnex_intprop_t..
  80  */
  81 typedef struct rootnex_intprop_s {
  82         char    *prop_name;
  83         int     prop_value;
  84 } rootnex_intprop_t;
______unchanged_portion_omitted_
```

```
**********************************************************
  106505 Mon Jul 28 07:44:58 2014
new/usr/src/uts/i86pc/vm/hat_i86.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1224 /*
1225  * enable/disable collection of stats for hat.
1226  */
1227 int
1228 hat_stats_enable(hat_t *hat)
1229 {
1230         atomic_inc_32(&hat->hat_stats);
1230         atomic_add_32(&hat->hat_stats, 1);
1231         return (1);
1232 }

1234 void
1235 hat_stats_disable(hat_t *hat)
1236 {
1237         atomic_dec_32(&hat->hat_stats);
1237         atomic_add_32(&hat->hat_stats, -1);
1238 }
______unchanged_portion_omitted_
```

```
**********************************************************
    8082 Mon Jul 28 07:44:58 2014
new/usr/src/uts/i86pc/vm/hat_i86.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_
  98 typedef struct hat hat_t;

 100 #define PGCNT_INC(hat, level)   \
 101         atomic_inc_ulong(&(hat)->hat_pages_mapped[level]);
 101         atomic_add_long(&(hat)->hat_pages_mapped[level], 1);
 102 #define PGCNT_DEC(hat, level)   \
 103         atomic_dec_ulong(&(hat)->hat_pages_mapped[level]);
 103         atomic_add_long(&(hat)->hat_pages_mapped[level], -1);

 105 /*
 106  * Flags for the hat_flags field
 107  *
 108  * HAT_FREEING - set when HAT is being destroyed - mostly used to detect that
 109  *       demap()s can be avoided.
 110  *
 111  * HAT_VLP - indicates a 32 bit process has a virtual address range less than
 112  *       the hardware's physical address range. (VLP->Virtual Less-than Physical)
 113  *       Note - never used on the hypervisor.
 114  *
 115  * HAT_VICTIM - This is set while a hat is being examined for page table
 116  *       stealing and prevents it from being freed.
 117  *
 118  * HAT_SHARED - The hat has exported it's page tables via hat_share()
 119  *
 120  * HAT_PINNED - On the hypervisor, indicates the top page table has been pinned.
 121  */
 122 #define HAT_FREEING     (0x0001)
 123 #define HAT_VLP         (0x0002)
 124 #define HAT_VICTIM      (0x0004)
 125 #define HAT_SHARED      (0x0008)
 126 #define HAT_PINNED      (0x0010)

 128 /*
 129  * Additional platform attribute for hat_devload() to force no caching.
 130  */
 131 #define HAT_PLAT_NOCACHE        (0x100000)

 133 /*
 134  * Simple statistics for the HAT. These are just counters that are
 135  * atomically incremented. They can be reset directly from the kernel
 136  * debugger.
 137  */
 138 struct hatstats {
 139         ulong_t hs_reap_attempts;
 140         ulong_t hs_reaped;
 141         ulong_t hs_steals;
 142         ulong_t hs_ptable_allocs;
 143         ulong_t hs_ptable_frees;
 144         ulong_t hs_htable_rgets;        /* allocs from reserve */
 145         ulong_t hs_htable_rputs;        /* putbacks to reserve */
 146         ulong_t hs_htable_shared;       /* number of htables shared */
 147         ulong_t hs_htable_unshared;     /* number of htables unshared */
 148         ulong_t hs_hm_alloc;
 149         ulong_t hs_hm_free;
 150         ulong_t hs_hm_put_reserve;
 151         ulong_t hs_hm_get_reserve;
 152         ulong_t hs_hm_steals;
 153         ulong_t hs_hm_steal_exam;
 154         ulong_t hs_tlb_inval_delayed;
 155 };
______unchanged_portion_omitted_
```

```
**********************************************************
   58262 Mon Jul 28 07:44:59 2014
new/usr/src/uts/i86pc/vm/htable.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_
 258 #endif /* __xpv */

 260 /*
 261  * Allocate a memory page for a hardware page table.
 262  *
 263  * A wrapper around page_get_physical(), with some extra checks.
 264  */
 265 static pfn_t
 266 ptable_alloc(uintptr_t seed)
 267 {
 268         pfn_t pfn;
 269         page_t *pp;

 271         pfn = PFN_INVALID;

 273         /*
 274          * The first check is to see if there is memory in the system. If we
 275          * drop to throttlefree, then fail the ptable_alloc() and let the
 276          * stealing code kick in. Note that we have to do this test here,
 277          * since the test in page_create_throttle() would let the NOSLEEP
 278          * allocation go through and deplete the page reserves.
 279          *
 280          * The !NOMEMWAIT() lets pageout, fsflush, etc. skip this check.
 281          */
 282         if (!NOMEMWAIT() && freemem <= throttlefree + 1)
 283                 return (PFN_INVALID);

 285 #ifdef DEBUG
 286         /*
 287          * This code makes htable_steal() easier to test. By setting
 288          * force_steal we force pagetable allocations to fall
 289          * into the stealing code. Roughly 1 in ever "force_steal"
 290          * page table allocations will fail.
 291          */
 292         if (proc_pageout != NULL && force_steal > 1 &&
 293             ++ptable_cnt > force_steal) {
 294                 ptable_cnt = 0;
 295                 return (PFN_INVALID);
 296         }
 297 #endif /* DEBUG */

 299         pp = page_get_physical(seed);
 300         if (pp == NULL)
 301                 return (PFN_INVALID);
 302         ASSERT(PAGE_SHARED(pp));
 303         pfn = pp->p_pagenum;
 304         if (pfn == PFN_INVALID)
 305                 panic("ptable_alloc(): Invalid PFN!!");
 306         atomic_inc_32(&active_ptables);
 306         atomic_add_32(&active_ptables, 1);
 307         HATSTAT_INC(hs_ptable_allocs);
 308         return (pfn);
 309 }

 311 /*
 312  * Free an htable's associated page table page.  See the comments
 313  * for ptable_alloc().
 314  */
 315 static void
 316 ptable_free(pfn_t pfn)
```



```
 317 {
 318         page_t *pp = page_numtopp_nolock(pfn);

 320         /*
 321          * need to destroy the page used for the pagetable
 322          */
 323         ASSERT(pfn != PFN_INVALID);
 324         HATSTAT_INC(hs_ptable_frees);
 325         atomic_dec_32(&active_ptables);
 325         atomic_add_32(&active_ptables, -1);
 326         if (pp == NULL)
 327                 panic("ptable_free(): no page for pfn!");
 328         ASSERT(PAGE_SHARED(pp));
 329         ASSERT(pfn == pp->p_pagenum);
 330         ASSERT(!IN_XPV_PANIC());

 332         /*
 333          * Get an exclusive lock, might have to wait for a kmem reader.
 334          */
 335         if (!page_tryupgrade(pp)) {
 336                 u_offset_t off = pp->p_offset;
 337                 page_unlock(pp);
 338                 pp = page_lookup(&kvp, off, SE_EXCL);
 339                 if (pp == NULL)
 340                         panic("page not found");
 341         }
 342 #ifdef __xpv
 343         if (kpm_vbase && xen_kpm_page(pfn, PT_VALID | PT_WRITABLE) < 0)
 344                 panic("failure making kpm r/w pfn=0x%lx", pfn);
 345 #endif
 346         page_hashout(pp, NULL);
 347         page_free(pp, 1);
 348         page_unresv(1);
 349 }
______unchanged_portion_omitted_


 430 /*
 431  * This routine steals htables from user processes for htable_alloc() or
 432  * for htable_reap().
 433  */
 434 static htable_t *
 435 htable_steal(uint_t cnt)
 436 {
 437         hat_t           *hat = kas.a_hat;       /* list starts with khat */
 438         htable_t        *list = NULL;
 439         htable_t        *ht;
 440         htable_t        *higher;
 441         uint_t          h;
 442         uint_t          h_start;
 443         static uint_t   h_seed = 0;
 444         uint_t          e;
 445         uintptr_t       va;
 446         x86pte_t        pte;
 447         uint_t          stolen = 0;
 448         uint_t          pass;
 449         uint_t          threshold;

 451         /*
 452          * Limit htable_steal_passes to something reasonable
 453          */
 454         if (htable_steal_passes == 0)
 455                 htable_steal_passes = 1;
 456         if (htable_steal_passes > mmu.ptes_per_table)
 457                 htable_steal_passes = mmu.ptes_per_table;
```

```
459         /*
460          * Loop through all user hats. The 1st pass takes cached htables that
461          * aren't in use. The later passes steal by removing mappings, too.
462          */
463         atomic_inc_32(&htable_dont_cache);
463         atomic_add_32(&htable_dont_cache, 1);
464         for (pass = 0; pass <= htable_steal_passes && stolen < cnt; ++pass) {
465                 threshold = pass * mmu.ptes_per_table / htable_steal_passes;
466                 hat = kas.a_hat;
467                 for (;;) {
469                         /*
470                          * Clear the victim flag and move to next hat
471                          */
472                         mutex_enter(&hat_list_lock);
473                         if (hat != kas.a_hat) {
474                                 hat->hat_flags &= ~HAT_VICTIM;
475                                 cv_broadcast(&hat_list_cv);
476                         }
477                         hat = hat->hat_next;
479                         /*
480                          * Skip any hat that is already being stolen from.
481                          *
482                          * We skip SHARED hats, as these are dummy
483                          * hats that host ISM shared page tables.
484                          *
485                          * We also skip if HAT_FREEING because hat_pte_unmap()
486                          * won't zero out the PTE's. That would lead to hitting
487                          * stale PTEs either here or under hat_unload() when we
488                          * steal and unload the same page table in competing
489                          * threads.
490                          */
491                         while (hat != NULL &&
492                             (hat->hat_flags &
493                             (HAT_VICTIM | HAT_SHARED | HAT_FREEING)) != 0)
494                                 hat = hat->hat_next;
496                         if (hat == NULL) {
497                                 mutex_exit(&hat_list_lock);
498                                 break;
499                         }
501                         /*
502                          * Are we finished?
503                          */
504                         if (stolen == cnt) {
505                                 /*
506                                  * Try to spread the pain of stealing,
507                                  * move victim HAT to the end of the HAT list.
508                                  */
509                                 if (pass >= 1 && cnt == 1 &&
510                                     kas.a_hat->hat_prev != hat) {
512                                         /* unlink victim hat */
513                                         if (hat->hat_prev)
514                                                 hat->hat_prev->hat_next =
515                                                     hat->hat_next;
516                                         else
517                                                 kas.a_hat->hat_next =
518                                                     hat->hat_next;
519                                         if (hat->hat_next)
520                                                 hat->hat_next->hat_prev =
521                                                     hat->hat_prev;
522                                         else
523                                                 kas.a_hat->hat_prev =
```



```
524                                                     hat->hat_prev;
527                                         /* relink at end of hat list */
528                                         hat->hat_next = NULL;
529                                         hat->hat_prev = kas.a_hat->hat_prev;
530                                         if (hat->hat_prev)
531                                                 hat->hat_prev->hat_next = hat;
532                                         else
533                                                 kas.a_hat->hat_next = hat;
534                                         kas.a_hat->hat_prev = hat;
536                                 }
538                                 mutex_exit(&hat_list_lock);
539                                 break;
540                         }
542                         /*
543                          * Mark the HAT as a stealing victim.
544                          */
545                         hat->hat_flags |= HAT_VICTIM;
546                         mutex_exit(&hat_list_lock);
548                         /*
549                          * Take any htables from the hat's cached "free" list.
550                          */
551                         hat_enter(hat);
552                         while ((ht = hat->hat_ht_cached) != NULL &&
553                             stolen < cnt) {
554                                 hat->hat_ht_cached = ht->ht_next;
555                                 ht->ht_next = list;
556                                 list = ht;
557                                 ++stolen;
558                         }
559                         hat_exit(hat);
561                         /*
562                          * Don't steal on first pass.
563                          */
564                         if (pass == 0 || stolen == cnt)
565                                 continue;
567                         /*
568                          * Search the active htables for one to steal.
569                          * Start at a different hash bucket every time to
570                          * help spread the pain of stealing.
571                          */
572                         h = h_start = h_seed++ % hat->hat_num_hash;
573                         do {
574                                 higher = NULL;
575                                 HTABLE_ENTER(h);
576                                 for (ht = hat->hat_ht_hash[h]; ht;
577                                     ht = ht->ht_next) {
579                                         /*
580                                          * Can we rule out reaping?
581                                          */
582                                         if (ht->ht_busy != 0 ||
583                                             (ht->ht_flags & HTABLE_SHARED_PFN)||
584                                             ht->ht_level > 0 ||
585                                             ht->ht_valid_cnt > threshold ||
586                                             ht->ht_lock_cnt != 0)
587                                                 continue;
589                                         /*
```

```
590                              * Increment busy so the htable can't
591                              * disappear. We drop the htable mutex
592                              * to avoid deadlocks with
593                              * hat_pageunload() and the hment mutex
594                              * while we call hat_pte_unmap()
595                              */
596                             ++ht->ht_busy;
597                             HTABLE_EXIT(h);
598
599                             /*
600                              * Try stealing.
601                              * - unload and invalidate all PTEs
602                              */
603                             for (e = 0, va = ht->ht_vaddr;
604                                 e < HTABLE_NUM_PTES(ht) &&
605                                 ht->ht_valid_cnt > 0 &&
606                                 ht->ht_busy == 1 &&
607                                 ht->ht_lock_cnt == 0;
608                                 ++e, va += MMU_PAGESIZE) {
609                                     pte = x86pte_get(ht, e);
610                                     if (!PTE_ISVALID(pte))
611                                             continue;
612                                     hat_pte_unmap(ht, e,
613                                         HAT_UNLOAD, pte, NULL);
614                             }
615
616                             /*
617                              * Reacquire htable lock. If we didn't
618                              * remove all mappings in the table,
619                              * or another thread added a new mapping
620                              * behind us, give up on this table.
621                              */
622                             HTABLE_ENTER(h);
623                             if (ht->ht_busy != 1 ||
624                                 ht->ht_valid_cnt != 0 ||
625                                 ht->ht_lock_cnt != 0) {
626                                     --ht->ht_busy;
627                                     continue;
628                             }
629
630                             /*
631                              * Steal it and unlink the page table.
632                              */
633                             higher = ht->ht_parent;
634                             unlink_ptp(higher, ht, ht->ht_vaddr);
635
636                             /*
637                              * remove from the hash list
638                              */
639                             if (ht->ht_next)
640                                     ht->ht_next->ht_prev =
641                                         ht->ht_prev;
642
643                             if (ht->ht_prev) {
644                                     ht->ht_prev->ht_next =
645                                         ht->ht_next;
646                             } else {
647                                     ASSERT(hat->hat_ht_hash[h] ==
648                                         ht);
649                                     hat->hat_ht_hash[h] =
650                                         ht->ht_next;
651                             }
652
653                             /*
654                              * Break to outer loop to release the
655                              * higher (ht_parent) pagetable. This
```



```
656                              * spreads out the pain caused by
657                              * pagefaults.
658                              */
659                             ht->ht_next = list;
660                             list = ht;
661                             ++stolen;
662                             break;
663                     }
664                     HTABLE_EXIT(h);
665                     if (higher != NULL)
666                             htable_release(higher);
667                     if (++h == hat->hat_num_hash)
668                             h = 0;
669             } while (stolen < cnt && h != h_start);
670     }
671     }
672     atomic_dec_32(&htable_dont_cache);
672     atomic_add_32(&htable_dont_cache, -1);
673     return (list);
674 }
_____unchanged_portion_omitted_


971 /*
972  * This is called when a hat is being destroyed or swapped out. We reap all
973  * the remaining htables in the hat cache. If destroying all left over
974  * htables are also destroyed.
975  *
976  * We also don't need to invalidate any of the PTPs nor do any demapping.
977  */
978 void
979 htable_purge_hat(hat_t *hat)
980 {
981     htable_t *ht;
982     int h;
983
984     /*
985      * Purge the htable cache if just reaping.
986      */
987     if (!(hat->hat_flags & HAT_FREEING)) {
988             atomic_inc_32(&htable_dont_cache);
988             atomic_add_32(&htable_dont_cache, 1);
989             for (;;) {
990                     hat_enter(hat);
991                     ht = hat->hat_ht_cached;
992                     if (ht == NULL) {
993                             hat_exit(hat);
994                             break;
995                     }
996                     hat->hat_ht_cached = ht->ht_next;
997                     hat_exit(hat);
998                     htable_free(ht);
999             }
1000            atomic_dec_32(&htable_dont_cache);
1000            atomic_add_32(&htable_dont_cache, -1);
1001            return;
1002    }
1003
1004    /*
1005     * if freeing, no locking is needed
1006     */
1007    while ((ht = hat->hat_ht_cached) != NULL) {
1008            hat->hat_ht_cached = ht->ht_next;
1009            htable_free(ht);
1010    }
```

```
1012         /*
1013          * walk thru the htable hash table and free all the htables in it.
1014          */
1015         for (h = 0; h < hat->hat_num_hash; ++h) {
1016                 while ((ht = hat->hat_ht_hash[h]) != NULL) {
1017                         if (ht->ht_next)
1018                                 ht->ht_next->ht_prev = ht->ht_prev;

1020                         if (ht->ht_prev) {
1021                                 ht->ht_prev->ht_next = ht->ht_next;
1022                         } else {
1023                                 ASSERT(hat->hat_ht_hash[h] == ht);
1024                                 hat->hat_ht_hash[h] = ht->ht_next;
1025                         }
1026                         htable_free(ht);
1027                 }
1028         }
1029 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   10982 Mon Jul 28 07:44:59 2014
new/usr/src/uts/i86pc/vm/htable.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
  23  * Use is subject to license terms.
  24  */

  26 #ifndef _VM_HTABLE_H
  27 #define _VM_HTABLE_H

  29 #pragma ident   "%Z%%M% %I%     %E% SMI"

  29 #ifdef __cplusplus
  30 extern "C" {
  31 #endif

  33 #if defined(__GNUC__) && defined(_ASM_INLINES) && defined(_KERNEL)
  34 #include <asm/htable.h>
  35 #endif

  37 extern void atomic_andb(uint8_t *addr, uint8_t value);
  38 extern void atomic_orb(uint8_t *addr, uint8_t value);
  39 extern void atomic_inc16(uint16_t *addr);
  40 extern void atomic_dec16(uint16_t *addr);
  41 extern void mmu_tlbflush_entry(caddr_t addr);

  43 /*
  44  * Each hardware page table has an htable_t describing it.
  45  *
  46  * We use a reference counter mechanism to detect when we can free an htable.
  47  * In the implmentation the reference count is split into 2 separate counters:
  48  *
  49  *      ht_busy is a traditional reference count of uses of the htable pointer
  50  *
  51  *      ht_valid_cnt is a count of how references are implied by valid PTE/PTP
  52  *              entries in the pagetable
  53  *
  54  * ht_busy is only incremented by htable_lookup() or htable_create()
  55  * while holding the appropriate hash_table mutex. While installing a new
  56  * valid PTE or PTP, in order to increment ht_valid_cnt a thread must have
  57  * done an htable_lookup() or htable_create() but not the htable_release yet.
  58  *
  59  * htable_release(), while holding the mutex, can know that if
```



```
  60  * busy == 1 and valid_cnt == 0, the htable can be free'd.
  61  *
  62  * The fields have been ordered to make htable_lookup() fast. Hence,
  63  * ht_hat, ht_vaddr, ht_level and ht_next need to be clustered together.
  64  */
  65 struct htable {
  66         struct htable   *ht_next;       /* forward link for hash table */
  67         struct hat      *ht_hat;        /* hat this mapping comes from */
  68         uintptr_t       ht_vaddr;       /* virt addr at start of this table */
  69         int8_t          ht_level;       /* page table level: 0=4K, 1=2M, ... */
  70         uint8_t         ht_flags;       /* see below */
  71         int16_t         ht_busy;        /* implements locking protocol */
  72         int16_t         ht_valid_cnt;   /* # of valid entries in this table */
  73         uint32_t        ht_lock_cnt;    /* # of locked entries in this table */
  74                                         /* never used for kernel hat */
  75         pfn_t           ht_pfn;         /* pfn of page of the pagetable */
  76         struct htable   *ht_prev;       /* backward link for hash table */
  77         struct htable   *ht_parent;     /* htable that points to this htable */
  78         struct htable   *ht_shares;     /* for HTABLE_SHARED_PFN only */
  79 };
______unchanged_portion_omitted_


 118 /*
 119  * Compute the last page aligned VA mapped by an htable.
 120  *
 121  * Given a va and a level, compute the virtual address of the start of the
 122  * next page at that level.
 123  *
 124  * XX64 - The check for the VA hole needs to be better generalized.
 125  */
 126 #if defined(__amd64)
 127 #define HTABLE_NUM_PTES(ht)     (((ht)->ht_flags & HTABLE_VLP) ? 4 : 512)

 129 #define HTABLE_LAST_PAGE(ht)                                            \
 130         ((ht)->ht_level == mmu.max_level ? ((uintptr_t)0UL - MMU_PAGESIZE) :\
 131         ((ht)->ht_vaddr - MMU_PAGESIZE +                                \
 132         ((uintptr_t)HTABLE_NUM_PTES(ht) << LEVEL_SHIFT((ht)->ht_level))))

 134 #define NEXT_ENTRY_VA(va, l)    \
 135         ((va & LEVEL_MASK(l)) + LEVEL_SIZE(l) == mmu.hole_start ?       \
 136         mmu.hole_end : (va & LEVEL_MASK(l)) + LEVEL_SIZE(l))

 138 #elif defined(__i386)

 140 #define HTABLE_NUM_PTES(ht)     \
 141         (!mmu.pae_hat ? 1024 : ((ht)->ht_level == 2 ? 4 : 512))

 143 #define HTABLE_LAST_PAGE(ht)    ((ht)->ht_vaddr - MMU_PAGESIZE + \
 144         ((uintptr_t)HTABLE_NUM_PTES(ht) << LEVEL_SHIFT((ht)->ht_level)))

 146 #define NEXT_ENTRY_VA(va, l) ((va & LEVEL_MASK(l)) + LEVEL_SIZE(l))

 148 #endif

 150 #if defined(_KERNEL)

 152 /*
 153  * initialization function called from hat_init()
 154  */
 155 extern void htable_init(void);

 157 /*
 158  * Functions to lookup, or "lookup and create", the htable corresponding
 159  * to the virtual address "vaddr"  in the "hat" at the given "level" of
 160  * page tables. htable_lookup() may return NULL if no such entry exists.
```

```
 *
 * On return the given htable is marked busy (a shared lock) - this prevents
 * the htable from being stolen or freed) until htable_release() is called.
 *
 * If kalloc_flag is set on an htable_create() we can't call kmem allocation
 * routines for this htable, since it's for the kernel hat itself.
 *
 * htable_acquire() is used when an htable pointer has been extracted from
 * an hment and we need to get a reference to the htable.
 */
extern htable_t *htable_lookup(struct hat *hat, uintptr_t vaddr, level_t level);
extern htable_t *htable_create(struct hat *hat, uintptr_t vaddr, level_t level,
        htable_t *shared);
extern void htable_acquire(htable_t *);

extern void htable_release(htable_t *ht);
extern void htable_destroy(htable_t *ht);

/*
 * Code to free all remaining htables for a hat. Called after the hat is no
 * longer in use by any thread.
 */
extern void htable_purge_hat(struct hat *hat);

/*
 * Find the htable, page table entry index, and PTE of the given virtual
 * address.  If not found returns NULL. When found, returns the htable_t *,
 * sets entry, and has a hold on the htable.
 */
extern htable_t *htable_getpte(struct hat *, uintptr_t, uint_t *, x86pte_t *,
        level_t);

/*
 * Similar to hat_getpte(), except that this only succeeds if a valid
 * page mapping is present.
 */
extern htable_t *htable_getpage(struct hat *hat, uintptr_t va, uint_t *entry);

/*
 * Called to allocate initial/additional htables for reserve.
 */
extern void htable_initial_reserve(uint_t);
extern void htable_reserve(uint_t);

/*
 * Used to readjust the htable reserve after the reserve list has been used.
 * Also called after boot to release left over boot reserves.
 */
extern void htable_adjust_reserve(void);

/*
 * return number of bytes mapped by all the htables in a given hat
 */
extern size_t htable_mapped(struct hat *);


/*
 * Attach initial pagetables as htables
 */
extern void htable_attach(struct hat *, uintptr_t, level_t, struct htable *,
    pfn_t);

/*
 * Routine to find the next populated htable at or above a given virtual
 * address. Can specify an upper limit, or HTABLE_WALK_TO_END to indicate
 * that it should search the entire address space.  Similar to
```



```
 * hat_getpte(), but used for walking through address ranges. It can be
 * used like this:
 *
 *      va = ...
 *      ht = NULL;
 *      while (va < end_va) {
 *              pte = htable_walk(hat, &ht, &va, end_va);
 *              if (!pte)
 *                      break;
 *
 *              ... code to operate on page at va ...
 *
 *              va += LEVEL_SIZE(ht->ht_level);
 *      }
 *      if (ht)
 *              htable_release(ht);
 *
 */
extern x86pte_t htable_walk(struct hat *hat, htable_t **ht, uintptr_t *va,
        uintptr_t eaddr);

#define HTABLE_WALK_TO_END ((uintptr_t)-1)

/*
 * Utilities convert between virtual addresses and page table entry indeces.
 */
extern uint_t htable_va2entry(uintptr_t va, htable_t *ht);
extern uintptr_t htable_e2va(htable_t *ht, uint_t entry);

/*
 * Interfaces that provide access to page table entries via the htable.
 *
 * Note that all accesses except x86pte_copy() and x86pte_zero() are atomic.
 */
extern void     x86pte_cpu_init(cpu_t *);
extern void     x86pte_cpu_fini(cpu_t *);

extern x86pte_t x86pte_get(htable_t *, uint_t entry);

/*
 * x86pte_set returns LPAGE_ERROR if it's asked to overwrite a page table
 * link with a large page mapping.
 */
#define LPAGE_ERROR (-(x86pte_t)1)
extern x86pte_t x86pte_set(htable_t *, uint_t entry, x86pte_t new, void *);

extern x86pte_t x86pte_inval(htable_t *ht, uint_t entry,
        x86pte_t old, x86pte_t *ptr);

extern x86pte_t x86pte_update(htable_t *ht, uint_t entry,
        x86pte_t old, x86pte_t new);

extern void     x86pte_copy(htable_t *src, htable_t *dest, uint_t entry,
        uint_t cnt);

/*
 * access to a pagetable knowing only the pfn
 */
extern x86pte_t *x86pte_mapin(pfn_t, uint_t, htable_t *);
extern void x86pte_mapout(void);

/*
 * these are actually inlines for "lock; incw", "lock; decw", etc. instructions.
 */
#define HTABLE_INC(x)   atomic_inc16((uint16_t *)&x)
#define HTABLE_DEC(x)   atomic_dec16((uint16_t *)&x)
```

```
293 #define HTABLE_LOCK_INC(ht)     atomic_inc_32(&(ht)->ht_lock_cnt)
294 #define HTABLE_LOCK_DEC(ht)     atomic_dec_32(&(ht)->ht_lock_cnt)
295 #define HTABLE_LOCK_INC(ht)     atomic_add_32(&(ht)->ht_lock_cnt, 1)
296 #define HTABLE_LOCK_DEC(ht)     atomic_add_32(&(ht)->ht_lock_cnt, -1)

296 #ifdef __xpv
297 extern void xen_flush_va(caddr_t va);
298 extern void xen_gflush_va(caddr_t va, cpuset_t);
299 extern void xen_flush_tlb(void);
300 extern void xen_gflush_tlb(cpuset_t);
301 extern void xen_pin(pfn_t, level_t);
302 extern void xen_unpin(pfn_t);
303 extern int xen_kpm_page(pfn_t, uint_t);

305 /*
306  * The hypervisor maps all page tables into our address space read-only.
307  * Under normal circumstances, the hypervisor then handles all updates to
308  * the page tables underneath the covers for us.  However, when we are
309  * trying to dump core after a hypervisor panic, the hypervisor is no
310  * longer available to do these updates.  To work around the protection
311  * problem, we simply disable write-protect checking for the duration of a
312  * pagetable update operation.
313  */
314 #define XPV_ALLOW_PAGETABLE_UPDATES()                                   \
315         {                                                               \
316                 if (IN_XPV_PANIC())                                     \
317                         setcr0((getcr0() & ~CR0_WP) & 0xffffffff);      \
318         }
319 #define XPV_DISALLOW_PAGETABLE_UPDATES()                                \
320         {                                                               \
321                 if (IN_XPV_PANIC() > 0)                                 \
322                         setcr0((getcr0() | CR0_WP) & 0xffffffff);       \
323         }

325 #else /* __xpv */

327 #define XPV_ALLOW_PAGETABLE_UPDATES()
328 #define XPV_DISALLOW_PAGETABLE_UPDATES()

330 #endif

332 #endif  /* _KERNEL */


335 #ifdef  __cplusplus
336 }
```

*unchanged_portion_omitted_*

****
24432 Mon Jul 28 07:44:59 2014
new/usr/src/uts/i86xpv/io/psm/xpv_uppc.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
246 /*ARGSUSED*/
247 static int
248 xen_uppc_addspl(int irqno, int ipl, int min_ipl, int max_ipl)
249 {
250         int ret = PSM_SUCCESS;
251         cpuset_t cpus;

253         if (irqno >= 0 && irqno <= MAX_ISA_IRQ)
254                 atomic_inc_16(&xen_uppc_irq_shared_table[irqno]);
254                 atomic_add_16(&xen_uppc_irq_shared_table[irqno], 1);

256         /*
257          * We are called at splhi() so we can't call anything that might end
258          * up trying to context switch.
259          */
260         if (irqno >= PIRQ_BASE && irqno < NR_PIRQS &&
261             DOMAIN_IS_INITDOMAIN(xen_info)) {
262                 CPUSET_ZERO(cpus);
263                 CPUSET_ADD(cpus, 0);
264                 ec_setup_pirq(irqno, ipl, &cpus);
265         } else {
266                 /*
267                  * Set priority/affinity/enable for non PIRQs
268                  */
269                 ret = ec_set_irq_priority(irqno, ipl);
270                 ASSERT(ret == 0);
271                 CPUSET_ZERO(cpus);
272                 CPUSET_ADD(cpus, 0);
273                 ec_set_irq_affinity(irqno, cpus);
274                 ec_enable_irq(irqno);
275         }

277         return (ret);
278 }

280 /*ARGSUSED*/
281 static int
282 xen_uppc_delspl(int irqno, int ipl, int min_ipl, int max_ipl)
283 {
284         int err = PSM_SUCCESS;

286         if (irqno >= 0 && irqno <= MAX_ISA_IRQ)
287                 atomic_dec_16(&xen_uppc_irq_shared_table[irqno]);
287                 atomic_add_16(&xen_uppc_irq_shared_table[irqno], -1);

289         if (irqno >= PIRQ_BASE && irqno < NR_PIRQS &&
290             DOMAIN_IS_INITDOMAIN(xen_info)) {
291                 if (max_ipl == PSM_INVALID_IPL) {
292                         /*
293                          * unbind if no more sharers of this irq/evtchn
294                          */
295                         (void) ec_block_irq(irqno);
296                         ec_unbind_irq(irqno);
297                 } else {
298                         /*
299                          * If still in use reset priority
300                          */
301                         err = ec_set_irq_priority(irqno, max_ipl);
```



```
302                 }
303         } else {
304                 (void) ec_block_irq(irqno);
305                 ec_unbind_irq(irqno);
306         }
307         return (err);
308 }
```
_____***unchanged_portion_omitted_***

```
**********************************************************
  164146 Mon Jul 28 07:44:59 2014
new/usr/src/uts/intel/io/scsi/adapters/arcmsr/arcmsr.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1820 /*
1821  * arcmsr_post_ccb - Send a protocol specific ARC send postcard to a AIOC.
1822  *
1823  * handle:              Handle of registered ARC protocol driver
1824  * adapter_id:          AIOC unique identifier(integer)
1825  * pPOSTCARD_SEND:      Pointer to ARC send postcard
1826  *
1827  * This routine posts a ARC send postcard to the request post FIFO of a
1828  * specific ARC adapter.
1829  */
1830 static int
1831 arcmsr_post_ccb(struct ACB *acb, struct CCB *ccb)
1832 {
1833         uint32_t cdb_phyaddr_pattern = ccb->cdb_phyaddr_pattern;
1834         struct scsi_pkt *pkt = ccb->pkt;
1835         struct ARCMSR_CDB *arcmsr_cdb;
1836         uint_t pkt_flags = pkt->pkt_flags;

1838         arcmsr_cdb = &ccb->arcmsr_cdb;

1840         /* TODO: Use correct offset and size for syncing? */
1841         if (ddi_dma_sync(acb->ccbs_pool_handle, 0, 0, DDI_DMA_SYNC_FORDEV) ==
1842             DDI_FAILURE)
1843                 return (DDI_FAILURE);

1845         atomic_inc_32(&acb->ccboutstandingcount);
1845         atomic_add_32(&acb->ccboutstandingcount, 1);
1846         ccb->ccb_time = (time_t)(ddi_get_time() + pkt->pkt_time);

1848         ccb->ccb_state = ARCMSR_CCB_START;
1849         switch (acb->adapter_type) {
1850         case ACB_ADAPTER_TYPE_A:
1851         {
1852                 struct HBA_msgUnit *phbamu;

1854                 phbamu = (struct HBA_msgUnit *)acb->pmu;
1855                 if (arcmsr_cdb->Flags & ARCMSR_CDB_FLAG_SGL_BSIZE) {
1856                         CHIP_REG_WRITE32(acb->reg_mu_acc_handle0,
1857                             &phbamu->inbound_queueport,
1858                             cdb_phyaddr_pattern |
1859                             ARCMSR_CCBPOST_FLAG_SGL_BSIZE);
1860                 } else {
1861                         CHIP_REG_WRITE32(acb->reg_mu_acc_handle0,
1862                             &phbamu->inbound_queueport, cdb_phyaddr_pattern);
1863                 }
1864                 if (pkt_flags & FLAG_NOINTR)
1865                         arcmsr_polling_hba_ccbdone(acb, ccb);
1866                 break;
1867         }

1869         case ACB_ADAPTER_TYPE_B:
1870         {
1871                 struct HBB_msgUnit *phbbmu;
1872                 int ending_index, index;

1874                 phbbmu = (struct HBB_msgUnit *)acb->pmu;
1875                 index = phbbmu->postq_index;
1876                 ending_index = ((index+1)%ARCMSR_MAX_HBB_POSTQUEUE);
1877                 phbbmu->post_qbuffer[ending_index] = 0;
```



```
1878                 if (arcmsr_cdb->Flags & ARCMSR_CDB_FLAG_SGL_BSIZE) {
1879                         phbbmu->post_qbuffer[index] =
1880                             (cdb_phyaddr_pattern|ARCMSR_CCBPOST_FLAG_SGL_BSIZE);
1881                 } else {
1882                         phbbmu->post_qbuffer[index] = cdb_phyaddr_pattern;
1883                 }
1884                 index++;
1885                 /* if last index number set it to 0 */
1886                 index %= ARCMSR_MAX_HBB_POSTQUEUE;
1887                 phbbmu->postq_index = index;
1888                 CHIP_REG_WRITE32(acb->reg_mu_acc_handle0,
1889                     &phbbmu->hbb_doorbell->drv2iop_doorbell,
1890                     ARCMSR_DRV2IOP_CDB_POSTED);

1892                 if (pkt_flags & FLAG_NOINTR)
1893                         arcmsr_polling_hbb_ccbdone(acb, ccb);
1894                 break;
1895         }

1897         case ACB_ADAPTER_TYPE_C:
1898         {
1899                 struct HBC_msgUnit *phbcmu;
1900                 uint32_t ccb_post_stamp, arc_cdb_size;

1902                 phbcmu = (struct HBC_msgUnit *)acb->pmu;
1903                 arc_cdb_size = (ccb->arc_cdb_size > 0x300) ? 0x300 :
1904                     ccb->arc_cdb_size;
1905                 ccb_post_stamp = (cdb_phyaddr_pattern |
1906                     ((arc_cdb_size-1) >> 6) |1);
1907                 if (acb->cdb_phyaddr_hi32) {
1908                         CHIP_REG_WRITE32(acb->reg_mu_acc_handle0,
1909                             &phbcmu->inbound_queueport_high,
1910                             acb->cdb_phyaddr_hi32);
1911                         CHIP_REG_WRITE32(acb->reg_mu_acc_handle0,
1912                             &phbcmu->inbound_queueport_low, ccb_post_stamp);
1913                 } else {
1914                         CHIP_REG_WRITE32(acb->reg_mu_acc_handle0,
1915                             &phbcmu->inbound_queueport_low, ccb_post_stamp);
1916                 }
1917                 if (pkt_flags & FLAG_NOINTR)
1918                         arcmsr_polling_hbc_ccbdone(acb, ccb);
1919                 break;
1920         }

1922         }
1923         return (DDI_SUCCESS);
1924 }


1927 static void
1928 arcmsr_ccb_complete(struct CCB *ccb, int flag)
1929 {
1930         struct ACB *acb = ccb->acb;
1931         struct scsi_pkt *pkt = ccb->pkt;

1933         if (pkt == NULL) {
1934                 return;
1935         }
1936         ccb->ccb_state |= ARCMSR_CCB_DONE;
1937         pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET |
1938             STATE_SENT_CMD | STATE_GOT_STATUS);

1940         if ((ccb->ccb_flags & CCB_FLAG_DMACONSISTENT) &&
1941             (pkt->pkt_state & STATE_XFERRED_DATA)) {
1942                 (void) ddi_dma_sync(ccb->pkt_dma_handle, 0, 0,
1943                     DDI_DMA_SYNC_FORCPU);
```

```
1944         }
1945         /*
1946          * TODO: This represents a potential race condition, and is
1947          * ultimately a poor design decision.  Revisit this code
1948          * and solve the mutex ownership issue correctly.
1949          */
1950         if (mutex_owned(&acb->isr_mutex)) {
1951                 mutex_exit(&acb->isr_mutex);
1952                 scsi_hba_pkt_comp(pkt);
1953                 mutex_enter(&acb->isr_mutex);
1954         } else {
1955                 scsi_hba_pkt_comp(pkt);
1956         }
1957         if (flag == 1) {
1958                 atomic_dec_32(&acb->ccboutstandingcount);
1958                 atomic_add_32(&acb->ccboutstandingcount, -1);
1959         }
1960 }
```

*unchanged_portion_omitted_*

```
**********************************************************
  422677 Mon Jul 28 07:45:00 2014
new/usr/src/uts/sfmmu/vm/hat_sfmmu.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

2125 /*
2126  * Duplicate the translations of an as into another newas
2127  */
2128 /* ARGSUSED */
2129 int
2130 hat_dup(struct hat *hat, struct hat *newhat, caddr_t addr, size_t len,
2131         uint_t flag)
2132 {
2133         sf_srd_t *srdp;
2134         sf_scd_t *scdp;
2135         int i;
2136         extern uint_t get_color_start(struct as *);

2138         ASSERT(hat->sfmmu_xhat_provider == NULL);
2139         ASSERT((flag == 0) || (flag == HAT_DUP_ALL) || (flag == HAT_DUP_COW) ||
2140             (flag == HAT_DUP_SRD));
2141         ASSERT(hat != ksfmmup);
2142         ASSERT(newhat != ksfmmup);
2143         ASSERT(flag != HAT_DUP_ALL || hat->sfmmu_srdp == newhat->sfmmu_srdp);

2145         if (flag == HAT_DUP_COW) {
2146                 panic("hat_dup: HAT_DUP_COW not supported");
2147         }

2149         if (flag == HAT_DUP_SRD && ((srdp = hat->sfmmu_srdp) != NULL)) {
2150                 ASSERT(srdp->srd_evp != NULL);
2151                 VN_HOLD(srdp->srd_evp);
2152                 ASSERT(srdp->srd_refcnt > 0);
2153                 newhat->sfmmu_srdp = srdp;
2154                 atomic_inc_32((volatile uint_t *)&srdp->srd_refcnt);
2154                 atomic_add_32((volatile uint_t *)&srdp->srd_refcnt, 1);
2155         }

2157         /*
2158          * HAT_DUP_ALL flag is used after as duplication is done.
2159          */
2160         if (flag == HAT_DUP_ALL && ((srdp = newhat->sfmmu_srdp) != NULL)) {
2161                 ASSERT(newhat->sfmmu_srdp->srd_refcnt >= 2);
2162                 newhat->sfmmu_rtteflags = hat->sfmmu_rtteflags;
2163                 if (hat->sfmmu_flags & HAT_4MTEXT_FLAG) {
2164                         newhat->sfmmu_flags |= HAT_4MTEXT_FLAG;
2165                 }

2167                 /* check if need to join scd */
2168                 if ((scdp = hat->sfmmu_scdp) != NULL &&
2169                     newhat->sfmmu_scdp != scdp) {
2170                         int ret;
2171                         SF_RGNMAP_IS_SUBSET(&newhat->sfmmu_region_map,
2172                             &scdp->scd_region_map, ret);
2173                         ASSERT(ret);
2174                         sfmmu_join_scd(scdp, newhat);
2175                         ASSERT(newhat->sfmmu_scdp == scdp &&
2176                             scdp->scd_refcnt >= 2);
2177                         for (i = 0; i < max_mmu_page_sizes; i++) {
2178                                 newhat->sfmmu_ismttecnt[i] =
2179                                     hat->sfmmu_ismttecnt[i];
2180                                 newhat->sfmmu_scdismttecnt[i] =
2181                                     hat->sfmmu_scdismttecnt[i];
2182                         }
```



```
2183                 }

2185                 sfmmu_check_page_sizes(newhat, 1);
2186         }

2188         if (flag == HAT_DUP_ALL && consistent_coloring == 0 &&
2189             update_proc_pgcolorbase_after_fork != 0) {
2190                 hat->sfmmu_clrbin = get_color_start(hat->sfmmu_as);
2191         }
2192         return (0);
2193 }
______unchanged_portion_omitted_

3065 /*
3066  * Function adds a tte entry into the hmeblk. It returns 0 if successful and 1
3067  * otherwise.
3068  */
3069 static int
3070 sfmmu_tteload_addentry(sfmmu_t *sfmmup, struct hme_blk *hmeblkp, tte_t *ttep,
3071         caddr_t vaddr, page_t **pps, uint_t flags, uint_t rid)
3072 {
3073         page_t *pp = *pps;
3074         int hmenum, size, remap;
3075         tte_t tteold, flush_tte;
3076 #ifdef DEBUG
3077         tte_t orig_old;
3078 #endif /* DEBUG */
3079         struct sf_hment *sfhme;
3080         kmutex_t *pml, *pmtx;
3081         hatlock_t *hatlockp;
3082         int myflt;

3084         /*
3085          * remove this panic when we decide to let user virtual address
3086          * space be >= USERLIMIT.
3087          */
3088         if (!TTE_IS_PRIVILEGED(ttep) && vaddr >= (caddr_t)USERLIMIT)
3089                 panic("user addr %p in kernel space", (void *)vaddr);
3090 #if defined(TTE_IS_GLOBAL)
3091         if (TTE_IS_GLOBAL(ttep))
3092                 panic("sfmmu_tteload: creating global tte");
3093 #endif

3095 #ifdef DEBUG
3096         if (pf_is_memory(sfmmu_ttetopfn(ttep, vaddr)) &&
3097             !TTE_IS_PCACHEABLE(ttep) && !sfmmu_allow_nc_trans)
3098                 panic("sfmmu_tteload: non cacheable memory tte");
3099 #endif /* DEBUG */

3101         /* don't simulate dirty bit for writeable ISM/DISM mappings */
3102         if ((flags & HAT_LOAD_SHARE) && TTE_IS_WRITABLE(ttep)) {
3103                 TTE_SET_REF(ttep);
3104                 TTE_SET_MOD(ttep);
3105         }

3107         if ((flags & HAT_LOAD_SHARE) || !TTE_IS_REF(ttep) ||
3108             !TTE_IS_MOD(ttep)) {
3109                 /*
3110                  * Don't load TSB for dummy as in ISM.  Also don't preload
3111                  * the TSB if the TTE isn't writable since we're likely to
3112                  * fault on it again -- preloading can be fairly expensive.
3113                  */
3114                 flags |= SFMMU_NO_TSBLOAD;
3115         }

3117         size = TTE_CSZ(ttep);
```

```
3118         switch (size) {
3119         case TTE8K:
3120                 SFMMU_STAT(sf_tteload8k);
3121                 break;
3122         case TTE64K:
3123                 SFMMU_STAT(sf_tteload64k);
3124                 break;
3125         case TTE512K:
3126                 SFMMU_STAT(sf_tteload512k);
3127                 break;
3128         case TTE4M:
3129                 SFMMU_STAT(sf_tteload4m);
3130                 break;
3131         case (TTE32M):
3132                 SFMMU_STAT(sf_tteload32m);
3133                 ASSERT(mmu_page_sizes == max_mmu_page_sizes);
3134                 break;
3135         case (TTE256M):
3136                 SFMMU_STAT(sf_tteload256m);
3137                 ASSERT(mmu_page_sizes == max_mmu_page_sizes);
3138                 break;
3139         }

3141         ASSERT(!((uintptr_t)vaddr & TTE_PAGE_OFFSET(size)));
3142         SFMMU_VALIDATE_HMERID(sfmmup, rid, vaddr, TTEBYTES(size));
3143         ASSERT(!SFMMU_IS_SHMERID_VALID(rid) || hmeblkp->hblk_shared);
3144         ASSERT(SFMMU_IS_SHMERID_VALID(rid) || !hmeblkp->hblk_shared);

3146         HBLKTOHME_IDX(sfhme, hmeblkp, vaddr, hmenum);

3148         /*
3149          * Need to grab mlist lock here so that pageunload
3150          * will not change tte behind us.
3151          */
3152         if (pp) {
3153                 pml = sfmmu_mlist_enter(pp);
3154         }

3156         sfmmu_copytte(&sfhme->hme_tte, &tteold);
3157         /*
3158          * Look for corresponding hment and if valid verify
3159          * pfns are equal.
3160          */
3161         remap = TTE_IS_VALID(&tteold);
3162         if (remap) {
3163                 pfn_t   new_pfn, old_pfn;

3165                 old_pfn = TTE_TO_PFN(vaddr, &tteold);
3166                 new_pfn = TTE_TO_PFN(vaddr, ttep);

3168                 if (flags & HAT_LOAD_REMAP) {
3169                         /* make sure we are remapping same type of pages */
3170                         if (pf_is_memory(old_pfn) != pf_is_memory(new_pfn)) {
3171                                 panic("sfmmu_tteload - tte remap io<->memory");
3172                         }
3173                         if (old_pfn != new_pfn &&
3174                             (pp != NULL || sfhme->hme_page != NULL)) {
3175                                 panic("sfmmu_tteload - tte remap pp != NULL");
3176                         }
3177                 } else if (old_pfn != new_pfn) {
3178                         panic("sfmmu_tteload - tte remap, hmeblkp 0x%p",
3179                             (void *)hmeblkp);
3180                 }
3181                 ASSERT(TTE_CSZ(&tteold) == TTE_CSZ(ttep));
3182         }
```



```
3184         if (pp) {
3185                 if (size == TTE8K) {
3186 #ifdef VAC
3187                         /*
3188                          * Handle VAC consistency
3189                          */
3190                         if (!remap && (cache & CACHE_VAC) && !PP_ISNC(pp)) {
3191                                 sfmmu_vac_conflict(sfmmup, vaddr, pp);
3192                         }
3193 #endif

3195                         if (TTE_IS_WRITABLE(ttep) && PP_ISRO(pp)) {
3196                                 pmtx = sfmmu_page_enter(pp);
3197                                 PP_CLRRO(pp);
3198                                 sfmmu_page_exit(pmtx);
3199                         } else if (!PP_ISMAPPED(pp) &&
3200                             (!TTE_IS_WRITABLE(ttep)) && !(PP_ISMOD(pp))) {
3201                                 pmtx = sfmmu_page_enter(pp);
3202                                 if (!(PP_ISMOD(pp))) {
3203                                         PP_SETRO(pp);
3204                                 }
3205                                 sfmmu_page_exit(pmtx);
3206                         }

3208                 } else if (sfmmu_pagearray_setup(vaddr, pps, ttep, remap)) {
3209                         /*
3210                          * sfmmu_pagearray_setup failed so return
3211                          */
3212                         sfmmu_mlist_exit(pml);
3213                         return (1);
3214                 }
3215         }

3217         /*
3218          * Make sure hment is not on a mapping list.
3219          */
3220         ASSERT(remap || (sfhme->hme_page == NULL));

3222         /* if it is not a remap then hme->next better be NULL */
3223         ASSERT((!remap) ? sfhme->hme_next == NULL : 1);

3225         if (flags & HAT_LOAD_LOCK) {
3226                 if ((hmeblkp->hblk_lckcnt + 1) >= MAX_HBLK_LCKCNT) {
3227                         panic("too high lckcnt-hmeblk %p",
3228                             (void *)hmeblkp);
3229                 }
3230                 atomic_inc_32(&hmeblkp->hblk_lckcnt);
3230                 atomic_add_32(&hmeblkp->hblk_lckcnt, 1);

3232                 HBLK_STACK_TRACE(hmeblkp, HBLK_LOCK);
3233         }

3235 #ifdef VAC
3236         if (pp && PP_ISNC(pp)) {
3237                 /*
3238                  * If the physical page is marked to be uncacheable, like
3239                  * by a vac conflict, make sure the new mapping is also
3240                  * uncacheable.
3241                  */
3242                 TTE_CLR_VCACHEABLE(ttep);
3243                 ASSERT(PP_GET_VCOLOR(pp) == NO_VCOLOR);
3244         }
3245 #endif
3246         ttep->tte_hmenum = hmenum;

3248 #ifdef DEBUG
```

```
3249         orig_old = tteold;
3250 #endif /* DEBUG */
3251
3252         while (sfmmu_modifytte_try(&tteold, ttep, &sfhme->hme_tte) < 0) {
3253                 if ((sfmmup == KHATID) &&
3254                     (flags & (HAT_LOAD_LOCK | HAT_LOAD_REMAP))) {
3255                         sfmmu_copytte(&sfhme->hme_tte, &tteold);
3256                 }
3257 #ifdef DEBUG
3258                 chk_tte(&orig_old, &tteold, ttep, hmeblkp);
3259 #endif /* DEBUG */
3260         }
3261         ASSERT(TTE_IS_VALID(&sfhme->hme_tte));
3262
3263         if (!TTE_IS_VALID(&tteold)) {
3264
3265                 atomic_inc_16(&hmeblkp->hblk_vcnt);
3265                 atomic_add_16(&hmeblkp->hblk_vcnt, 1);
3266                 if (rid == SFMMU_INVALID_SHMERID) {
3267                         atomic_inc_ulong(&sfmmup->sfmmu_ttecnt[size]);
3267                         atomic_add_long(&sfmmup->sfmmu_ttecnt[size], 1);
3268                 } else {
3269                         sf_srd_t *srdp = sfmmup->sfmmu_srdp;
3270                         sf_region_t *rgnp = srdp->srd_hmergnp[rid];
3271                         /*
3272                          * We already accounted for region ttecnt's in sfmmu
3273                          * during hat_join_region() processing. Here we
3274                          * only update ttecnt's in region struture.
3275                          */
3276                         atomic_inc_ulong(&rgnp->rgn_ttecnt[size]);
3276                         atomic_add_long(&rgnp->rgn_ttecnt[size], 1);
3277                 }
3278         }
3279
3280         myflt = (astosfmmu(curthread->t_procp->p_as) == sfmmup);
3281         if (size > TTE8K && (flags & HAT_LOAD_SHARE) == 0 &&
3282             sfmmup != ksfmmup) {
3283                 uchar_t tteflag = 1 << size;
3284                 if (rid == SFMMU_INVALID_SHMERID) {
3285                         if (!(sfmmup->sfmmu_tteflags & tteflag)) {
3286                                 hatlockp = sfmmu_hat_enter(sfmmup);
3287                                 sfmmup->sfmmu_tteflags |= tteflag;
3288                                 sfmmu_hat_exit(hatlockp);
3289                         }
3290                 } else if (!(sfmmup->sfmmu_rtteflags & tteflag)) {
3291                         hatlockp = sfmmu_hat_enter(sfmmup);
3292                         sfmmup->sfmmu_rtteflags |= tteflag;
3293                         sfmmu_hat_exit(hatlockp);
3294                 }
3295                 /*
3296                  * Update the current CPU tsbmiss area, so the current thread
3297                  * won't need to take the tsbmiss for the new pagesize.
3298                  * The other threads in the process will update their tsb
3299                  * miss area lazily in sfmmu_tsbmiss_exception() when they
3300                  * fail to find the translation for a newly added pagesize.
3301                  */
3302                 if (size > TTE64K && myflt) {
3303                         struct tsbmiss *tsbmp;
3304                         kpreempt_disable();
3305                         tsbmp = &tsbmiss_area[CPU->cpu_id];
3306                         if (rid == SFMMU_INVALID_SHMERID) {
3307                                 if (!(tsbmp->uhat_tteflags & tteflag)) {
3308                                         tsbmp->uhat_tteflags |= tteflag;
3309                                 }
3310                         } else {
3311                                 if (!(tsbmp->uhat_rtteflags & tteflag)) {
```



```
3312                                         tsbmp->uhat_rtteflags |= tteflag;
3313                                 }
3314                         }
3315                         kpreempt_enable();
3316                 }
3317         }
3318
3319         if (size >= TTE4M && (flags & HAT_LOAD_TEXT) &&
3320             !SFMMU_FLAGS_ISSET(sfmmup, HAT_4MTEXT_FLAG)) {
3321                 hatlockp = sfmmu_hat_enter(sfmmup);
3322                 SFMMU_FLAGS_SET(sfmmup, HAT_4MTEXT_FLAG);
3323                 sfmmu_hat_exit(hatlockp);
3324         }
3325
3326         flush_tte.tte_intlo = (tteold.tte_intlo ^ ttep->tte_intlo) &
3327             hw_tte.tte_intlo;
3328         flush_tte.tte_inthi = (tteold.tte_inthi ^ ttep->tte_inthi) &
3329             hw_tte.tte_inthi;
3330
3331         if (remap && (flush_tte.tte_inthi || flush_tte.tte_intlo)) {
3332                 /*
3333                  * If remap and new tte differs from old tte we need
3334                  * to sync the mod bit and flush TLB/TSB.  We don't
3335                  * need to sync ref bit because we currently always set
3336                  * ref bit in tteload.
3337                  */
3338                 ASSERT(TTE_IS_REF(ttep));
3339                 if (TTE_IS_MOD(&tteold)) {
3340                         sfmmu_ttesync(sfmmup, vaddr, &tteold, pp);
3341                 }
3342                 /*
3343                  * hwtte bits shouldn't change for SRD hmeblks as long as SRD
3344                  * hmes are only used for read only text. Adding this code for
3345                  * completeness and future use of shared hmeblks with writable
3346                  * mappings of VMODSORT vnodes.
3347                  */
3348                 if (hmeblkp->hblk_shared) {
3349                         cpuset_t cpuset = sfmmu_rgntlb_demap(vaddr,
3350                             sfmmup->sfmmu_srdp->srd_hmergnp[rid], hmeblkp, 1);
3351                         xt_sync(cpuset);
3352                         SFMMU_STAT_ADD(sf_region_remap_demap, 1);
3353                 } else {
3354                         sfmmu_tlb_demap(vaddr, sfmmup, hmeblkp, 0, 0);
3355                         xt_sync(sfmmup->sfmmu_cpusran);
3356                 }
3357         }
3358
3359         if ((flags & SFMMU_NO_TSBLOAD) == 0) {
3360                 /*
3361                  * We only preload 8K and 4M mappings into the TSB, since
3362                  * 64K and 512K mappings are replicated and hence don't
3363                  * have a single, unique TSB entry. Ditto for 32M/256M.
3364                  */
3365                 if (size == TTE8K || size == TTE4M) {
3366                         sf_scd_t *scdp;
3367                         hatlockp = sfmmu_hat_enter(sfmmup);
3368                         /*
3369                          * Don't preload private TSB if the mapping is used
3370                          * by the shctx in the SCD.
3371                          */
3372                         scdp = sfmmup->sfmmu_scdp;
3373                         if (rid == SFMMU_INVALID_SHMERID || scdp == NULL ||
3374                             !SF_RGNMAP_TEST(scdp->scd_hmeregion_map, rid)) {
3375                                 sfmmu_load_tsb(sfmmup, vaddr, &sfhme->hme_tte,
3376                                     size);
3377                         }
```

```
3378                        sfmmu_hat_exit(hatlockp);
3379                }
3380        }
3381        if (pp) {
3382                if (!remap) {
3383                        HME_ADD(sfhme, pp);
3384                        atomic_inc_16(&hmeblkp->hblk_hmecnt);
3384                        atomic_add_16(&hmeblkp->hblk_hmecnt, 1);
3385                        ASSERT(hmeblkp->hblk_hmecnt > 0);

3387                        /*
3388                         * Cannot ASSERT(hmeblkp->hblk_hmecnt <= NHMENTS)
3389                         * see pageunload() for comment.
3390                         */
3391                }
3392                sfmmu_mlist_exit(pml);
3393        }

3395        return (0);
3396 }
```

*unchanged_portion_omitted_*

```
4117 /*
4118  * Function to unlock a range of addresses in an hmeblk.  It returns the
4119  * next address that needs to be unlocked.
4120  * Should be called with the hash lock held.
4121  */
4122 static caddr_t
4123 sfmmu_hblk_unlock(struct hme_blk *hmeblkp, caddr_t addr, caddr_t endaddr)
4124 {
4125        struct sf_hment *sfhme;
4126        tte_t tteold, ttemod;
4127        int ttesz, ret;

4129        ASSERT(in_hblk_range(hmeblkp, addr));
4130        ASSERT(hmeblkp->hblk_shw_bit == 0);

4132        endaddr = MIN(endaddr, get_hblk_endaddr(hmeblkp));
4133        ttesz = get_hblk_ttesz(hmeblkp);

4135        HBLKTOHME(sfhme, hmeblkp, addr);
4136        while (addr < endaddr) {
4137 readtte:
4138                sfmmu_copytte(&sfhme->hme_tte, &tteold);
4139                if (TTE_IS_VALID(&tteold)) {

4141                        ttemod = tteold;

4143                        ret = sfmmu_modifytte_try(&tteold, &ttemod,
4144                            &sfhme->hme_tte);

4146                        if (ret < 0)
4147                                goto readtte;

4149                        if (hmeblkp->hblk_lckcnt == 0)
4150                                panic("zero hblk lckcnt");

4152                        if (((uintptr_t)addr + TTEBYTES(ttesz)) >
4153                            (uintptr_t)endaddr)
4154                                panic("can't unlock large tte");

4156                        ASSERT(hmeblkp->hblk_lckcnt > 0);
4157                        atomic_dec_32(&hmeblkp->hblk_lckcnt);
4157                        atomic_add_32(&hmeblkp->hblk_lckcnt, -1);
4158                        HBLK_STACK_TRACE(hmeblkp, HBLK_UNLOCK);
4159                } else {
```



```
4160                        panic("sfmmu_hblk_unlock: invalid tte");
4161                }
4162                addr += TTEBYTES(ttesz);
4163                sfhme++;
4164        }
4165        return (addr);
4166 }
```

*unchanged_portion_omitted_*

```
6019 /*
6020  * This function unloads a range of addresses for an hmeblk.
6021  * It returns the next address to be unloaded.
6022  * It should be called with the hash lock held.
6023  */
6024 static caddr_t
6025 sfmmu_hblk_unload(struct hat *sfmmup, struct hme_blk *hmeblkp, caddr_t addr,
6026         caddr_t endaddr, demap_range_t *dmrp, uint_t flags)
6027 {
6028         tte_t   tte, ttemod;
6029         struct  sf_hment *sfhmep;
6030         int     ttesz;
6031         long    ttecnt;
6032         page_t *pp;
6033         kmutex_t *pml;
6034         int ret;
6035         int use_demap_range;

6037         ASSERT(in_hblk_range(hmeblkp, addr));
6038         ASSERT(!hmeblkp->hblk_shw_bit);
6039         ASSERT(sfmmup != NULL || hmeblkp->hblk_shared);
6040         ASSERT(sfmmup == NULL || !hmeblkp->hblk_shared);
6041         ASSERT(dmrp == NULL || !hmeblkp->hblk_shared);

6043 #ifdef DEBUG
6044         if (get_hblk_ttesz(hmeblkp) != TTE8K &&
6045             (endaddr < get_hblk_endaddr(hmeblkp))) {
6046                 panic("sfmmu_hblk_unload: partial unload of large page");
6047         }
6048 #endif /* DEBUG */

6050         endaddr = MIN(endaddr, get_hblk_endaddr(hmeblkp));
6051         ttesz = get_hblk_ttesz(hmeblkp);

6053         use_demap_range = ((dmrp == NULL) ||
6054             (TTEBYTES(ttesz) == DEMAP_RANGE_PGSZ(dmrp)));

6056         if (use_demap_range) {
6057                 DEMAP_RANGE_CONTINUE(dmrp, addr, endaddr);
6058         } else if (dmrp != NULL) {
6059                 DEMAP_RANGE_FLUSH(dmrp);
6060         }
6061         ttecnt = 0;
6062         HBLKTOHME(sfhmep, hmeblkp, addr);

6064         while (addr < endaddr) {
6065                 pml = NULL;
6066                 sfmmu_copytte(&sfhmep->hme_tte, &tte);
6067                 if (TTE_IS_VALID(&tte)) {
6068                         pp = sfhmep->hme_page;
6069                         if (pp != NULL) {
6070                                 pml = sfmmu_mlist_enter(pp);
6071                         }

6073                         /*
6074                          * Verify if hme still points to 'pp' now that
6075                          * we have p_mapping lock.
```

```
6076                          */
6077                         if (sfhmep->hme_page != pp) {
6078                                 if (pp != NULL && sfhmep->hme_page != NULL) {
6079                                         ASSERT(pml != NULL);
6080                                         sfmmu_mlist_exit(pml);
6081                                         /* Re-start this iteration. */
6082                                         continue;
6083                                 }
6084                                 ASSERT((pp != NULL) &&
6085                                     (sfhmep->hme_page == NULL));
6086                                 goto tte_unloaded;
6087                         }
6088
6089                         /*
6090                          * This point on we have both HASH and p_mapping
6091                          * lock.
6092                          */
6093                         ASSERT(pp == sfhmep->hme_page);
6094                         ASSERT(pp == NULL || sfmmu_mlist_held(pp));
6095
6096                         /*
6097                          * We need to loop on modify tte because it is
6098                          * possible for pagesync to come along and
6099                          * change the software bits beneath us.
6100                          *
6101                          * Page_unload can also invalidate the tte after
6102                          * we read tte outside of p_mapping lock.
6103                          */
6104 again:
6105                         ttemod = tte;
6106
6107                         TTE_SET_INVALID(&ttemod);
6108                         ret = sfmmu_modifytte_try(&tte, &ttemod,
6109                             &sfhmep->hme_tte);
6110
6111                         if (ret <= 0) {
6112                                 if (TTE_IS_VALID(&tte)) {
6113                                         ASSERT(ret < 0);
6114                                         goto again;
6115                                 }
6116                                 if (pp != NULL) {
6117                                         panic("sfmmu_hblk_unload: pp = 0x%p "
6118                                             "tte became invalid under mlist"
6119                                             " lock = 0x%p", (void *)pp,
6120                                             (void *)pml);
6121                                 }
6122                                 continue;
6123                         }
6124
6125                         if (!(flags & HAT_UNLOAD_NOSYNC)) {
6126                                 sfmmu_ttesync(sfmmup, addr, &tte, pp);
6127                         }
6128
6129                         /*
6130                          * Ok- we invalidated the tte. Do the rest of the job.
6131                          */
6132                         ttecnt++;
6133
6134                         if (flags & HAT_UNLOAD_UNLOCK) {
6135                                 ASSERT(hmeblkp->hblk_lckcnt > 0);
6136                                 atomic_dec_32(&hmeblkp->hblk_lckcnt);
6136                                 atomic_add_32(&hmeblkp->hblk_lckcnt, -1);
6137                                 HBLK_STACK_TRACE(hmeblkp, HBLK_UNLOCK);
6138                         }
6139
6140                         /*
```



```
6141                          * Normally we would need to flush the page
6142                          * from the virtual cache at this point in
6143                          * order to prevent a potential cache alias
6144                          * inconsistency.
6145                          * The particular scenario we need to worry
6146                          * about is:
6147                          * Given:  va1 and va2 are two virtual address
6148                          * that alias and map the same physical
6149                          * address.
6150                          * 1.   mapping exists from va1 to pa and data
6151                          * has been read into the cache.
6152                          * 2.   unload va1.
6153                          * 3.   load va2 and modify data using va2.
6154                          * 4    unload va2.
6155                          * 5.   load va1 and reference data.  Unless we
6156                          * flush the data cache when we unload we will
6157                          * get stale data.
6158                          * Fortunately, page coloring eliminates the
6159                          * above scenario by remembering the color a
6160                          * physical page was last or is currently
6161                          * mapped to.  Now, we delay the flush until
6162                          * the loading of translations.  Only when the
6163                          * new translation is of a different color
6164                          * are we forced to flush.
6165                          */
6166                         if (use_demap_range) {
6167                                 /*
6168                                  * Mark this page as needing a demap.
6169                                  */
6170                                 DEMAP_RANGE_MARKPG(dmrp, addr);
6171                         } else {
6172                                 ASSERT(sfmmup != NULL);
6173                                 ASSERT(!hmeblkp->hblk_shared);
6174                                 sfmmu_tlb_demap(addr, sfmmup, hmeblkp,
6175                                     sfmmup->sfmmu_free, 0);
6176                         }
6177
6178                         if (pp) {
6179                                 /*
6180                                  * Remove the hment from the mapping list
6181                                  */
6182                                 ASSERT(hmeblkp->hblk_hmecnt > 0);
6183
6184                                 /*
6185                                  * Again, we cannot
6186                                  * ASSERT(hmeblkp->hblk_hmecnt <= NHMENTS);
6187                                  */
6188                                 HME_SUB(sfhmep, pp);
6189                                 membar_stst();
6190                                 atomic_dec_16(&hmeblkp->hblk_hmecnt);
6190                                 atomic_add_16(&hmeblkp->hblk_hmecnt, -1);
6191                         }
6192
6193                         ASSERT(hmeblkp->hblk_vcnt > 0);
6194                         atomic_dec_16(&hmeblkp->hblk_vcnt);
6194                         atomic_add_16(&hmeblkp->hblk_vcnt, -1);
6195
6196                         ASSERT(hmeblkp->hblk_hmecnt || hmeblkp->hblk_vcnt ||
6197                             !hmeblkp->hblk_lckcnt);
6198
6199 #ifdef VAC
6200                         if (pp && (pp->p_nrm & (P_KPMC | P_KPMS | P_TNC))) {
6201                                 if (PP_ISTNC(pp)) {
6202                                         /*
6203                                          * If page was temporary
6204                                          * uncached, try to recache
```

```
6205                                     * it. Note that HME_SUB() was
6206                                     * called above so p_index and
6207                                     * mlist had been updated.
6208                                     */
6209                                    conv_tnc(pp, ttesz);
6210                            } else if (pp->p_mapping == NULL) {
6211                                    ASSERT(kpm_enable);
6212                                    /*
6213                                     * Page is marked to be in VAC conflict
6214                                     * to an existing kpm mapping and/or is
6215                                     * kpm mapped using only the regular
6216                                     * pagesize.
6217                                     */
6218                                    sfmmu_kpm_hme_unload(pp);
6219                            }
6220                    }
6221 #endif  /* VAC */
6222                } else if ((pp = sfhmep->hme_page) != NULL) {
6223                            /*
6224                             * TTE is invalid but the hme
6225                             * still exists. let pageunload
6226                             * complete its job.
6227                             */
6228                            ASSERT(pml == NULL);
6229                            pml = sfmmu_mlist_enter(pp);
6230                            if (sfhmep->hme_page != NULL) {
6231                                    sfmmu_mlist_exit(pml);
6232                                    continue;
6233                            }
6234                            ASSERT(sfhmep->hme_page == NULL);
6235                } else if (hmeblkp->hblk_hmecnt != 0) {
6236                        /*
6237                         * pageunload may have not finished decrementing
6238                         * hblk_vcnt and hblk_hmecnt. Find page_t if any and
6239                         * wait for pageunload to finish. Rely on pageunload
6240                         * to decrement hblk_hmecnt after hblk_vcnt.
6241                         */
6242                        pfn_t pfn = TTE_TO_TTEPFN(&tte);
6243                        ASSERT(pml == NULL);
6244                        if (pf_is_memory(pfn)) {
6245                                pp = page_numtopp_nolock(pfn);
6246                                if (pp != NULL) {
6247                                        pml = sfmmu_mlist_enter(pp);
6248                                        sfmmu_mlist_exit(pml);
6249                                        pml = NULL;
6250                                }
6251                        }
6252                }

6254 tte_unloaded:
6255                /*
6256                 * At this point, the tte we are looking at
6257                 * should be unloaded, and hme has been unlinked
6258                 * from page too. This is important because in
6259                 * pageunload, it does ttesync() then HME_SUB.
6260                 * We need to make sure HME_SUB has been completed
6261                 * so we know ttesync() has been completed. Otherwise,
6262                 * at exit time, after return from hat layer, VM will
6263                 * release as structure which hat_setstat() (called
6264                 * by ttesync()) needs.
6265                 */
6266 #ifdef DEBUG
6267                {
6268                        tte_t   dtte;

6270                        ASSERT(sfhmep->hme_page == NULL);
```



```
6272                            sfmmu_copytte(&sfhmep->hme_tte, &dtte);
6273                            ASSERT(!TTE_IS_VALID(&dtte));
6274                    }
6275 #endif

6277                    if (pml) {
6278                            sfmmu_mlist_exit(pml);
6279                    }

6281                    addr += TTEBYTES(ttesz);
6282                    sfhmep++;
6283                    DEMAP_RANGE_NEXTPG(dmrp);
6284            }
6285            /*
6286             * For shared hmeblks this routine is only called when region is freed
6287             * and no longer referenced.  So no need to decrement ttecnt
6288             * in the region structure here.
6289             */
6290            if (ttecnt > 0 && sfmmup != NULL) {
6291                    atomic_add_long(&sfmmup->sfmmu_ttecnt[ttesz], -ttecnt);
6292            }
6293            return (addr);
6294 }
_____unchanged_portion_omitted_

7280 cpuset_t
7281 sfmmu_pageunload(page_t *pp, struct sf_hment *sfhme, int cons)
7282 {
7283            struct hme_blk *hmeblkp;
7284            sfmmu_t *sfmmup;
7285            tte_t tte, ttemod;
7286 #ifdef DEBUG
7287            tte_t orig_old;
7288 #endif /* DEBUG */
7289            caddr_t addr;
7290            int ttesz;
7291            int ret;
7292            cpuset_t cpuset;

7294            ASSERT(pp != NULL);
7295            ASSERT(sfmmu_mlist_held(pp));
7296            ASSERT(!PP_ISKAS(pp));

7298            CPUSET_ZERO(cpuset);

7300            hmeblkp = sfmmu_hmetohblk(sfhme);

7302 readtte:
7303            sfmmu_copytte(&sfhme->hme_tte, &tte);
7304            if (TTE_IS_VALID(&tte)) {
7305                    sfmmup = hblktosfmmu(hmeblkp);
7306                    ttesz = get_hblk_ttesz(hmeblkp);
7307                    /*
7308                     * Only unload mappings of 'cons' size.
7309                     */
7310                    if (ttesz != cons)
7311                            return (cpuset);

7313                    /*
7314                     * Note that we have p_mapping lock, but no hash lock here.
7315                     * hblk_unload() has to have both hash lock AND p_mapping
7316                     * lock before it tries to modify tte. So, the tte could
7317                     * not become invalid in the sfmmu_modifytte_try() below.
7318                     */
7319                    ttemod = tte;
```

```
7320 #ifdef DEBUG
7321                 orig_old = tte;
7322 #endif /* DEBUG */

7324                 TTE_SET_INVALID(&ttemod);
7325                 ret = sfmmu_modifytte_try(&tte, &ttemod, &sfhme->hme_tte);
7326                 if (ret < 0) {
7327 #ifdef DEBUG
7328                         /* only R/M bits can change. */
7329                         chk_tte(&orig_old, &tte, &ttemod, hmeblkp);
7330 #endif /* DEBUG */
7331                         goto readtte;
7332                 }

7334                 if (ret == 0) {
7335                         panic("pageunload: cas failed?");
7336                 }

7338                 addr = tte_to_vaddr(hmeblkp, tte);

7340                 if (hmeblkp->hblk_shared) {
7341                         sf_srd_t *srdp = (sf_srd_t *)sfmmup;
7342                         uint_t rid = hmeblkp->hblk_tag.htag_rid;
7343                         sf_region_t *rgnp;
7344                         ASSERT(SFMMU_IS_SHMERID_VALID(rid));
7345                         ASSERT(rid < SFMMU_MAX_HME_REGIONS);
7346                         ASSERT(srdp != NULL);
7347                         rgnp = srdp->srd_hmergnp[rid];
7348                         SFMMU_VALIDATE_SHAREDHBLK(hmeblkp, srdp, rgnp, rid);
7349                         cpuset = sfmmu_rgntlb_demap(addr, rgnp, hmeblkp, 1);
7350                         sfmmu_ttesync(NULL, addr, &tte, pp);
7351                         ASSERT(rgnp->rgn_ttecnt[ttesz] > 0);
7352                         atomic_dec_ulong(&rgnp->rgn_ttecnt[ttesz]);
7352                         atomic_add_long(&rgnp->rgn_ttecnt[ttesz], -1);
7353                 } else {
7354                         sfmmu_ttesync(sfmmup, addr, &tte, pp);
7355                         atomic_dec_ulong(&sfmmup->sfmmu_ttecnt[ttesz]);
7355                         atomic_add_long(&sfmmup->sfmmu_ttecnt[ttesz], -1);

7357                         /*
7358                          * We need to flush the page from the virtual cache
7359                          * in order to prevent a virtual cache alias
7360                          * inconsistency. The particular scenario we need
7361                          * to worry about is:
7362                          * Given:  va1 and va2 are two virtual address that
7363                          * alias and will map the same physical address.
7364                          * 1.   mapping exists from va1 to pa and data has
7365                          *      been read into the cache.
7366                          * 2.   unload va1.
7367                          * 3.   load va2 and modify data using va2.
7368                          * 4    unload va2.
7369                          * 5.   load va1 and reference data.  Unless we flush
7370                          *      the data cache when we unload we will get
7371                          *      stale data.
7372                          * This scenario is taken care of by using virtual
7373                          * page coloring.
7374                          */
7375                         if (sfmmup->sfmmu_ismhat) {
7376                                 /*
7377                                  * Flush TSBs, TLBs and caches
7378                                  * of every process
7379                                  * sharing this ism segment.
7380                                  */
7381                                 sfmmu_hat_lock_all();
7382                                 mutex_enter(&ism_mlist_lock);
7383                                 kpreempt_disable();
```



```
7384                                 sfmmu_ismtlbcache_demap(addr, sfmmup, hmeblkp,
7385                                     pp->p_pagenum, CACHE_NO_FLUSH);
7386                                 kpreempt_enable();
7387                                 mutex_exit(&ism_mlist_lock);
7388                                 sfmmu_hat_unlock_all();
7389                                 cpuset = cpu_ready_set;
7390                         } else {
7391                                 sfmmu_tlb_demap(addr, sfmmup, hmeblkp, 0, 0);
7392                                 cpuset = sfmmup->sfmmu_cpusran;
7393                         }
7394                 }

7396                 /*
7397                  * Hme_sub has to run after ttesync() and a_rss update.
7398                  * See hblk_unload().
7399                  */
7400                 HME_SUB(sfhme, pp);
7401                 membar_stst();

7403                 /*
7404                  * We can not make ASSERT(hmeblkp->hblk_hmecnt <= NHMENTS)
7405                  * since pteload may have done a HME_ADD() right after
7406                  * we did the HME_SUB() above. Hmecnt is now maintained
7407                  * by cas only. no lock guranteed its value. The only
7408                  * gurantee we have is the hmecnt should not be less than
7409                  * what it should be so the hblk will not be taken away.
7410                  * It's also important that we decremented the hmecnt after
7411                  * we are done with hmeblkp so that this hmeblk won't be
7412                  * stolen.
7413                  */
7414                 ASSERT(hmeblkp->hblk_hmecnt > 0);
7415                 ASSERT(hmeblkp->hblk_vcnt > 0);
7416                 atomic_dec_16(&hmeblkp->hblk_vcnt);
7417                 atomic_dec_16(&hmeblkp->hblk_hmecnt);
7416                 atomic_add_16(&hmeblkp->hblk_vcnt, -1);
7417                 atomic_add_16(&hmeblkp->hblk_hmecnt, -1);
7418                 /*
7419                  * This is bug 4063182.
7420                  * XXX: fixme
7421                  * ASSERT(hmeblkp->hblk_hmecnt || hmeblkp->hblk_vcnt ||
7422                  *      !hmeblkp->hblk_lckcnt);
7423                  */
7424         } else {
7425                 panic("invalid tte? pp %p &tte %p",
7426                     (void *)pp, (void *)&tte);
7427         }

7429         return (cpuset);
7430 }
```

_____***unchanged_portion_omitted_***

```
13782 /*
13783  * SRD support
13784  */
13785 #define SRD_HASH_FUNCTION(vp)   (((((uintptr_t)(vp)) >> 4) ^ \
13786                                     (((uintptr_t)(vp)) >> 11)) & \
13787                                     srd_hashmask)

13789 /*
13790  * Attach the process to the srd struct associated with the exec vnode
13791  * from which the process is started.
13792  */
13793 void
13794 hat_join_srd(struct hat *sfmmup, vnode_t *evp)
13795 {
```

```
13796         uint_t hash = SRD_HASH_FUNCTION(evp);
13797         sf_srd_t *srdp;
13798         sf_srd_t *newsrdp;

13800         ASSERT(sfmmup != ksfmmup);
13801         ASSERT(sfmmup->sfmmu_srdp == NULL);

13803         if (!shctx_on) {
13804                 return;
13805         }

13807         VN_HOLD(evp);

13809         if (srd_buckets[hash].srdb_srdp != NULL) {
13810                 mutex_enter(&srd_buckets[hash].srdb_lock);
13811                 for (srdp = srd_buckets[hash].srdb_srdp; srdp != NULL;
13812                     srdp = srdp->srd_hash) {
13813                         if (srdp->srd_evp == evp) {
13814                                 ASSERT(srdp->srd_refcnt >= 0);
13815                                 sfmmup->sfmmu_srdp = srdp;
13816                                 atomic_inc_32(
13817                                     (volatile uint_t *)&srdp->srd_refcnt);
13816                                 atomic_add_32(
13817                                     (volatile uint_t *)&srdp->srd_refcnt, 1);
13818                                 mutex_exit(&srd_buckets[hash].srdb_lock);
13819                                 return;
13820                         }
13821                 }
13822                 mutex_exit(&srd_buckets[hash].srdb_lock);
13823         }
13824         newsrdp = kmem_cache_alloc(srd_cache, KM_SLEEP);
13825         ASSERT(newsrdp->srd_next_ismrid == 0 && newsrdp->srd_next_hmerid == 0);

13827         newsrdp->srd_evp = evp;
13828         newsrdp->srd_refcnt = 1;
13829         newsrdp->srd_hmergnfree = NULL;
13830         newsrdp->srd_ismrgnfree = NULL;

13832         mutex_enter(&srd_buckets[hash].srdb_lock);
13833         for (srdp = srd_buckets[hash].srdb_srdp; srdp != NULL;
13834             srdp = srdp->srd_hash) {
13835                 if (srdp->srd_evp == evp) {
13836                         ASSERT(srdp->srd_refcnt >= 0);
13837                         sfmmup->sfmmu_srdp = srdp;
13838                         atomic_inc_32((volatile uint_t *)&srdp->srd_refcnt);
13838                         atomic_add_32((volatile uint_t *)&srdp->srd_refcnt, 1);
13839                         mutex_exit(&srd_buckets[hash].srdb_lock);
13840                         kmem_cache_free(srd_cache, newsrdp);
13841                         return;
13842                 }
13843         }
13844         newsrdp->srd_hash = srd_buckets[hash].srdb_srdp;
13845         srd_buckets[hash].srdb_srdp = newsrdp;
13846         sfmmup->sfmmu_srdp = newsrdp;

13848         mutex_exit(&srd_buckets[hash].srdb_lock);

13850 }

13852 static void
13853 sfmmu_leave_srd(sfmmu_t *sfmmup)
13854 {
13855         vnode_t *evp;
13856         sf_srd_t *srdp = sfmmup->sfmmu_srdp;
13857         uint_t hash;
13858         sf_srd_t **prev_srdpp;
```



```
13859         sf_region_t *rgnp;
13860         sf_region_t *nrgnp;
13861 #ifdef DEBUG
13862         int rgns = 0;
13863 #endif
13864         int i;

13866         ASSERT(sfmmup != ksfmmup);
13867         ASSERT(srdp != NULL);
13868         ASSERT(srdp->srd_refcnt > 0);
13869         ASSERT(sfmmup->sfmmu_scdp == NULL);
13870         ASSERT(sfmmup->sfmmu_free == 1);

13872         sfmmup->sfmmu_srdp = NULL;
13873         evp = srdp->srd_evp;
13874         ASSERT(evp != NULL);
13875         if (atomic_dec_32_nv((volatile uint_t *)&srdp->srd_refcnt)) {
13875         if (atomic_add_32_nv(
13876             (volatile uint_t *)&srdp->srd_refcnt, -1)) {
13876                 VN_RELE(evp);
13877                 return;
13878         }

13880         hash = SRD_HASH_FUNCTION(evp);
13881         mutex_enter(&srd_buckets[hash].srdb_lock);
13882         for (prev_srdpp = &srd_buckets[hash].srdb_srdp;
13883             (srdp = *prev_srdpp) != NULL; prev_srdpp = &srdp->srd_hash) {
13884                 if (srdp->srd_evp == evp) {
13885                         break;
13886                 }
13887         }
13888         if (srdp == NULL || srdp->srd_refcnt) {
13889                 mutex_exit(&srd_buckets[hash].srdb_lock);
13890                 VN_RELE(evp);
13891                 return;
13892         }
13893         *prev_srdpp = srdp->srd_hash;
13894         mutex_exit(&srd_buckets[hash].srdb_lock);

13896         ASSERT(srdp->srd_refcnt == 0);
13897         VN_RELE(evp);

13899 #ifdef DEBUG
13900         for (i = 0; i < SFMMU_MAX_REGION_BUCKETS; i++) {
13901                 ASSERT(srdp->srd_rgnhash[i] == NULL);
13902         }
13903 #endif /* DEBUG */

13905         /* free each hme regions in the srd */
13906         for (rgnp = srdp->srd_hmergnfree; rgnp != NULL; rgnp = nrgnp) {
13907                 nrgnp = rgnp->rgn_next;
13908                 ASSERT(rgnp->rgn_id < srdp->srd_next_hmerid);
13909                 ASSERT(rgnp->rgn_refcnt == 0);
13910                 ASSERT(rgnp->rgn_sfmmu_head == NULL);
13911                 ASSERT(rgnp->rgn_flags & SFMMU_REGION_FREE);
13912                 ASSERT(rgnp->rgn_hmeflags == 0);
13913                 ASSERT(srdp->srd_hmergnp[rgnp->rgn_id] == rgnp);
13914 #ifdef DEBUG
13915                 for (i = 0; i < MMU_PAGE_SIZES; i++) {
13916                         ASSERT(rgnp->rgn_ttecnt[i] == 0);
13917                 }
13918                 rgns++;
13919 #endif /* DEBUG */
13920                 kmem_cache_free(region_cache, rgnp);
13921         }
13922         ASSERT(rgns == srdp->srd_next_hmerid);
```

```
13924 #ifdef DEBUG
13925         rgns = 0;
13926 #endif
13927         /* free each ism rgns in the srd */
13928         for (rgnp = srdp->srd_ismrgnfree; rgnp != NULL; rgnp = nrgnp) {
13929                 nrgnp = rgnp->rgn_next;
13930                 ASSERT(rgnp->rgn_id < srdp->srd_next_ismrid);
13931                 ASSERT(rgnp->rgn_refcnt == 0);
13932                 ASSERT(rgnp->rgn_sfmmu_head == NULL);
13933                 ASSERT(rgnp->rgn_flags & SFMMU_REGION_FREE);
13934                 ASSERT(srdp->srd_ismrgnp[rgnp->rgn_id] == rgnp);
13935 #ifdef DEBUG
13936                 for (i = 0; i < MMU_PAGE_SIZES; i++) {
13937                         ASSERT(rgnp->rgn_ttecnt[i] == 0);
13938                 }
13939                 rgns++;
13940 #endif /* DEBUG */
13941                 kmem_cache_free(region_cache, rgnp);
13942         }
13943         ASSERT(rgns == srdp->srd_next_ismrid);
13944         ASSERT(srdp->srd_ismbusyrgns == 0);
13945         ASSERT(srdp->srd_hmebusyrgns == 0);

13947         srdp->srd_next_ismrid = 0;
13948         srdp->srd_next_hmerid = 0;

13950         bzero((void *)srdp->srd_ismrgnp,
13951             sizeof (sf_region_t *) * SFMMU_MAX_ISM_REGIONS);
13952         bzero((void *)srdp->srd_hmergnp,
13953             sizeof (sf_region_t *) * SFMMU_MAX_HME_REGIONS);

13955         ASSERT(srdp->srd_scdp == NULL);
13956         kmem_cache_free(srd_cache, srdp);
13957 }
```

*unchanged_portion_omitted_*

```
13981 /*
13982  * The caller makes sure hat_join_region()/hat_leave_region() can't be called
13983  * at the same time for the same process and address range. This is ensured by
13984  * the fact that address space is locked as writer when a process joins the
13985  * regions. Therefore there's no need to hold an srd lock during the entire
13986  * execution of hat_join_region()/hat_leave_region().
13987  */

13989 #define RGN_HASH_FUNCTION(obj)  (((((uintptr_t)(obj)) >> 4) ^ \
13990                                   (((uintptr_t)(obj)) >> 11)) & \
13991                                       srd_rgn_hashmask)
13992 /*
13993  * This routine implements the shared context functionality required when
13994  * attaching a segment to an address space. It must be called from
13995  * hat_share() for D(ISM) segments and from segvn_create() for segments
13996  * with the MAP_PRIVATE and MAP_TEXT flags set. It returns a region_cookie
13997  * which is saved in the private segment data for hme segments and
13998  * the ism_map structure for ism segments.
13999  */
14000 hat_region_cookie_t
14001 hat_join_region(struct hat *sfmmup,
14002         caddr_t r_saddr,
14003         size_t r_size,
14004         void *r_obj,
14005         u_offset_t r_objoff,
14006         uchar_t r_perm,
14007         uchar_t r_pgszc,
14008         hat_rgn_cb_func_t r_cb_function,
14009         uint_t flags)
```



```
14010 {
14011         sf_srd_t *srdp = sfmmup->sfmmu_srdp;
14012         uint_t rhash;
14013         uint_t rid;
14014         hatlock_t *hatlockp;
14015         sf_region_t *rgnp;
14016         sf_region_t *new_rgnp = NULL;
14017         int i;
14018         uint16_t *nextidp;
14019         sf_region_t **freelistp;
14020         int maxids;
14021         sf_region_t **rarrp;
14022         uint16_t *busyrgnsp;
14023         ulong_t rttecnt;
14024         uchar_t tteflag;
14025         uchar_t r_type = flags & HAT_REGION_TYPE_MASK;
14026         int text = (r_type == HAT_REGION_TEXT);

14028         if (srdp == NULL || r_size == 0) {
14029                 return (HAT_INVALID_REGION_COOKIE);
14030         }

14032         ASSERT(sfmmup->sfmmu_xhat_provider == NULL);
14033         ASSERT(sfmmup != ksfmmup);
14034         ASSERT(AS_WRITE_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));
14035         ASSERT(srdp->srd_refcnt > 0);
14036         ASSERT(!(flags & ~HAT_REGION_TYPE_MASK));
14037         ASSERT(flags == HAT_REGION_TEXT || flags == HAT_REGION_ISM);
14038         ASSERT(r_pgszc < mmu_page_sizes);
14039         if (!IS_P2ALIGNED(r_saddr, TTEBYTES(r_pgszc)) ||
14040             !IS_P2ALIGNED(r_size, TTEBYTES(r_pgszc))) {
14041                 panic("hat_join_region: region addr or size is not aligned\n");
14042         }


14045         r_type = (r_type == HAT_REGION_ISM) ? SFMMU_REGION_ISM :
14046             SFMMU_REGION_HME;
14047         /*
14048          * Currently only support shared hmes for the read only main text
14049          * region.
14050          */
14051         if (r_type == SFMMU_REGION_HME && ((r_obj != srdp->srd_evp) ||
14052             (r_perm & PROT_WRITE))) {
14053                 return (HAT_INVALID_REGION_COOKIE);
14054         }

14056         rhash = RGN_HASH_FUNCTION(r_obj);

14058         if (r_type == SFMMU_REGION_ISM) {
14059                 nextidp = &srdp->srd_next_ismrid;
14060                 freelistp = &srdp->srd_ismrgnfree;
14061                 maxids = SFMMU_MAX_ISM_REGIONS;
14062                 rarrp = srdp->srd_ismrgnp;
14063                 busyrgnsp = &srdp->srd_ismbusyrgns;
14064         } else {
14065                 nextidp = &srdp->srd_next_hmerid;
14066                 freelistp = &srdp->srd_hmergnfree;
14067                 maxids = SFMMU_MAX_HME_REGIONS;
14068                 rarrp = srdp->srd_hmergnp;
14069                 busyrgnsp = &srdp->srd_hmebusyrgns;
14070         }

14072         mutex_enter(&srdp->srd_mutex);

14074         for (rgnp = srdp->srd_rgnhash[rhash]; rgnp != NULL;
14075             rgnp = rgnp->rgn_hash) {
```

```
14076                 if (rgnp->rgn_saddr == r_saddr && rgnp->rgn_size == r_size &&
14077                     rgnp->rgn_obj == r_obj && rgnp->rgn_objoff == r_objoff &&
14078                     rgnp->rgn_perm == r_perm && rgnp->rgn_pgszc == r_pgszc) {
14079                         break;
14080                 }
14081         }

14083 rfound:
14084         if (rgnp != NULL) {
14085                 ASSERT((rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK) == r_type);
14086                 ASSERT(rgnp->rgn_cb_function == r_cb_function);
14087                 ASSERT(rgnp->rgn_refcnt >= 0);
14088                 rid = rgnp->rgn_id;
14089                 ASSERT(rid < maxids);
14090                 ASSERT(rarrp[rid] == rgnp);
14091                 ASSERT(rid < *nextidp);
14092                 atomic_inc_32((volatile uint_t *)&rgnp->rgn_refcnt);
14093                 atomic_add_32((volatile uint_t *)&rgnp->rgn_refcnt, 1);
14093                 mutex_exit(&srdp->srd_mutex);
14094                 if (new_rgnp != NULL) {
14095                         kmem_cache_free(region_cache, new_rgnp);
14096                 }
14097                 if (r_type == SFMMU_REGION_HME) {
14098                         int myjoin =
14099                             (sfmmup == astosfmmu(curthread->t_procp->p_as));

14101                         sfmmu_link_to_hmeregion(sfmmup, rgnp);
14102                         /*
14103                          * bitmap should be updated after linking sfmmu on
14104                          * region list so that pageunload() doesn't skip
14105                          * TSB/TLB flush. As soon as bitmap is updated another
14106                          * thread in this process can already start accessing
14107                          * this region.
14108                          */
14109                         /*
14110                          * Normally ttecnt accounting is done as part of
14111                          * pagefault handling. But a process may not take any
14112                          * pagefaults on shared hmeblks created by some other
14113                          * process. To compensate for this assume that the
14114                          * entire region will end up faulted in using
14115                          * the region's pagesize.
14116                          *
14117                          */
14118                         if (r_pgszc > TTE8K) {
14119                                 tteflag = 1 << r_pgszc;
14120                                 if (disable_large_pages & tteflag) {
14121                                         tteflag = 0;
14122                                 }
14123                         } else {
14124                                 tteflag = 0;
14125                         }
14126                         if (tteflag && !(sfmmup->sfmmu_rtteflags & tteflag)) {
14127                                 hatlockp = sfmmu_hat_enter(sfmmup);
14128                                 sfmmup->sfmmu_rtteflags |= tteflag;
14129                                 sfmmu_hat_exit(hatlockp);
14130                         }
14131                         hatlockp = sfmmu_hat_enter(sfmmup);

14133                         /*
14134                          * Preallocate 1/4 of ttecnt's in 8K TSB for >= 4M
14135                          * region to allow for large page allocation failure.
14136                          */
14137                         if (r_pgszc >= TTE4M) {
14138                                 sfmmup->sfmmu_tsb0_4minflcnt +=
14139                                     r_size >> (TTE_PAGE_SHIFT(TTE8K) + 2);
14140                         }
```



```
14142                         /* update sfmmu_ttecnt with the shme rgn ttecnt */
14143                         rttecnt = r_size >> TTE_PAGE_SHIFT(r_pgszc);
14144                         atomic_add_long(&sfmmup->sfmmu_ttecnt[r_pgszc],
14145                             rttecnt);

14147                         if (text && r_pgszc >= TTE4M &&
14148                             (tteflag || ((disable_large_pages >> TTE4M) &
14149                             ((1 << (r_pgszc - TTE4M + 1)) - 1))) &&
14150                             !SFMMU_FLAGS_ISSET(sfmmup, HAT_4MTEXT_FLAG)) {
14151                                 SFMMU_FLAGS_SET(sfmmup, HAT_4MTEXT_FLAG);
14152                         }

14154                         sfmmu_hat_exit(hatlockp);
14155                         /*
14156                          * On Panther we need to make sure TLB is programmed
14157                          * to accept 32M/256M pages.  Call
14158                          * sfmmu_check_page_sizes() now to make sure TLB is
14159                          * setup before making hmeregions visible to other
14160                          * threads.
14161                          */
14162                         sfmmu_check_page_sizes(sfmmup, 1);
14163                         hatlockp = sfmmu_hat_enter(sfmmup);
14164                         SF_RGNMAP_ADD(sfmmup->sfmmu_hmeregion_map, rid);

14166                         /*
14167                          * if context is invalid tsb miss exception code will
14168                          * call sfmmu_check_page_sizes() and update tsbmiss
14169                          * area later.
14170                          */
14171                         kpreempt_disable();
14172                         if (myjoin &&
14173                             (sfmmup->sfmmu_ctxs[CPU_MMU_IDX(CPU)].cnum
14174                             != INVALID_CONTEXT)) {
14175                                 struct tsbmiss *tsbmp;

14177                                 tsbmp = &tsbmiss_area[CPU->cpu_id];
14178                                 ASSERT(sfmmup == tsbmp->usfmmup);
14179                                 BT_SET(tsbmp->shmermap, rid);
14180                                 if (r_pgszc > TTE64K) {
14181                                         tsbmp->uhat_rtteflags |= tteflag;
14182                                 }

14184                         }
14185                         kpreempt_enable();

14187                         sfmmu_hat_exit(hatlockp);
14188                         ASSERT((hat_region_cookie_t)((uint64_t)rid) !=
14189                             HAT_INVALID_REGION_COOKIE);
14190                 } else {
14191                         hatlockp = sfmmu_hat_enter(sfmmup);
14192                         SF_RGNMAP_ADD(sfmmup->sfmmu_ismregion_map, rid);
14193                         sfmmu_hat_exit(hatlockp);
14194                 }
14195                 ASSERT(rid < maxids);

14197                 if (r_type == SFMMU_REGION_ISM) {
14198                         sfmmu_find_scd(sfmmup);
14199                 }
14200                 return ((hat_region_cookie_t)((uint64_t)rid));
14201         }

14203         ASSERT(new_rgnp == NULL);

14205         if (*busyrgnsp >= maxids) {
14206                 mutex_exit(&srdp->srd_mutex);
```

```
14207                   return (HAT_INVALID_REGION_COOKIE);
14208           }

14210           ASSERT(MUTEX_HELD(&srdp->srd_mutex));
14211           if (*freelistp != NULL) {
14212                   rgnp = *freelistp;
14213                   *freelistp = rgnp->rgn_next;
14214                   ASSERT(rgnp->rgn_id < *nextidp);
14215                   ASSERT(rgnp->rgn_id < maxids);
14216                   ASSERT(rgnp->rgn_flags & SFMMU_REGION_FREE);
14217                   ASSERT((rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK)
14218                       == r_type);
14219                   ASSERT(rarrp[rgnp->rgn_id] == rgnp);
14220                   ASSERT(rgnp->rgn_hmeflags == 0);
14221           } else {
14222                   /*
14223                    * release local locks before memory allocation.
14224                    */
14225                   mutex_exit(&srdp->srd_mutex);

14227                   new_rgnp = kmem_cache_alloc(region_cache, KM_SLEEP);

14229                   mutex_enter(&srdp->srd_mutex);
14230                   for (rgnp = srdp->srd_rgnhash[rhash]; rgnp != NULL;
14231                       rgnp = rgnp->rgn_hash) {
14232                           if (rgnp->rgn_saddr == r_saddr &&
14233                               rgnp->rgn_size == r_size &&
14234                               rgnp->rgn_obj == r_obj &&
14235                               rgnp->rgn_objoff == r_objoff &&
14236                               rgnp->rgn_perm == r_perm &&
14237                               rgnp->rgn_pgszc == r_pgszc) {
14238                                   break;
14239                           }
14240                   }
14241                   if (rgnp != NULL) {
14242                           goto rfound;
14243                   }

14245                   if (*nextidp >= maxids) {
14246                           mutex_exit(&srdp->srd_mutex);
14247                           goto fail;
14248                   }
14249                   rgnp = new_rgnp;
14250                   new_rgnp = NULL;
14251                   rgnp->rgn_id = (*nextidp)++;
14252                   ASSERT(rgnp->rgn_id < maxids);
14253                   ASSERT(rarrp[rgnp->rgn_id] == NULL);
14254                   rarrp[rgnp->rgn_id] = rgnp;
14255           }

14257           ASSERT(rgnp->rgn_sfmmu_head == NULL);
14258           ASSERT(rgnp->rgn_hmeflags == 0);
14259 #ifdef DEBUG
14260           for (i = 0; i < MMU_PAGE_SIZES; i++) {
14261                   ASSERT(rgnp->rgn_ttecnt[i] == 0);
14262           }
14263 #endif
14264           rgnp->rgn_saddr = r_saddr;
14265           rgnp->rgn_size = r_size;
14266           rgnp->rgn_obj = r_obj;
14267           rgnp->rgn_objoff = r_objoff;
14268           rgnp->rgn_perm = r_perm;
14269           rgnp->rgn_pgszc = r_pgszc;
14270           rgnp->rgn_flags = r_type;
14271           rgnp->rgn_refcnt = 0;
14272           rgnp->rgn_cb_function = r_cb_function;
```



```
14273           rgnp->rgn_hash = srdp->srd_rgnhash[rhash];
14274           srdp->srd_rgnhash[rhash] = rgnp;
14275           (*busyrgnsp)++;
14276           ASSERT(*busyrgnsp <= maxids);
14277           goto rfound;

14279 fail:
14280           ASSERT(new_rgnp != NULL);
14281           kmem_cache_free(region_cache, new_rgnp);
14282           return (HAT_INVALID_REGION_COOKIE);
14283 }

14285 /*
14286  * This function implements the shared context functionality required
14287  * when detaching a segment from an address space. It must be called
14288  * from hat_unshare() for all D(ISM) segments and from segvn_unmap(),
14289  * for segments with a valid region_cookie.
14290  * It will also be called from all seg_vn routines which change a
14291  * segment's attributes such as segvn_setprot(), segvn_setpagesize(),
14292  * segvn_clrszc() & segvn_advise(), as well as in the case of COW fault
14293  * from segvn_fault().
14294  */
14295 void
14296 hat_leave_region(struct hat *sfmmup, hat_region_cookie_t rcookie, uint_t flags)
14297 {
14298           sf_srd_t *srdp = sfmmup->sfmmu_srdp;
14299           sf_scd_t *scdp;
14300           uint_t rhash;
14301           uint_t rid = (uint_t)((uint64_t)rcookie);
14302           hatlock_t *hatlockp = NULL;
14303           sf_region_t *rgnp;
14304           sf_region_t **prev_rgnpp;
14305           sf_region_t *cur_rgnp;
14306           void *r_obj;
14307           int i;
14308           caddr_t r_saddr;
14309           caddr_t r_eaddr;
14310           size_t  r_size;
14311           uchar_t r_pgszc;
14312           uchar_t r_type = flags & HAT_REGION_TYPE_MASK;

14314           ASSERT(sfmmup != ksfmmup);
14315           ASSERT(srdp != NULL);
14316           ASSERT(srdp->srd_refcnt > 0);
14317           ASSERT(!(flags & ~HAT_REGION_TYPE_MASK));
14318           ASSERT(flags == HAT_REGION_TEXT || flags == HAT_REGION_ISM);
14319           ASSERT(!sfmmup->sfmmu_free || sfmmup->sfmmu_scdp == NULL);

14321           r_type = (r_type == HAT_REGION_ISM) ? SFMMU_REGION_ISM :
14322               SFMMU_REGION_HME;

14324           if (r_type == SFMMU_REGION_ISM) {
14325                   ASSERT(SFMMU_IS_ISMRID_VALID(rid));
14326                   ASSERT(rid < SFMMU_MAX_ISM_REGIONS);
14327                   rgnp = srdp->srd_ismrgnp[rid];
14328           } else {
14329                   ASSERT(SFMMU_IS_SHMERID_VALID(rid));
14330                   ASSERT(rid < SFMMU_MAX_HME_REGIONS);
14331                   rgnp = srdp->srd_hmergnp[rid];
14332           }
14333           ASSERT(rgnp != NULL);
14334           ASSERT(rgnp->rgn_id == rid);
14335           ASSERT((rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK) == r_type);
14336           ASSERT(!(rgnp->rgn_flags & SFMMU_REGION_FREE));
14337           ASSERT(AS_LOCK_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));
```

```
14339         ASSERT(sfmmup->sfmmu_xhat_provider == NULL);
14340         if (r_type == SFMMU_REGION_HME && sfmmup->sfmmu_as->a_xhat != NULL) {
14341                 xhat_unload_callback_all(sfmmup->sfmmu_as, rgnp->rgn_saddr,
14342                     rgnp->rgn_size, 0, NULL);
14343         }

14345         if (sfmmup->sfmmu_free) {
14346                 ulong_t rttecnt;
14347                 r_pgszc = rgnp->rgn_pgszc;
14348                 r_size = rgnp->rgn_size;

14350                 ASSERT(sfmmup->sfmmu_scdp == NULL);
14351                 if (r_type == SFMMU_REGION_ISM) {
14352                         SF_RGNMAP_DEL(sfmmup->sfmmu_ismregion_map, rid);
14353                 } else {
14354                         /* update shme rgns ttecnt in sfmmu_ttecnt */
14355                         rttecnt = r_size >> TTE_PAGE_SHIFT(r_pgszc);
14356                         ASSERT(sfmmup->sfmmu_ttecnt[r_pgszc] >= rttecnt);

14358                         atomic_add_long(&sfmmup->sfmmu_ttecnt[r_pgszc],
14359                             -rttecnt);

14361                         SF_RGNMAP_DEL(sfmmup->sfmmu_hmeregion_map, rid);
14362                 }
14363         } else if (r_type == SFMMU_REGION_ISM) {
14364                 hatlockp = sfmmu_hat_enter(sfmmup);
14365                 ASSERT(rid < srdp->srd_next_ismrid);
14366                 SF_RGNMAP_DEL(sfmmup->sfmmu_ismregion_map, rid);
14367                 scdp = sfmmup->sfmmu_scdp;
14368                 if (scdp != NULL &&
14369                     SF_RGNMAP_TEST(scdp->scd_ismregion_map, rid)) {
14370                         sfmmu_leave_scd(sfmmup, r_type);
14371                         ASSERT(sfmmu_hat_lock_held(sfmmup));
14372                 }
14373                 sfmmu_hat_exit(hatlockp);
14374         } else {
14375                 ulong_t rttecnt;
14376                 r_pgszc = rgnp->rgn_pgszc;
14377                 r_saddr = rgnp->rgn_saddr;
14378                 r_size = rgnp->rgn_size;
14379                 r_eaddr = r_saddr + r_size;

14381                 ASSERT(r_type == SFMMU_REGION_HME);
14382                 hatlockp = sfmmu_hat_enter(sfmmup);
14383                 ASSERT(rid < srdp->srd_next_hmerid);
14384                 SF_RGNMAP_DEL(sfmmup->sfmmu_hmeregion_map, rid);

14386                 /*
14387                  * If region is part of an SCD call sfmmu_leave_scd().
14388                  * Otherwise if process is not exiting and has valid context
14389                  * just drop the context on the floor to lose stale TLB
14390                  * entries and force the update of tsb miss area to reflect
14391                  * the new region map. After that clean our TSB entries.
14392                  */
14393                 scdp = sfmmup->sfmmu_scdp;
14394                 if (scdp != NULL &&
14395                     SF_RGNMAP_TEST(scdp->scd_hmeregion_map, rid)) {
14396                         sfmmu_leave_scd(sfmmup, r_type);
14397                         ASSERT(sfmmu_hat_lock_held(sfmmup));
14398                 }
14399                 sfmmu_invalidate_ctx(sfmmup);

14401                 i = TTE8K;
14402                 while (i < mmu_page_sizes) {
14403                         if (rgnp->rgn_ttecnt[i] != 0) {
14404                                 sfmmu_unload_tsb_range(sfmmup, r_saddr,
```



```
14405                                     r_eaddr, i);
14406                                 if (i < TTE4M) {
14407                                         i = TTE4M;
14408                                         continue;
14409                                 } else {
14410                                         break;
14411                                 }
14412                         }
14413                         i++;
14414                 }
14415                 /* Remove the preallocated 1/4 8k ttecnt for 4M regions. */
14416                 if (r_pgszc >= TTE4M) {
14417                         rttecnt = r_size >> (TTE_PAGE_SHIFT(TTE8K) + 2);
14418                         ASSERT(sfmmup->sfmmu_tsb0_4minflcnt >=
14419                             rttecnt);
14420                         sfmmup->sfmmu_tsb0_4minflcnt -= rttecnt;
14421                 }

14423                 /* update shme rgns ttecnt in sfmmu_ttecnt */
14424                 rttecnt = r_size >> TTE_PAGE_SHIFT(r_pgszc);
14425                 ASSERT(sfmmup->sfmmu_ttecnt[r_pgszc] >= rttecnt);
14426                 atomic_add_long(&sfmmup->sfmmu_ttecnt[r_pgszc], -rttecnt);

14428                 sfmmu_hat_exit(hatlockp);
14429                 if (scdp != NULL && sfmmup->sfmmu_scdp == NULL) {
14430                         /* sfmmup left the scd, grow private tsb */
14431                         sfmmu_check_page_sizes(sfmmup, 1);
14432                 } else {
14433                         sfmmu_check_page_sizes(sfmmup, 0);
14434                 }
14435         }

14437         if (r_type == SFMMU_REGION_HME) {
14438                 sfmmu_unlink_from_hmeregion(sfmmup, rgnp);
14439         }

14441         r_obj = rgnp->rgn_obj;
14442         if (atomic_dec_32_nv((volatile uint_t *)&rgnp->rgn_refcnt)) {
14443         if (atomic_add_32_nv((volatile uint_t *)&rgnp->rgn_refcnt, -1)) {
14443                 return;
14444         }

14446         /*
14447          * looks like nobody uses this region anymore. Free it.
14448          */
14449         rhash = RGN_HASH_FUNCTION(r_obj);
14450         mutex_enter(&srdp->srd_mutex);
14451         for (prev_rgnpp = &srdp->srd_rgnhash[rhash];
14452             (cur_rgnp = *prev_rgnpp) != NULL;
14453             prev_rgnpp = &cur_rgnp->rgn_hash) {
14454                 if (cur_rgnp == rgnp && cur_rgnp->rgn_refcnt == 0) {
14455                         break;
14456                 }
14457         }

14459         if (cur_rgnp == NULL) {
14460                 mutex_exit(&srdp->srd_mutex);
14461                 return;
14462         }

14464         ASSERT((rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK) == r_type);
14465         *prev_rgnpp = rgnp->rgn_hash;
14466         if (r_type == SFMMU_REGION_ISM) {
14467                 rgnp->rgn_flags |= SFMMU_REGION_FREE;
14468                 ASSERT(rid < srdp->srd_next_ismrid);
14469                 rgnp->rgn_next = srdp->srd_ismrgnfree;
```

```
14470                   srdp->srd_ismrgnfree = rgnp;
14471                   ASSERT(srdp->srd_ismbusyrgns > 0);
14472                   srdp->srd_ismbusyrgns--;
14473                   mutex_exit(&srdp->srd_mutex);
14474                   return;
14475           }
14476           mutex_exit(&srdp->srd_mutex);

14478           /*
14479            * Destroy region's hmeblks.
14480            */
14481           sfmmu_unload_hmeregion(srdp, rgnp);

14483           rgnp->rgn_hmeflags = 0;

14485           ASSERT(rgnp->rgn_sfmmu_head == NULL);
14486           ASSERT(rgnp->rgn_id == rid);
14487           for (i = 0; i < MMU_PAGE_SIZES; i++) {
14488                   rgnp->rgn_ttecnt[i] = 0;
14489           }
14490           rgnp->rgn_flags |= SFMMU_REGION_FREE;
14491           mutex_enter(&srdp->srd_mutex);
14492           ASSERT(rid < srdp->srd_next_hmerid);
14493           rgnp->rgn_next = srdp->srd_hmergnfree;
14494           srdp->srd_hmergnfree = rgnp;
14495           ASSERT(srdp->srd_hmebusyrgns > 0);
14496           srdp->srd_hmebusyrgns--;
14497           mutex_exit(&srdp->srd_mutex);
14498   }

14500   /*
14501    * For now only called for hmeblk regions and not for ISM regions.
14502    */
14503   void
14504   hat_dup_region(struct hat *sfmmup, hat_region_cookie_t rcookie)
14505   {
14506           sf_srd_t *srdp = sfmmup->sfmmu_srdp;
14507           uint_t rid = (uint_t)((uint64_t)rcookie);
14508           sf_region_t *rgnp;
14509           sf_rgn_link_t *rlink;
14510           sf_rgn_link_t *hrlink;
14511           ulong_t rttecnt;

14513           ASSERT(sfmmup != ksfmmup);
14514           ASSERT(srdp != NULL);
14515           ASSERT(srdp->srd_refcnt > 0);

14517           ASSERT(rid < srdp->srd_next_hmerid);
14518           ASSERT(SFMMU_IS_SHMERID_VALID(rid));
14519           ASSERT(rid < SFMMU_MAX_HME_REGIONS);

14521           rgnp = srdp->srd_hmergnp[rid];
14522           ASSERT(rgnp->rgn_refcnt > 0);
14523           ASSERT(rgnp->rgn_id == rid);
14524           ASSERT((rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK) == SFMMU_REGION_HME);
14525           ASSERT(!(rgnp->rgn_flags & SFMMU_REGION_FREE));

14527           atomic_inc_32((volatile uint_t *)&rgnp->rgn_refcnt);
14528           atomic_add_32((volatile uint_t *)&rgnp->rgn_refcnt, 1);

14529           /* LINTED: constant in conditional context */
14530           SFMMU_HMERID2RLINKP(sfmmup, rid, rlink, 1, 0);
14531           ASSERT(rlink != NULL);
14532           mutex_enter(&rgnp->rgn_mutex);
14533           ASSERT(rgnp->rgn_sfmmu_head != NULL);
14534           /* LINTED: constant in conditional context */
```



```
14535           SFMMU_HMERID2RLINKP(rgnp->rgn_sfmmu_head, rid, hrlink, 0, 0);
14536           ASSERT(hrlink != NULL);
14537           ASSERT(hrlink->prev == NULL);
14538           rlink->next = rgnp->rgn_sfmmu_head;
14539           rlink->prev = NULL;
14540           hrlink->prev = sfmmup;
14541           /*
14542            * make sure rlink's next field is correct
14543            * before making this link visible.
14544            */
14545           membar_stst();
14546           rgnp->rgn_sfmmu_head = sfmmup;
14547           mutex_exit(&rgnp->rgn_mutex);

14549           /* update sfmmu_ttecnt with the shme rgn ttecnt */
14550           rttecnt = rgnp->rgn_size >> TTE_PAGE_SHIFT(rgnp->rgn_pgszc);
14551           atomic_add_long(&sfmmup->sfmmu_ttecnt[rgnp->rgn_pgszc], rttecnt);
14552           /* update tsb0 inflation count */
14553           if (rgnp->rgn_pgszc >= TTE4M) {
14554                   sfmmup->sfmmu_tsb0_4minflcnt +=
14555                       rgnp->rgn_size >> (TTE_PAGE_SHIFT(TTE8K) + 2);
14556           }
14557           /*
14558            * Update regionid bitmask without hat lock since no other thread
14559            * can update this region bitmask right now.
14560            */
14561           SF_RGNMAP_ADD(sfmmup->sfmmu_hmeregion_map, rid);
14562   }
_____unchanged_portion_omitted_

15233   /*
15234    * This routine is called in order to check if there is an SCD which matches
15235    * the process's region map if not then a new SCD may be created.
15236    */
15237   static void
15238   sfmmu_find_scd(sfmmu_t *sfmmup)
15239   {
15240           sf_srd_t *srdp = sfmmup->sfmmu_srdp;
15241           sf_scd_t *scdp, *new_scdp;
15242           int ret;

15244           ASSERT(srdp != NULL);
15245           ASSERT(AS_WRITE_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));

15247           mutex_enter(&srdp->srd_scd_mutex);
15248           for (scdp = srdp->srd_scdp; scdp != NULL;
15249               scdp = scdp->scd_next) {
15250                   SF_RGNMAP_EQUAL(&scdp->scd_region_map,
15251                       &sfmmup->sfmmu_region_map, ret);
15252                   if (ret == 1) {
15253                           SF_SCD_INCR_REF(scdp);
15254                           mutex_exit(&srdp->srd_scd_mutex);
15255                           sfmmu_join_scd(scdp, sfmmup);
15256                           ASSERT(scdp->scd_refcnt >= 2);
15257                           atomic_dec_32((volatile uint32_t *)&scdp->scd_refcnt);
15258                           atomic_add_32((volatile uint32_t *)
15259                               &scdp->scd_refcnt, -1);
15258                           return;
15259                   } else {
15260                           /*
15261                            * If the sfmmu region map is a subset of the scd
15262                            * region map, then the assumption is that this process
15263                            * will continue attaching to ISM segments until the
15264                            * region maps are equal.
15265                            */
15266                           SF_RGNMAP_IS_SUBSET(&scdp->scd_region_map,
```

```
15267                               &sfmmup->sfmmu_region_map, ret);
15268                           if (ret == 1) {
15269                                   mutex_exit(&srdp->srd_scd_mutex);
15270                                   return;
15271                           }
15272                   }
15273           }

15275           ASSERT(scdp == NULL);
15276           /*
15277            * No matching SCD has been found, create a new one.
15278            */
15279           if ((new_scdp = sfmmu_alloc_scd(srdp, &sfmmup->sfmmu_region_map)) ==
15280               NULL) {
15281                   mutex_exit(&srdp->srd_scd_mutex);
15282                   return;
15283           }

15285           /*
15286            * sfmmu_alloc_scd() returns with a ref count of 1 on the scd.
15287            */

15289           /* Set scd_rttecnt for shme rgns in SCD */
15290           sfmmu_set_scd_rttecnt(srdp, new_scdp);

15292           /*
15293            * Link scd onto srd_scdp list and scd sfmmu onto region/iment lists.
15294            */
15295           sfmmu_link_scd_to_regions(srdp, new_scdp);
15296           sfmmu_add_scd(&srdp->srd_scdp, new_scdp);
15297           SFMMU_STAT_ADD(sf_create_scd, 1);

15299           mutex_exit(&srdp->srd_scd_mutex);
15300           sfmmu_join_scd(new_scdp, sfmmup);
15301           ASSERT(new_scdp->scd_refcnt >= 2);
15302           atomic_dec_32((volatile uint32_t *)&new_scdp->scd_refcnt);
15304           atomic_add_32((volatile uint32_t *)&new_scdp->scd_refcnt, -1);
15303   }
```

*unchanged_portion_omitted_*

```
**********************************************************
   87074 Mon Jul 28 07:45:00 2014
new/usr/src/uts/sfmmu/vm/hat_sfmmu.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 361 #define SF_SCD_INCR_REF(scdp) {                                         \
 362         atomic_inc_32((volatile uint32_t *)&(scdp)->scd_refcnt);        \
 362         atomic_add_32((volatile uint32_t *)&(scdp)->scd_refcnt, 1);     \
 363 }

 365 #define SF_SCD_DECR_REF(srdp, scdp) {                                   \
 366         sf_region_map_t _scd_rmap = (scdp)->scd_region_map;             \
 367         if (!atomic_dec_32_nv((volatile uint32_t *)&(scdp)->scd_refcnt)) {\
 367         if (!atomic_add_32_nv(                                          \
 368             (volatile uint32_t *)&(scdp)->scd_refcnt, -1)) {            \
 368                 sfmmu_destroy_scd((srdp), (scdp), &_scd_rmap);          \
 369         }                                                               \
 370 }
______unchanged_portion_omitted_
```

```
**********************************************************
   44066 Mon Jul 28 07:45:01 2014
new/usr/src/uts/sparc/dtrace/fasttrap_isa.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/fasttrap_isa.h>
#include <sys/fasttrap_impl.h>
#include <sys/dtrace.h>
#include <sys/dtrace_impl.h>
#include <sys/cmn_err.h>
#include <sys/frame.h>
#include <sys/stack.h>
#include <sys/sysmacros.h>
#include <sys/trap.h>

#include <v9/sys/machpcb.h>
#include <v9/sys/privregs.h>

/*
 * Lossless User-Land Tracing on SPARC
 * -----------------------------------
 *
 * The Basic Idea
 *
 * The most important design constraint is, of course, correct execution of
 * the user thread above all else. The next most important goal is rapid
 * execution. We combine execution of instructions in user-land with
 * emulation of certain instructions in the kernel to aim for complete
 * correctness and maximal performance.
 *
 * We take advantage of the split PC/NPC architecture to speed up logical
 * single-stepping; when we copy an instruction out to the scratch space in
 * the ulwp_t structure (held in the %g7 register on SPARC), we can
 * effectively single step by setting the PC to our scratch space and leaving
 * the NPC alone. This executes the replaced instruction and then continues
 * on without having to reenter the kernel as with single- stepping. The
 * obvious caveat is for instructions whose execution is PC dependant --
 * branches, call and link instructions (call and jmpl), and the rdpc
```



```
 * instruction. These instructions cannot be executed in the manner described
 * so they must be emulated in the kernel.
 *
 * Emulation for this small set of instructions if fairly simple; the most
 * difficult part being emulating branch conditions.
 *
 *
 * A Cache Heavy Portfolio
 *
 * It's important to note at this time that copying an instruction out to the
 * ulwp_t scratch space in user-land is rather complicated. SPARC has
 * separate data and instruction caches so any writes to the D$ (using a
 * store instruction for example) aren't necessarily reflected in the I$.
 * The flush instruction can be used to synchronize the two and must be used
 * for any self-modifying code, but the flush instruction only applies to the
 * primary address space (the absence of a flusha analogue to the flush
 * instruction that accepts an ASI argument is an obvious omission from SPARC
 * v9 where the notion of the alternate address space was introduced on
 * SPARC). To correctly copy out the instruction we must use a block store
 * that doesn't allocate in the D$ and ensures synchronization with the I$;
 * see dtrace_blksuword32() for the implementation  (this function uses
 * ASI_BLK_COMMIT_S to write a block through the secondary ASI in the manner
 * described). Refer to the UltraSPARC I/II manual for details on the
 * ASI_BLK_COMMIT_S ASI.
 *
 *
 * Return Subtleties
 *
 * When we're firing a return probe we need to expose the value returned by
 * the function being traced. Since the function can set the return value
 * in its last instruction, we need to fire the return probe only _after_
 * the effects of the instruction are apparent. For instructions that we
 * emulate, we can call dtrace_probe() after we've performed the emulation;
 * for instructions that we execute after we return to user-land, we set
 * %pc to the instruction we copied out (as described above) and set %npc
 * to a trap instruction stashed in the ulwp_t structure. After the traced
 * instruction is executed, the trap instruction returns control to the
 * kernel where we can fire the return probe.
 *
 * This need for a second trap in cases where we execute the traced
 * instruction makes it all the more important to emulate the most common
 * instructions to avoid the second trip in and out of the kernel.
 *
 *
 * Making it Fast
 *
 * Since copying out an instruction is neither simple nor inexpensive for the
 * CPU, we should attempt to avoid doing it in as many cases as possible.
 * Since function entry and return are usually the most interesting probe
 * sites, we attempt to tune the performance of the fasttrap provider around
 * instructions typically in those places.
 *
 * Looking at a bunch of functions in libraries and executables reveals that
 * most functions begin with either a save or a sethi (to setup a larger
 * argument to the save) and end with a restore or an or (in the case of leaf
 * functions). To try to improve performance, we emulate all of these
 * instructions in the kernel.
 *
 * The save and restore instructions are a little tricky since they perform
 * register window maniplulation. Rather than trying to tinker with the
 * register windows from the kernel, we emulate the implicit add that takes
 * place as part of those instructions and set the %pc to point to a simple
 * save or restore we've hidden in the ulwp_t structure. If we're in a return
 * probe so want to make it seem as though the tracepoint has been completely
 * executed we need to remember that we've pulled this trick with restore and
 * pull registers from the previous window (the one that we'll switch to once
```

```
126  * the simple store instruction is executed) rather than the current one. This
127  * is why in the case of emulating a restore we set the DTrace CPU flag
128  * CPU_DTRACE_FAKERESTORE before calling dtrace_probe() for the return probes
129  * (see fasttrap_return_common()).
130  */

132 #define OP(x)           ((x) >> 30)
133 #define OP2(x)          (((x) >> 22) & 0x07)
134 #define OP3(x)          (((x) >> 19) & 0x3f)
135 #define RCOND(x)        (((x) >> 25) & 0x07)
136 #define COND(x)         (((x) >> 25) & 0x0f)
137 #define A(x)            (((x) >> 29) & 0x01)
138 #define I(x)            (((x) >> 13) & 0x01)
139 #define RD(x)           (((x) >> 25) & 0x1f)
140 #define RS1(x)          (((x) >> 14) & 0x1f)
141 #define RS2(x)          (((x) >> 0) & 0x1f)
142 #define CC(x)           (((x) >> 20) & 0x03)
143 #define DISP16(x)       ((((x) >> 6) & 0xc000) | ((x) & 0x3fff))
144 #define DISP22(x)       ((x) & 0x3fffff)
145 #define DISP19(x)       ((x) & 0x7ffff)
146 #define DISP30(x)       ((x) & 0x3fffffff)
147 #define SW_TRAP(x)      ((x) & 0x7f)

149 #define OP3_OR          0x02
150 #define OP3_RD          0x28
151 #define OP3_JMPL        0x38
152 #define OP3_RETURN      0x39
153 #define OP3_TCC         0x3a
154 #define OP3_SAVE        0x3c
155 #define OP3_RESTORE     0x3d

157 #define OP3_PREFETCH    0x2d
158 #define OP3_CASA        0x3c
159 #define OP3_PREFETCHA   0x3d
160 #define OP3_CASXA       0x3e

162 #define OP2_ILLTRAP     0x0
163 #define OP2_BPcc        0x1
164 #define OP2_Bicc        0x2
165 #define OP2_BPr         0x3
166 #define OP2_SETHI       0x4
167 #define OP2_FBPfcc      0x5
168 #define OP2_FBfcc       0x6

170 #define R_G0            0
171 #define R_O0            8
172 #define R_SP            14
173 #define R_I0            24
174 #define R_I1            25
175 #define R_I2            26
176 #define R_I3            27
177 #define R_I4            28

179 /*
180  * Check the comment in fasttrap.h when changing these offsets or adding
181  * new instructions.
182  */
183 #define FASTTRAP_OFF_SAVE       64
184 #define FASTTRAP_OFF_RESTORE    68
185 #define FASTTRAP_OFF_FTRET      72
186 #define FASTTRAP_OFF_RETURN     76

188 #define BREAKPOINT_INSTR        0x91d02001      /* ta 1 */

190 /*
191  * Tunable to let users turn off the fancy save instruction optimization.
```



```
192  * If a program is non-ABI compliant, there's a possibility that the save
193  * instruction optimization could cause an error.
194  */
195 int fasttrap_optimize_save = 1;

197 static uint64_t
198 fasttrap_anarg(struct regs *rp, int argno)
199 {
200         uint64_t value;

202         if (argno < 6)
203                 return ((&rp->r_o0)[argno]);

205         if (curproc->p_model == DATAMODEL_NATIVE) {
206                 struct frame *fr = (struct frame *)(rp->r_sp + STACK_BIAS);

208                 DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
209                 value = dtrace_fulword(&fr->fr_argd[argno]);
210                 DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT | CPU_DTRACE_BADADDR |
211                     CPU_DTRACE_BADALIGN);
212         } else {
213                 struct frame32 *fr = (struct frame32 *)rp->r_sp;

215                 DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
216                 value = dtrace_fuword32(&fr->fr_argd[argno]);
217                 DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT | CPU_DTRACE_BADADDR |
218                     CPU_DTRACE_BADALIGN);
219         }

221         return (value);
222 }
_____unchanged_portion_omitted_

1379 static uint64_t fasttrap_getreg_fast_cnt;
1380 static uint64_t fasttrap_getreg_mpcb_cnt;
1381 static uint64_t fasttrap_getreg_slow_cnt;

1383 static ulong_t
1384 fasttrap_getreg(struct regs *rp, uint_t reg)
1385 {
1386         ulong_t value;
1387         dtrace_icookie_t cookie;
1388         struct machpcb *mpcb;
1389         extern ulong_t dtrace_getreg_win(uint_t, uint_t);

1391         /*
1392          * We have the %os and %gs in our struct regs, but if we need to
1393          * snag a %l or %i we need to go scrounging around in the process's
1394          * address space.
1395          */
1396         if (reg == 0)
1397                 return (0);

1399         if (reg < 16)
1400                 return ((&rp->r_g1)[reg - 1]);

1402         /*
1403          * Before we look at the user's stack, we'll check the register
1404          * windows to see if the information we want is in there.
1405          */
1406         cookie = dtrace_interrupt_disable();
1407         if (dtrace_getotherwin() > 0) {
1408                 value = dtrace_getreg_win(reg, 1);
1409                 dtrace_interrupt_enable(cookie);

1411                 atomic_inc_64(&fasttrap_getreg_fast_cnt);
```

```
1413                *atomic_add_64(&fasttrap_getreg_fast_cnt, 1);*

1413                return (value);
1414        }
1415        dtrace_interrupt_enable(cookie);

1417        /*
1418         * First check the machpcb structure to see if we've already read
1419         * in the register window we're looking for; if we haven't, (and
1420         * we probably haven't) try to copy in the value of the register.
1421         */
1422        /* LINTED - alignment */
1423        mpcb = (struct machpcb *)((caddr_t)rp - REGOFF);

1425        if (get_udatamodel() == DATAMODEL_NATIVE) {
1426                struct frame *fr = (struct frame *)(rp->r_sp + STACK_BIAS);

1428                if (mpcb->mpcb_wbcnt > 0) {
1429                        struct rwindow *rwin = (void *)mpcb->mpcb_wbuf;
1430                        int i = mpcb->mpcb_wbcnt;
1431                        do {
1432                                i--;
1433                                if ((long)mpcb->mpcb_spbuf[i] != rp->r_sp)
1434                                        continue;

1436                                **atomic_inc_64(&fasttrap_getreg_mpcb_cnt);**
1438                                *atomic_add_64(&fasttrap_getreg_mpcb_cnt, 1);*
1437                                return (rwin[i].rw_local[reg - 16]);
1438                        } while (i > 0);
1439                }

1441                if (fasttrap_fulword(&fr->fr_local[reg - 16], &value) != 0)
1442                        goto err;
1443        } else {
1444                struct frame32 *fr =
1445                    (struct frame32 *)(uintptr_t)(caddr32_t)rp->r_sp;
1446                uint32_t *v32 = (uint32_t *)&value;

1448                if (mpcb->mpcb_wbcnt > 0) {
1449                        struct rwindow32 *rwin = (void *)mpcb->mpcb_wbuf;
1450                        int i = mpcb->mpcb_wbcnt;
1451                        do {
1452                                i--;
1453                                if ((long)mpcb->mpcb_spbuf[i] != rp->r_sp)
1454                                        continue;

1456                                **atomic_inc_64(&fasttrap_getreg_mpcb_cnt);**
1458                                *atomic_add_64(&fasttrap_getreg_mpcb_cnt, 1);*
1457                                return (rwin[i].rw_local[reg - 16]);
1458                        } while (i > 0);
1459                }

1461                if (fasttrap_fuword32(&fr->fr_local[reg - 16], &v32[1]) != 0)
1462                        goto err;

1464                v32[0] = 0;
1465        }

1467        **atomic_inc_64(&fasttrap_getreg_slow_cnt);**
1469        *atomic_add_64(&fasttrap_getreg_slow_cnt, 1);*
1468        return (value);

1470 err:
1471        /*
1472         * If the copy in failed, the process will be in a irrecoverable
1473         * state, and we have no choice but to kill it.
```



```
1474         */
1475        psignal(ttoproc(curthread), SIGILL);
1476        return (0);
1477 }

1479 static uint64_t fasttrap_putreg_fast_cnt;
1480 static uint64_t fasttrap_putreg_mpcb_cnt;
1481 static uint64_t fasttrap_putreg_slow_cnt;

1483 static void
1484 fasttrap_putreg(struct regs *rp, uint_t reg, ulong_t value)
1485 {
1486        dtrace_icookie_t cookie;
1487        struct machpcb *mpcb;
1488        extern void dtrace_putreg_win(uint_t, ulong_t);

1490        if (reg == 0)
1491                return;

1493        if (reg < 16) {
1494                (&rp->r_g1)[reg - 1] = value;
1495                return;
1496        }

1498        /*
1499         * If the user process is still using some register windows, we
1500         * can just place the value in the correct window.
1501         */
1502        cookie = dtrace_interrupt_disable();
1503        if (dtrace_getotherwin() > 0) {
1504                dtrace_putreg_win(reg, value);
1505                dtrace_interrupt_enable(cookie);
1506                **atomic_inc_64(&fasttrap_putreg_fast_cnt);**
1508                *atomic_add_64(&fasttrap_putreg_fast_cnt, 1);*
1507                return;
1508        }
1509        dtrace_interrupt_enable(cookie);

1511        /*
1512         * First see if there's a copy of the register window in the
1513         * machpcb structure that we can modify; if there isn't try to
1514         * copy out the value. If that fails, we try to create a new
1515         * register window in the machpcb structure. While this isn't
1516         * _precisely_ the intended use of the machpcb structure, it
1517         * can't cause any problems since we know at this point in the
1518         * code that all of the user's data have been flushed out of the
1519         * register file (since %otherwin is 0).
1520         */
1521        /* LINTED - alignment */
1522        mpcb = (struct machpcb *)((caddr_t)rp - REGOFF);

1524        if (get_udatamodel() == DATAMODEL_NATIVE) {
1525                struct frame *fr = (struct frame *)(rp->r_sp + STACK_BIAS);
1526                /* LINTED - alignment */
1527                struct rwindow *rwin = (struct rwindow *)mpcb->mpcb_wbuf;

1529                if (mpcb->mpcb_wbcnt > 0) {
1530                        int i = mpcb->mpcb_wbcnt;
1531                        do {
1532                                i--;
1533                                if ((long)mpcb->mpcb_spbuf[i] != rp->r_sp)
1534                                        continue;

1536                                rwin[i].rw_local[reg - 16] = value;
1537                                **atomic_inc_64(&fasttrap_putreg_mpcb_cnt);**
1539                                *atomic_add_64(&fasttrap_putreg_mpcb_cnt, 1);*
```

```
1538                                return;
1539                        } while (i > 0);
1540                }

1542                if (fasttrap_sulword(&fr->fr_local[reg - 16], value) != 0) {
1543                        if (mpcb->mpcb_wbcnt >= MAXWIN || copyin(fr,
1544                            &rwin[mpcb->mpcb_wbcnt], sizeof (*rwin)) != 0)
1545                                goto err;

1547                        rwin[mpcb->mpcb_wbcnt].rw_local[reg - 16] = value;
1548                        mpcb->mpcb_spbuf[mpcb->mpcb_wbcnt] = (caddr_t)rp->r_sp;
1549                        mpcb->mpcb_wbcnt++;
1550                        atomic_inc_64(&fasttrap_putreg_mpcb_cnt);
1552                        atomic_add_64(&fasttrap_putreg_mpcb_cnt, 1);
1551                        return;
1552                }
1553        } else {
1554                struct frame32 *fr =
1555                    (struct frame32 *)(uintptr_t)(caddr32_t)rp->r_sp;
1556                /* LINTED - alignment */
1557                struct rwindow32 *rwin = (struct rwindow32 *)mpcb->mpcb_wbuf;
1558                uint32_t v32 = (uint32_t)value;

1560                if (mpcb->mpcb_wbcnt > 0) {
1561                        int i = mpcb->mpcb_wbcnt;
1562                        do {
1563                                i--;
1564                                if ((long)mpcb->mpcb_spbuf[i] != rp->r_sp)
1565                                        continue;

1567                                rwin[i].rw_local[reg - 16] = v32;
1568                                atomic_inc_64(&fasttrap_putreg_mpcb_cnt);
1570                                atomic_add_64(&fasttrap_putreg_mpcb_cnt, 1);
1569                                return;
1570                        } while (i > 0);
1571                }

1573                if (fasttrap_suword32(&fr->fr_local[reg - 16], v32) != 0) {
1574                        if (mpcb->mpcb_wbcnt >= MAXWIN || copyin(fr,
1575                            &rwin[mpcb->mpcb_wbcnt], sizeof (*rwin)) != 0)
1576                                goto err;

1578                        rwin[mpcb->mpcb_wbcnt].rw_local[reg - 16] = v32;
1579                        mpcb->mpcb_spbuf[mpcb->mpcb_wbcnt] = (caddr_t)rp->r_sp;
1580                        mpcb->mpcb_wbcnt++;
1581                        atomic_inc_64(&fasttrap_putreg_mpcb_cnt);
1583                        atomic_add_64(&fasttrap_putreg_mpcb_cnt, 1);
1582                        return;
1583                }
1584        }

1586        atomic_inc_64(&fasttrap_putreg_slow_cnt);
1588        atomic_add_64(&fasttrap_putreg_slow_cnt, 1);
1587        return;

1589 err:
1590        /*
1591         * If we couldn't record this register's value, the process is in an
1592         * irrecoverable state and we have no choice but to euthanize it.
1593         */
1594        psignal(ttoproc(curthread), SIGILL);
1595 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   25455 Mon Jul 28 07:45:01 2014
new/usr/src/uts/sparc/fpu/fpu_simulator.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
  23  * Use is subject to license terms.
  24  */

  26 /* Main procedures for sparc FPU simulator. */

  28 #include <sys/fpu/fpu_simulator.h>
  29 #include <sys/fpu/globals.h>
  30 #include <sys/fpu/fpusystm.h>
  31 #include <sys/proc.h>
  32 #include <sys/signal.h>
  33 #include <sys/siginfo.h>
  34 #include <sys/thread.h>
  35 #include <sys/cpuvar.h>
  36 #include <sys/cmn_err.h>
  37 #include <sys/atomic.h>
  38 #include <sys/privregs.h>
  39 #include <sys/vis_simulator.h>

  41 #define FPUINFO_KSTAT(opcode)   {                                       \
  42         extern void __dtrace_probe___fpuinfo_##opcode(uint64_t *);      \
  43         uint64_t *stataddr = &fpuinfo.opcode.value.ui64;                \
  44         __dtrace_probe___fpuinfo_##opcode(stataddr);                    \
  45         atomic_inc_64(&fpuinfo.opcode.value.ui64);                      \
  45         atomic_add_64(&fpuinfo.opcode.value.ui64, 1);                   \
  46 }
______unchanged_portion_omitted_

 788 void
 789 fp_kstat_update(enum ftt_type ftt)
 790 {
 791         ASSERT((ftt == ftt_ieee) || (ftt == ftt_unfinished) ||
 792             (ftt == ftt_unimplemented));
 793         if (ftt == ftt_ieee)
 794                 atomic_inc_64(&fpustat.fpu_ieee_traps.value.ui64);
 794                 atomic_add_64(&fpustat.fpu_ieee_traps.value.ui64, 1);
 795         else if (ftt == ftt_unfinished)
 796                 atomic_inc_64(&fpustat.fpu_unfinished_traps.value.ui64);
 796                 atomic_add_64(&fpustat.fpu_unfinished_traps.value.ui64, 1);
 797         else if (ftt == ftt_unimplemented)
```



```
 798                 atomic_inc_64(&fpustat.fpu_unimplemented_traps.value.ui64);
 798                 atomic_add_64(&fpustat.fpu_unimplemented_traps.value.ui64, 1);
 799 }
______unchanged_portion_omitted_
```

****
15100 Mon Jul 28 07:45:01 2014
new/usr/src/uts/sparc/sys/fpu/fpu_simulator.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
 378 #define VISINFO_KSTAT(opcode)   {                                       \
 379         extern void __dtrace_probe___visinfo_##opcode(uint64_t *);      \
 380         uint64_t *stataddr = &visinfo.opcode.value.ui64;                \
 381         __dtrace_probe___visinfo_##opcode(stataddr);                    \
 382         atomic_inc_64(&visinfo.opcode.value.ui64);                      \
 382         atomic_add_64(&visinfo.opcode.value.ui64, 1);                   \
 383 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    8236 Mon Jul 28 07:45:01 2014
new/usr/src/uts/sun4/os/memnode.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/systm.h>
#include <sys/platform_module.h>
#include <sys/sysmacros.h>
#include <sys/atomic.h>
#include <sys/memlist.h>
#include <sys/memnode.h>
#include <vm/vm_dep.h>

int max_mem_nodes = 1;          /* max memory nodes on this system */

struct mem_node_conf mem_node_config[MAX_MEM_NODES];
int mem_node_pfn_shift;
/*
 * num_memnodes should be updated atomically and always >=
 * the number of bits in memnodes_mask or the algorithm may fail.
 */
uint16_t num_memnodes;
mnodeset_t memnodes_mask; /* assumes 8*(sizeof(mnodeset_t)) >= MAX_MEM_NODES */

/*
 * If set, mem_node_physalign should be a power of two, and
 * should reflect the minimum address alignment of each node.
 */
uint64_t mem_node_physalign;

/*
 * Platform hooks we will need.
 */

#pragma weak plat_build_mem_nodes
#pragma weak plat_slice_add
#pragma weak plat_slice_del

/*
 * Adjust the memnode config after a DR operation.
 *
```



```
 * It is rather tricky to do these updates since we can't
 * protect the memnode structures with locks, so we must
 * be mindful of the order in which updates and reads to
 * these values can occur.
 */
void
mem_node_add_slice(pfn_t start, pfn_t end)
{
        int mnode;
        mnodeset_t newmask, oldmask;

        /*
         * DR will pass us the first pfn that is allocatable.
         * We need to round down to get the real start of
         * the slice.
         */
        if (mem_node_physalign) {
                start &= ~(btop(mem_node_physalign) - 1);
                end = roundup(end, btop(mem_node_physalign)) - 1;
        }

        mnode = PFN_2_MEM_NODE(start);
        ASSERT(mnode < max_mem_nodes);

        if (atomic_cas_32((uint32_t *)&mem_node_config[mnode].exists, 0, 1)) {
                /*
                 * Add slice to existing node.
                 */
                if (start < mem_node_config[mnode].physbase)
                        mem_node_config[mnode].physbase = start;
                if (end > mem_node_config[mnode].physmax)
                        mem_node_config[mnode].physmax = end;
        } else {
                mem_node_config[mnode].physbase = start;
                mem_node_config[mnode].physmax = end;
                atomic_inc_16(&num_memnodes);
                atomic_add_16(&num_memnodes, 1);
                do {
                        oldmask = memnodes_mask;
                        newmask = memnodes_mask | (1ull << mnode);
                } while (atomic_cas_64(&memnodes_mask, oldmask, newmask) !=
                         oldmask);
        }
        /*
         * Let the common lgrp framework know about the new memory
         */
        lgrp_config(LGRP_CONFIG_MEM_ADD, mnode, MEM_NODE_2_LGRPHAND(mnode));
}

/*
 * Remove a PFN range from a memnode.  On some platforms,
 * the memnode will be created with physbase at the first
 * allocatable PFN, but later deleted with the MC slice
 * base address converted to a PFN, in which case we need
 * to assume physbase and up.
 */
void
mem_node_del_slice(pfn_t start, pfn_t end)
{
        int mnode;
        pgcnt_t delta_pgcnt, node_size;
        mnodeset_t omask, nmask;

        if (mem_node_physalign) {
                start &= ~(btop(mem_node_physalign) - 1);
                end = roundup(end, btop(mem_node_physalign)) - 1;
```

```
127         }
128         mnode = PFN_2_MEM_NODE(start);

130         ASSERT(mnode < max_mem_nodes);
131         ASSERT(mem_node_config[mnode].exists == 1);

133         delta_pgcnt = end - start;
134         node_size = mem_node_config[mnode].physmax -
135             mem_node_config[mnode].physbase;

137         if (node_size > delta_pgcnt) {
138                 /*
139                  * Subtract the slice from the memnode.
140                  */
141                 if (start <= mem_node_config[mnode].physbase)
142                         mem_node_config[mnode].physbase = end + 1;
143                 ASSERT(end <= mem_node_config[mnode].physmax);
144                 if (end == mem_node_config[mnode].physmax)
145                         mem_node_config[mnode].physmax = start - 1;
146         } else {

148                 /*
149                  * Let the common lgrp framework know the mnode is
150                  * leaving
151                  */
152                 lgrp_config(LGRP_CONFIG_MEM_DEL, mnode,
153                     MEM_NODE_2_LGRPHAND(mnode));

155                 /*
156                  * Delete the whole node.
157                  */
158                 ASSERT(MNODE_PGCNT(mnode) == 0);
159                 do {
160                         omask = memnodes_mask;
161                         nmask = omask & ~(1ull << mnode);
162                 } while (atomic_cas_64(&memnodes_mask, omask, nmask) != omask);
163                 atomic_dec_16(&num_memnodes);
164                 mem_node_config[mnode].exists = 0;
165         }
166 }
```

_____***unchanged_portion_omitted_***

```
210 /*
211  * Allocate an unassigned memnode.
212  */
213 int
214 mem_node_alloc()
215 {
216         int mnode;
217         mnodeset_t newmask, oldmask;

219         /*
220          * Find an unused memnode.  Update it atomically to prevent
221          * a first time memnode creation race.
222          */
223         for (mnode = 0; mnode < max_mem_nodes; mnode++)
224                 if (atomic_cas_32((uint32_t *)&mem_node_config[mnode].exists,
225                     0, 1) == 0)
226                         break;

228         if (mnode >= max_mem_nodes)
229                         panic("Out of free memnodes\n");

231         mem_node_config[mnode].physbase = (uint64_t)-1;
232         mem_node_config[mnode].physmax = 0;
```



```
233         atomic_inc_16(&num_memnodes);
233         atomic_add_16(&num_memnodes, 1);
234         do {
235                 oldmask = memnodes_mask;
236                 newmask = memnodes_mask | (1ull << mnode);
237         } while (atomic_cas_64(&memnodes_mask, oldmask, newmask) != oldmask);

239         return (mnode);
240 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   16801 Mon Jul 28 07:45:01 2014
new/usr/src/uts/sun4/os/prom_subr.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

 189 /*
 190  * PROM Locking Primitives
 191  *
 192  * These routines are called immediately before and immediately after calling
 193  * into the firmware.  The firmware is single-threaded and assumes that the
 194  * kernel will implement locking to prevent simultaneous service calls.  In
 195  * addition, some service calls (particularly character rendering) can be
 196  * slow, so we would like to sleep if we cannot acquire the lock to allow the
 197  * caller's CPU to continue to perform useful work in the interim.  Service
 198  * routines may also be called early in boot as part of slave CPU startup
 199  * when mutexes and cvs are not yet available (i.e. they are still running on
 200  * the prom's TLB handlers and cannot touch curthread).  Therefore, these
 201  * routines must reduce to a simple compare-and-swap spin lock when necessary.
 202  * Finally, kernel code may wish to acquire the firmware lock before executing
 203  * a block of code that includes service calls, so we also allow the firmware
 204  * lock to be acquired recursively by the owning CPU after disabling preemption.
 205  *
 206  * To meet these constraints, the lock itself is implemented as a compare-and-
 207  * swap spin lock on the global prom_cpu pointer.  We implement recursion by
 208  * atomically incrementing the integer prom_holdcnt after acquiring the lock.
 209  * If the current CPU is an "adult" (determined by testing cpu_m.mutex_ready),
 210  * we disable preemption before acquiring the lock and leave it disabled once
 211  * the lock is held.  The kern_postprom() routine then enables preemption if
 212  * we drop the lock and prom_holdcnt returns to zero.  If the current CPU is
 213  * an adult and the lock is held by another adult CPU, we can safely sleep
 214  * until the lock is released.  To do so, we acquire the adaptive prom_mutex
 215  * and then sleep on prom_cv.  Therefore, service routines must not be called
 216  * from above LOCK_LEVEL on any adult CPU.  Finally, if recursive entry is
 217  * attempted on an adult CPU, we must also verify that curthread matches the
 218  * saved prom_thread (the original owner) to ensure that low-level interrupt
 219  * threads do not step on other threads running on the same CPU.
 220  */

 222 static cpu_t *volatile prom_cpu;
 223 static kthread_t *volatile prom_thread;
 224 static uint32_t prom_holdcnt;
 225 static kmutex_t prom_mutex;
 226 static kcondvar_t prom_cv;

 228 /*
 229  * The debugger uses PROM services, and is thus unable to run if any of the
 230  * CPUs on the system are executing in the PROM at the time of debugger entry.
 231  * If a CPU is determined to be in the PROM when the debugger is entered,
 232  * prom_return_enter_debugger will be set, thus triggering a programmed debugger
 233  * entry when the given CPU returns from the PROM.  That CPU is then released by
 234  * the debugger, and is allowed to complete PROM-related work.
 235  */
 236 int prom_exit_enter_debugger;

 238 void
 239 kern_preprom(void)
 240 {
 241         for (;;) {
 242                 /*
 243                  * Load the current CPU pointer and examine the mutex_ready bit.
 244                  * It doesn't matter if we are preempted here because we are
 245                  * only trying to determine if we are in the *set* of mutex
 246                  * ready CPUs.  We cannot disable preemption until we confirm
 247                  * that we are running on a CPU in this set, since a call to
```



```
 248                  * kpreempt_disable() requires access to curthread.
 249                  */
 250                 processorid_t cpuid = getprocessorid();
 251                 cpu_t *cp = cpu[cpuid];
 252                 cpu_t *prcp;

 254                 if (panicstr)
 255                         return; /* just return if we are currently panicking */

 257                 if (CPU_IN_SET(cpu_ready_set, cpuid) && cp->cpu_m.mutex_ready) {
 258                         /*
 259                          * Disable premption, and reload the current CPU.  We
 260                          * can't move from a mutex_ready cpu to a non-ready cpu
 261                          * so we don't need to re-check cp->cpu_m.mutex_ready.
 262                          */
 263                         kpreempt_disable();
 264                         cp = CPU;
 265                         ASSERT(cp->cpu_m.mutex_ready);

 267                         /*
 268                          * Try the lock.  If we don't get the lock, re-enable
 269                          * preemption and see if we should sleep.  If we are
 270                          * already the lock holder, remove the effect of the
 271                          * previous kpreempt_disable() before returning since
 272                          * preemption was disabled by an earlier kern_preprom.
 273                          */
 274                         prcp = atomic_cas_ptr((void *)&prom_cpu, NULL, cp);
 275                         if (prcp == NULL ||
 276                             (prcp == cp && prom_thread == curthread)) {
 277                                 if (prcp == cp)
 278                                         kpreempt_enable();
 279                                 break;
 280                         }

 282                         kpreempt_enable();

 284                         /*
 285                          * We have to be very careful here since both prom_cpu
 286                          * and prcp->cpu_m.mutex_ready can be changed at any
 287                          * time by a non mutex_ready cpu holding the lock.
 288                          * If the owner is mutex_ready, holding prom_mutex
 289                          * prevents kern_postprom() from completing.  If the
 290                          * owner isn't mutex_ready, we only know it will clear
 291                          * prom_cpu before changing cpu_m.mutex_ready, so we
 292                          * issue a membar after checking mutex_ready and then
 293                          * re-verify that prom_cpu is still held by the same
 294                          * cpu before actually proceeding to cv_wait().
 295                          */
 296                         mutex_enter(&prom_mutex);
 297                         prcp = prom_cpu;
 298                         if (prcp != NULL && prcp->cpu_m.mutex_ready != 0) {
 299                                 membar_consumer();
 300                                 if (prcp == prom_cpu)
 301                                         cv_wait(&prom_cv, &prom_mutex);
 302                         }
 303                         mutex_exit(&prom_mutex);

 305                 } else {
 306                         /*
 307                          * If we are not yet mutex_ready, just attempt to grab
 308                          * the lock.  If we get it or already hold it, break.
 309                          */
 310                         ASSERT(getpil() == PIL_MAX);
 311                         prcp = atomic_cas_ptr((void *)&prom_cpu, NULL, cp);
 312                         if (prcp == NULL || prcp == cp)
 313                                 break;
```

```
314                 }
315         }

317         /*
318          * We now hold the prom_cpu lock.  Increment the hold count by one
319          * and assert our current state before returning to the caller.
320          */
321         atomic_inc_32(&prom_holdcnt);
321         atomic_add_32(&prom_holdcnt, 1);
322         ASSERT(prom_holdcnt >= 1);
323         prom_thread = curthread;
324 }

326 /*
327  * Drop the prom lock if it is held by the current CPU.  If the lock is held
328  * recursively, return without clearing prom_cpu.  If the hold count is now
329  * zero, clear prom_cpu and cv_signal any waiting CPU.
330  */
331 void
332 kern_postprom(void)
333 {
334         processorid_t cpuid = getprocessorid();
335         cpu_t *cp = cpu[cpuid];

337         if (panicstr)
338                 return; /* do not modify lock further if we have panicked */

340         if (prom_cpu != cp)
341                 panic("kern_postprom: not owner, cp=%p owner=%p",
342                     (void *)cp, (void *)prom_cpu);

344         if (prom_holdcnt == 0)
345                 panic("kern_postprom: prom_holdcnt == 0, owner=%p",
346                     (void *)prom_cpu);

348         if (atomic_dec_32_nv(&prom_holdcnt) != 0)
348         if (atomic_add_32_nv(&prom_holdcnt, -1) != 0)
349                 return; /* prom lock is held recursively by this CPU */

351         if ((boothowto & RB_DEBUG) && prom_exit_enter_debugger)
352                 kmdb_enter();

354         prom_thread = NULL;
355         membar_producer();

357         prom_cpu = NULL;
358         membar_producer();

360         if (CPU_IN_SET(cpu_ready_set, cpuid) && cp->cpu_m.mutex_ready) {
361                 mutex_enter(&prom_mutex);
362                 cv_signal(&prom_cv);
363                 mutex_exit(&prom_mutex);
364                 kpreempt_enable();
365         }
366 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  129904 Mon Jul 28 07:45:02 2014
new/usr/src/uts/sun4u/cpu/spitfire.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

4231 /*ARGSUSED*/
4232 static void
4233 leaky_bucket_timeout(void *arg)
4234 {
4235         int i;
4236         struct ce_info *psimm = mem_ce_simm;

4238         for (i = 0; i < mem_ce_simm_size; i++) {
4239                 if (psimm[i].leaky_bucket_cnt > 0)
4240                         atomic_dec_16(&psimm[i].leaky_bucket_cnt);
4240                         atomic_add_16(&psimm[i].leaky_bucket_cnt, -1);
4241         }
4242         add_leaky_bucket_timeout();
4243 }
______unchanged_portion_omitted_

4278 /*
4279  * Legacy Correctable ECC Error Hash
4280  *
4281  * All of the code below this comment is used to implement a legacy array
4282  * which counted intermittent, persistent, and sticky CE errors by unum,
4283  * and then was later extended to publish the data as a kstat for SunVTS.
4284  * All of this code is replaced by FMA, and remains here until such time
4285  * that the UltraSPARC-I/II CPU code is converted to FMA, or is EOLed.
4286  *
4287  * Errors are saved in three buckets per-unum:
4288  * (1) sticky - scrub was unsuccessful, cannot be scrubbed
4289  *     This could represent a problem, and is immediately printed out.
4290  * (2) persistent - was successfully scrubbed
4291  *     These errors use the leaky bucket algorithm to determine
4292  *     if there is a serious problem.
4293  * (3) intermittent - may have originated from the cpu or upa/safari bus,
4294  *     and does not necessarily indicate any problem with the dimm itself,
4295  *     is critical information for debugging new hardware.
4296  *     Because we do not know if it came from the dimm, it would be
4297  *     inappropriate to include these in the leaky bucket counts.
4298  *
4299  * If the E$ line was modified before the scrub operation began, then the
4300  * displacement flush at the beginning of scrubphys() will cause the modified
4301  * line to be written out, which will clean up the CE.  Then, any subsequent
4302  * read will not cause an error, which will cause persistent errors to be
4303  * identified as intermittent.
4304  *
4305  * If a DIMM is going bad, it will produce true persistents as well as
4306  * false intermittents, so these intermittents can be safely ignored.
4307  *
4308  * If the error count is excessive for a DIMM, this function will return
4309  * PR_MCE, and the CPU module may then decide to remove that page from use.
4310  */
4311 static int
4312 ce_count_unum(int status, int len, char *unum)
4313 {
4314         int i;
4315         struct ce_info *psimm = mem_ce_simm;
4316         int page_status = PR_OK;

4318         ASSERT(psimm != NULL);

4320         if (len <= 0 ||
```



```
4321             (status & (ECC_STICKY | ECC_PERSISTENT | ECC_INTERMITTENT)) == 0)
4322                 return (page_status);

4324         /*
4325          * Initialize the leaky_bucket timeout
4326          */
4327         if (atomic_cas_ptr(&leaky_bucket_timeout_id,
4328             TIMEOUT_NONE, TIMEOUT_SET) == TIMEOUT_NONE)
4329                 add_leaky_bucket_timeout();

4331         for (i = 0; i < mem_ce_simm_size; i++) {
4332                 if (psimm[i].name[0] == '\0') {
4333                         /*
4334                          * Hit the end of the valid entries, add
4335                          * a new one.
4336                          */
4337                         (void) strncpy(psimm[i].name, unum, len);
4338                         if (status & ECC_STICKY) {
4339                                 /*
4340                                  * Sticky - the leaky bucket is used to track
4341                                  * soft errors.  Since a sticky error is a
4342                                  * hard error and likely to be retired soon,
4343                                  * we do not count it in the leaky bucket.
4344                                  */
4345                                 psimm[i].leaky_bucket_cnt = 0;
4346                                 psimm[i].intermittent_total = 0;
4347                                 psimm[i].persistent_total = 0;
4348                                 psimm[i].sticky_total = 1;
4349                                 cmn_err(CE_NOTE,
4350                                     "[AFT0] Sticky Softerror encountered "
4351                                     "on Memory Module %s\n", unum);
4352                                 page_status = PR_MCE;
4353                         } else if (status & ECC_PERSISTENT) {
4354                                 psimm[i].leaky_bucket_cnt = 1;
4355                                 psimm[i].intermittent_total = 0;
4356                                 psimm[i].persistent_total = 1;
4357                                 psimm[i].sticky_total = 0;
4358                         } else {
4359                                 /*
4360                                  * Intermittent - Because the scrub operation
4361                                  * cannot find the error in the DIMM, we will
4362                                  * not count these in the leaky bucket
4363                                  */
4364                                 psimm[i].leaky_bucket_cnt = 0;
4365                                 psimm[i].intermittent_total = 1;
4366                                 psimm[i].persistent_total = 0;
4367                                 psimm[i].sticky_total = 0;
4368                         }
4369                         ecc_error_info_data.count.value.ui32++;
4370                         break;
4371                 } else if (strncmp(unum, psimm[i].name, len) == 0) {
4372                         /*
4373                          * Found an existing entry for the current
4374                          * memory module, adjust the counts.
4375                          */
4376                         if (status & ECC_STICKY) {
4377                                 psimm[i].sticky_total++;
4378                                 cmn_err(CE_NOTE,
4379                                     "[AFT0] Sticky Softerror encountered "
4380                                     "on Memory Module %s\n", unum);
4381                                 page_status = PR_MCE;
4382                         } else if (status & ECC_PERSISTENT) {
4383                                 int new_value;

4385                                 new_value = atomic_inc_16_nv(
4386                                     &psimm[i].leaky_bucket_cnt);
```

```
4385                                *new_value = atomic_add_16_nv(
4386                                    &psimm[i].leaky_bucket_cnt, 1);
4387                                psimm[i].persistent_total++;
4388                                if (new_value > ecc_softerr_limit) {
4389                                        cmn_err(CE_NOTE, "[AFT0] Most recent %d"
4390                                            " soft errors from Memory Module"
4391                                            " %s exceed threshold (N=%d,"
4392                                            " T=%dh:%02dm) triggering page"
4393                                            " retire", new_value, unum,
4394                                            ecc_softerr_limit,
4395                                            ecc_softerr_interval / 60,
4396                                            ecc_softerr_interval % 60);
4397                                        atomic_dec_16(
4398                                            &psimm[i].leaky_bucket_cnt);
4397                                        atomic_add_16(
4398                                            &psimm[i].leaky_bucket_cnt, -1);
4399                                        page_status = PR_MCE;
4400                                }
4401                        } else { /* Intermittent */
4402                                psimm[i].intermittent_total++;
4403                        }
4404                        break;
4405                }
4406        }

4408        if (i >= mem_ce_simm_size)
4409                cmn_err(CE_CONT, "[AFT0] Softerror: mem_ce_simm[] out of "
4410                    "space.\n");

4412        return (page_status);
4413 }
```

_____***unchanged_portion_omitted_***

****
210247 Mon Jul 28 07:45:02 2014
new/usr/src/uts/sun4u/cpu/us3_common.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
****
_____***unchanged_portion_omitted_***

```
5631 /*
5632  * If scrubbing is enabled, increment the outstanding request counter.  If it
5633  * is 1 (meaning there were no previous requests outstanding), call
5634  * setsoftint_tl1 through xt_one_unchecked, which eventually ends up doing
5635  * a self trap.
5636  */
5637 static void
5638 do_scrub(struct scrub_info *csi)
5639 {
5640         ch_scrub_misc_t *csmp = CPU_PRIVATE_PTR(CPU, chpr_scrub_misc);
5641         int index = csi->csi_index;
5642         uint32_t *outstanding = &csmp->chsm_outstanding[index];

5644         if (*(csi->csi_enable) && (csmp->chsm_enable[index])) {
5645                 if (atomic_inc_32_nv(outstanding) == 1) {
5645                 if (atomic_add_32_nv(outstanding, 1) == 1) {
5646                         xt_one_unchecked(CPU->cpu_id, setsoftint_tl1,
5647                             csi->csi_inum, 0);
5648                 }
5649         }
5650 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   37770 Mon Jul 28 07:45:02 2014
new/usr/src/uts/sun4u/os/memscrub.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
______unchanged_portion_omitted_

1404 /*ARGSUSED*/
1405 static void
1406 memscrub_mem_config_post_add(
1407         void *arg,
1408         pgcnt_t delta_pages)
1409 {
1410         /*
1411          * We increment pause_memscrub before entering new_memscrub(). This
1412          * will force the memscrubber to sleep, allowing the DR callback
1413          * thread to acquire memscrub_lock in new_memscrub(). The use of
1414          * atomic_add_32() allows concurrent memory DR operations to use the
1415          * callbacks safely.
1416          */
1417         atomic_inc_32(&pause_memscrub);
1417         atomic_add_32(&pause_memscrub, 1);
1418         ASSERT(pause_memscrub != 0);

1420         /*
1421          * "Don't care" if we are not scrubbing new memory.
1422          */
1423         (void) new_memscrub(0);         /* retain page retire list */

1425         /* Restore the pause setting. */
1426         atomic_dec_32(&pause_memscrub);
1426         atomic_add_32(&pause_memscrub, -1);
1427 }
_____unchanged_portion_omitted_

1439 /*ARGSUSED*/
1440 static void
1441 memscrub_mem_config_post_del(
1442         void *arg,
1443         pgcnt_t delta_pages,
1444         int cancelled)
1445 {
1446         /*
1447          * We increment pause_memscrub before entering new_memscrub(). This
1448          * will force the memscrubber to sleep, allowing the DR callback
1449          * thread to acquire memscrub_lock in new_memscrub(). The use of
1450          * atomic_add_32() allows concurrent memory DR operations to use the
1451          * callbacks safely.
1452          */
1453         atomic_inc_32(&pause_memscrub);
1453         atomic_add_32(&pause_memscrub, 1);
1454         ASSERT(pause_memscrub != 0);

1456         /*
1457          * Must stop scrubbing deleted memory as it may be disconnected.
1458          */
1459         if (new_memscrub(1)) {  /* update page retire list */
1460                 disable_memscrub = 1;
1461         }

1463         /* Restore the pause setting. */
1464         atomic_dec_32(&pause_memscrub);
1464         atomic_add_32(&pause_memscrub, -1);
1465 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   15845 Mon Jul 28 07:45:02 2014
new/usr/src/uts/sun4u/sunfire/io/ac_test.c
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/ddi_impldefs.h>
#include <sys/obpdefs.h>
#include <sys/cmn_err.h>
#include <sys/errno.h>
#include <sys/kmem.h>
#include <sys/vmem.h>
#include <sys/debug.h>
#include <sys/sysmacros.h>
#include <sys/machsystm.h>
#include <sys/machparam.h>
#include <sys/modctl.h>
#include <sys/atomic.h>
#include <sys/fhc.h>
#include <sys/ac.h>
#include <sys/jtag.h>
#include <sys/cpu_module.h>
#include <sys/spitregs.h>
#include <sys/vm.h>
#include <vm/seg_kmem.h>
#include <vm/hat_sfmmu.h>

/* memory setup parameters */
#define TEST_PAGESIZE   MMU_PAGESIZE

struct test_info {
        struct test_info        *next;          /* linked list of tests */
        struct ac_mem_info      *mem_info;
        uint_t                  board;
        uint_t                  bank;
```



```
        caddr_t                 bufp;           /* pointer to buffer page */
        caddr_t                 va;             /* test target VA */
        ac_mem_test_start_t     info;
        uint_t                  in_test;        /* count of threads in test */
};
______unchanged_portion_omitted_

int
ac_mem_test_start(ac_cfga_pkt_t *pkt, int flag)
{
        struct ac_soft_state    *softsp;
        struct ac_mem_info      *mem_info;
        struct bd_list          *board;
        struct test_info        *test;
        uint64_t                decode;

        /* XXX if ac ever detaches... */
        if (test_mutex_initialized == FALSE) {
                mutex_init(&test_mutex, NULL, MUTEX_DEFAULT, NULL);
                test_mutex_initialized = TRUE;
        }

        /*
         * Is the specified bank testable?
         */

        board = fhc_bdlist_lock(pkt->softsp->board);
        if (board == NULL || board->ac_softsp == NULL) {
                fhc_bdlist_unlock();
                AC_ERR_SET(pkt, AC_ERR_BD);
                return (EINVAL);
        }
        ASSERT(pkt->softsp == board->ac_softsp);

        /* verify the board is of the correct type */
        switch (board->sc.type) {
        case CPU_BOARD:
        case MEM_BOARD:
                break;
        default:
                fhc_bdlist_unlock();
                AC_ERR_SET(pkt, AC_ERR_BD_TYPE);
                return (EINVAL);
        }

        /*
         * Memory must be in the spare state to be testable.
         * However, spare memory that is testing can't be tested
         * again, instead return the current test info.
         */
        softsp = pkt->softsp;
        mem_info = &softsp->bank[pkt->bank];
        if (!MEM_BOARD_VISIBLE(board) ||
            fhc_bd_busy(softsp->board) ||
            mem_info->rstate != SYSC_CFGA_RSTATE_CONNECTED ||
            mem_info->ostate != SYSC_CFGA_OSTATE_UNCONFIGURED) {
                fhc_bdlist_unlock();
                AC_ERR_SET(pkt, AC_ERR_BD_STATE);
                return (EINVAL);
        }
        if (mem_info->busy) {   /* oops, testing? */
                /*
                 * find the test entry
                 */
                ASSERT(test_mutex_initialized);
                mutex_enter(&test_mutex);
```

```
153                 for (test = test_base; test != NULL; test = test->next) {
154                         if (test->board == softsp->board &&
155                             test->bank == pkt->bank)
156                                 break;
157                 }
158                 if (test == NULL) {
159                         mutex_exit(&test_mutex);
160                         fhc_bdlist_unlock();
161                         /* Not busy testing. */
162                         AC_ERR_SET(pkt, AC_ERR_BD_STATE);
163                         return (EINVAL);
164                 }

166                 /*
167                  * return the current test information to the new caller
168                  */
169                 if (ddi_copyout(&test->info, pkt->cmd_cfga.private,
170                     sizeof (ac_mem_test_start_t), flag) != 0) {
171                         mutex_exit(&test_mutex);
172                         fhc_bdlist_unlock();
173                         return (EFAULT);                /* !broken user app */
174                 }
175                 mutex_exit(&test_mutex);
176                 fhc_bdlist_unlock();
177                 AC_ERR_SET(pkt, AC_ERR_MEM_BK);
178                 return (EBUSY);                         /* signal bank in use */
179         }

181         /*
182          * at this point, we have an available bank to test.
183          * create a test buffer
184          */
185         test = kmem_zalloc(sizeof (struct test_info), KM_SLEEP);
186         test->va = vmem_alloc(heap_arena, PAGESIZE, VM_SLEEP);

188         /* fill in all the test info details now */
189         test->mem_info = mem_info;
190         test->board = softsp->board;
191         test->bank = pkt->bank;
192         test->bufp = kmem_alloc(TEST_PAGESIZE, KM_SLEEP);
193         test->info.handle = atomic_inc_32_nv(&mem_test_sequence_id);
195         test->info.handle = atomic_add_32_nv(&mem_test_sequence_id, 1);
194         (void) drv_getparm(PPID, (ulong_t *)(&(test->info.tester_pid)));
195         test->info.prev_condition = mem_info->condition;
196         test->info.page_size = TEST_PAGESIZE;
197         /* If Blackbird ever gets a variable line size, this will change. */
198         test->info.line_size = cpunodes[CPU->cpu_id].ecache_linesize;
199         decode = (pkt->bank == Bank0) ?
200             *softsp->ac_memdecode0 : *softsp->ac_memdecode1;
201         test->info.afar_base = GRP_REALBASE(decode);
202         test->info.bank_size = GRP_UK2SPAN(decode);

204         /* return the information to the user */
205         if (ddi_copyout(&test->info, pkt->cmd_cfga.private,
206             sizeof (ac_mem_test_start_t), flag) != 0) {

208                 /* oh well, tear down the test now */
209                 kmem_free(test->bufp, TEST_PAGESIZE);
210                 vmem_free(heap_arena, test->va, PAGESIZE);
211                 kmem_free(test, sizeof (struct test_info));

213                 fhc_bdlist_unlock();
214                 return (EFAULT);
215         }

217         mem_info->busy = TRUE;
```



```
219         /* finally link us into the test database */
220         mutex_enter(&test_mutex);
221         test->next = test_base;
222         test_base = test;
223         mutex_exit(&test_mutex);

225         fhc_bdlist_unlock();

227 #ifdef DEBUG
228         cmn_err(CE_NOTE, "!memtest: start test[%u]: board %d, bank %d",
229                 test->info.handle, test->board, test->bank);
230 #endif /* DEBUG */
231         return (DDI_SUCCESS);
232 }
______unchanged_portion_omitted_

351 int
352 ac_mem_test_read(ac_cfga_pkt_t *pkt, int flag)
353 {
354         struct test_info *test;
355         uint_t page_offset;
356         uint64_t page_pa;
357         uint_t pstate_save;
358         caddr_t src_va, dst_va;
359         uint64_t orig_err;
360         int retval = DDI_SUCCESS;
361         sunfire_processor_error_regs_t error_buf;
362         int error_found;
363         ac_mem_test_read_t t_read;

365 #ifdef _MULTI_DATAMODEL
366         switch (ddi_model_convert_from(flag & FMODELS)) {
367         case DDI_MODEL_ILP32: {
368                 ac_mem_test_read32_t t_read32;

370                 if (ddi_copyin(pkt->cmd_cfga.private, &t_read32,
371                     sizeof (ac_mem_test_read32_t), flag) != 0)
372                         return (EFAULT);
373                 t_read.handle = t_read32.handle;
374                 t_read.page_buf = (void *)(uintptr_t)t_read32.page_buf;
375                 t_read.address = t_read32.address;
376                 t_read.error_buf = (sunfire_processor_error_regs_t *)
377                     (uintptr_t)t_read32.error_buf;
378                 break;
379         }
380         case DDI_MODEL_NONE:
381                 if (ddi_copyin(pkt->cmd_cfga.private, &t_read,
382                     sizeof (ac_mem_test_read_t), flag) != 0)
383                         return (EFAULT);
384                 break;
385         }
386 #else /* _MULTI_DATAMODEL */
387         if (ddi_copyin(pkt->cmd_cfga.private, &t_read,
388             sizeof (ac_mem_test_read_t), flag) != 0)
389                 return (EFAULT);
390 #endif /* _MULTI_DATAMODEL */

392         /* verify the handle */
393         mutex_enter(&test_mutex);
394         for (test = test_base; test != NULL; test = test->next) {
395                 if (test->info.handle == t_read.handle)
396                         break;
397         }
398         if (test == NULL) {
399                 mutex_exit(&test_mutex);
```

```
400                 AC_ERR_SET(pkt, AC_ERR_MEM_TEST);
401                 return (EINVAL);
402         }

404         /* bump the busy bit */
405         atomic_inc_32(&test->in_test);
407         atomic_add_32(&test->in_test, 1);
406         mutex_exit(&test_mutex);

408         /* verify the remaining parameters */
409         if ((t_read.address.page_num >=
410             test->info.bank_size / test->info.page_size) ||
411             (t_read.address.line_count == 0) ||
412             (t_read.address.line_count >
413             test->info.page_size / test->info.line_size) ||
414             (t_read.address.line_offset >=
415             test->info.page_size / test->info.line_size) ||
416             ((t_read.address.line_offset + t_read.address.line_count) >
417             test->info.page_size / test->info.line_size)) {
418                 AC_ERR_SET(pkt, AC_ERR_MEM_TEST_PAR);
419                 retval = EINVAL;
420                 goto read_done;
421         }

423         page_offset = t_read.address.line_offset * test->info.line_size;
424         page_pa = test->info.afar_base +
425             t_read.address.page_num * test->info.page_size;
426         dst_va = test->bufp + page_offset;
427         src_va = test->va + page_offset;

429         /* time to go quiet */
430         kpreempt_disable();

432         /* we need a va for the block instructions */
433         ac_mapin(page_pa, test->va);

435         pstate_save = disable_vec_intr();

437         /* disable errors */
438         orig_err = get_error_enable();
439         set_error_enable(orig_err & ~(EER_CEEN | EER_NCEEN));

441         /* copy the data again (using our very special copy) */
442         ac_blkcopy(src_va, dst_va, t_read.address.line_count,
443             test->info.line_size);

445         /* process errors (if any) */
446         error_buf.module_id = CPU->cpu_id;
447         get_asyncflt(&(error_buf.afsr));
448         get_asyncaddr(&(error_buf.afar));
449         get_udb_errors(&(error_buf.udbh_error_reg),
450             &(error_buf.udbl_error_reg));

452         /*
453          * clean up after our no-error copy but before enabling ints.
454          * XXX what to do about other error types?
455          */
456         if (error_buf.afsr & (P_AFSR_CE | P_AFSR_UE)) {
457                 extern void clr_datapath(void); /* XXX */

459                 clr_datapath();
460                 set_asyncflt(error_buf.afsr);
461                 retval = EIO;
462                 error_found = TRUE;
463         } else {
464                 error_found = FALSE;
```



```
465         }

467         /* errors back on */
468         set_error_enable(orig_err);

470         enable_vec_intr(pstate_save);

472         /* tear down translation (who needs an mmu) */
473         ac_unmap(test->va);

475         /* we're back! */
476         kpreempt_enable();

478         /*
479          * If there was a data error, attempt to return the error_buf
480          * to the user.
481          */
482         if (error_found) {
483                 if (ddi_copyout(&error_buf, t_read.error_buf,
484                     sizeof (sunfire_processor_error_regs_t), flag) != 0) {
485                         retval = EFAULT;
486                         /* Keep going */
487                 }
488         }

490         /*
491          * Then, return the page to the user (always)
492          */
493         if (ddi_copyout(dst_va, (caddr_t)(t_read.page_buf) + page_offset,
494             t_read.address.line_count * test->info.line_size, flag) != 0) {
495                 retval = EFAULT;
496         }

498 read_done:
499         atomic_dec_32(&test->in_test);
501         atomic_add_32(&test->in_test, -1);
500         return (retval);
501 }

503 int
504 ac_mem_test_write(ac_cfga_pkt_t *pkt, int flag)
505 {
506         struct test_info *test;
507         uint_t page_offset;
508         uint64_t page_pa;
509         uint_t pstate_save;
510         caddr_t src_va, dst_va;
511         int retval = DDI_SUCCESS;
512         ac_mem_test_write_t t_write;

514 #ifdef _MULTI_DATAMODEL
515         switch (ddi_model_convert_from(flag & FMODELS)) {
516         case DDI_MODEL_ILP32: {
517                 ac_mem_test_write32_t t_write32;

519                 if (ddi_copyin(pkt->cmd_cfga.private, &t_write32,
520                     sizeof (ac_mem_test_write32_t), flag) != 0)
521                         return (EFAULT);
522                 t_write.handle = t_write32.handle;
523                 t_write.page_buf = (void *)(uintptr_t)t_write32.page_buf;
524                 t_write.address = t_write32.address;
525                 break;
526         }
527         case DDI_MODEL_NONE:
528                 if (ddi_copyin(pkt->cmd_cfga.private, &t_write,
529                     sizeof (ac_mem_test_write_t), flag) != 0)
```

```
530                         return (EFAULT);
531                 break;
532         }
533 #else /* _MULTI_DATAMODEL */
534         if (ddi_copyin(pkt->cmd_cfga.private, &t_write,
535             sizeof (ac_mem_test_write_t), flag) != 0)
536                 return (EFAULT);
537 #endif /* _MULTI_DATAMODEL */

539         /* verify the handle */
540         mutex_enter(&test_mutex);
541         for (test = test_base; test != NULL; test = test->next) {
542                 if (test->info.handle == t_write.handle)
543                         break;
544         }
545         if (test == NULL) {
546                 mutex_exit(&test_mutex);
547                 return (EINVAL);
548         }

550         /* bump the busy bit */
551         atomic_inc_32(&test->in_test);
553         atomic_add_32(&test->in_test, 1);
552         mutex_exit(&test_mutex);

554         /* verify the remaining parameters */
555         if ((t_write.address.page_num >=
556             test->info.bank_size / test->info.page_size) ||
557             (t_write.address.line_count == 0) ||
558             (t_write.address.line_count >
559             test->info.page_size / test->info.line_size) ||
560             (t_write.address.line_offset >=
561             test->info.page_size / test->info.line_size) ||
562             ((t_write.address.line_offset + t_write.address.line_count) >
563             test->info.page_size / test->info.line_size)) {
564                 AC_ERR_SET(pkt, AC_ERR_MEM_TEST_PAR);
565                 retval = EINVAL;
566                 goto write_done;
567         }

569         page_offset = t_write.address.line_offset * test->info.line_size;
570         page_pa = test->info.afar_base +
571             t_write.address.page_num * test->info.page_size;
572         src_va = test->bufp + page_offset;
573         dst_va = test->va + page_offset;

575         /* copy in the specified user data */
576         if (ddi_copyin((caddr_t)(t_write.page_buf) + page_offset, src_va,
577             t_write.address.line_count * test->info.line_size, flag) != 0) {
578                 retval = EFAULT;
579                 goto write_done;
580         }

582         /* time to go quiet */
583         kpreempt_disable();

585         /* we need a va for the block instructions */
586         ac_mapin(page_pa, test->va);

588         pstate_save = disable_vec_intr();

590         /* copy the data again (using our very special copy) */
591         ac_blkcopy(src_va, dst_va, t_write.address.line_count,
592             test->info.line_size);

594         enable_vec_intr(pstate_save);
```



```
596         /* tear down translation (who needs an mmu) */
597         ac_unmap(test->va);

599         /* we're back! */
600         kpreempt_enable();

602 write_done:
603         atomic_dec_32(&test->in_test);
605         atomic_add_32(&test->in_test, -1);
604         return (retval);
605 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
    1602 Mon Jul 28 07:45:03 2014
new/usr/src/uts/sun4u/sys/pci/pci_axq.h
5045 use atomic_{inc,dec}_* instead of atomic_add_*
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2001 by Sun Microsystems, Inc.
 * All rights reserved.
 */

#ifndef _SYS_PCI_AXQ_H
#define _SYS_PCI_AXQ_H

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/types.h>
#include <sys/atomic.h>

#ifdef  __cplusplus
extern "C" {
#endif

#define PIO_LIMIT_ENTER(p)      { \
                                int n;\
                                for (;;) {\
                                        do {\
                                                n = p->pbm_pio_counter;\
                                        } while (n <= 0);\
                                        if (atomic_dec_32_nv(\
                                            (uint_t *)&p->pbm_pio_counter)\
                                        if (atomic_add_32_nv(\
                                            (uint_t *)&p->pbm_pio_counter, -1)\
                                            == (n - 1))\
                                                break;\
                                        atomic_inc_32(\
                                            (uint_t *)&p->pbm_pio_counter);\
                                        atomic_add_32(\
                                            (uint_t *)&p->pbm_pio_counter, 1);\
                                }\
                                }



#define PIO_LIMIT_EXIT(p)       atomic_inc_32((uint_t *)&p->pbm_pio_counter);
#define PIO_LIMIT_EXIT(p)       atomic_add_32((uint_t *)&p->pbm_pio_counter, 1);
```



```
extern void pci_axq_setup(ddi_map_req_t *mp, pbm_t *pbm_p);
extern void pci_axq_pio_limit(pbm_t *pbm_p);

#ifdef  __cplusplus
}
_____unchanged_portion_omitted_
```