```
**********************************************************
   21346 Fri May  8 18:10:20 2015
new/usr/src/cmd/mdb/common/modules/genunix/findstack.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

/*ARGSUSED*/
int
stacks(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
{
        size_t idx;

        char *seen = NULL;

        const char *caller_str = NULL;
        const char *excl_caller_str = NULL;
        uintptr_t caller = 0, excl_caller = 0;
        const char *module_str = NULL;
        const char *excl_module_str = NULL;
        stacks_module_t module, excl_module;
        const char *sobj = NULL;
        const char *excl_sobj = NULL;
        uintptr_t sobj_ops = 0, excl_sobj_ops = 0;
        const char *tstate_str = NULL;
        const char *excl_tstate_str = NULL;
        uint_t tstate = -1U;
        uint_t excl_tstate = -1U;
        uint_t printed = 0;

        uint_t all = 0;
        uint_t force = 0;
        uint_t interesting = 0;
        uint_t verbose = 0;

        /*
         * We have a slight behavior difference between having piped
         * input and 'addr::stacks'.  Without a pipe, we assume the
         * thread pointer given is a representative thread, and so
         * we include all similar threads in the system in our output.
         *
         * With a pipe, we filter down to just the threads in our
         * input.
         */
        uint_t addrspec = (flags & DCMD_ADDRSPEC);
        uint_t only_matching = addrspec && (flags & DCMD_PIPE);

        mdb_pipe_t p;

        bzero(&module, sizeof (module));
        bzero(&excl_module, sizeof (excl_module));

        if (mdb_getopts(argc, argv,
            'a', MDB_OPT_SETBITS, TRUE, &all,
            'f', MDB_OPT_SETBITS, TRUE, &force,
            'i', MDB_OPT_SETBITS, TRUE, &interesting,
            'v', MDB_OPT_SETBITS, TRUE, &verbose,
            'c', MDB_OPT_STR, &caller_str,
            'C', MDB_OPT_STR, &excl_caller_str,
            'm', MDB_OPT_STR, &module_str,
```



```
            'M', MDB_OPT_STR, &excl_module_str,
            's', MDB_OPT_STR, &sobj,
            'S', MDB_OPT_STR, &excl_sobj,
            't', MDB_OPT_STR, &tstate_str,
            'T', MDB_OPT_STR, &excl_tstate_str,
            NULL) != argc)
                return (DCMD_USAGE);

        if (interesting) {
                if (sobj != NULL || excl_sobj != NULL ||
                    tstate_str != NULL || excl_tstate_str != NULL) {
                        mdb_warn(
                            "stacks: -i is incompatible with -[sStT]\n");
                        return (DCMD_USAGE);
                }
                excl_sobj = "CV";
                excl_tstate_str = "FREE";
        }

        if (caller_str != NULL) {
                mdb_set_dot(0);
                if (mdb_eval(caller_str) != 0) {
                        mdb_warn("stacks: evaluation of \"%s\" failed",
                            caller_str);
                        return (DCMD_ABORT);
                }
                caller = mdb_get_dot();
        }

        if (excl_caller_str != NULL) {
                mdb_set_dot(0);
                if (mdb_eval(excl_caller_str) != 0) {
                        mdb_warn("stacks: evaluation of \"%s\" failed",
                            excl_caller_str);
                        return (DCMD_ABORT);
                }
                excl_caller = mdb_get_dot();
        }
        mdb_set_dot(addr);

        if (module_str != NULL && stacks_module_find(module_str, &module) != 0)
                return (DCMD_ABORT);

        if (excl_module_str != NULL &&
            stacks_module_find(excl_module_str, &excl_module) != 0)
                return (DCMD_ABORT);

        if (sobj != NULL && text_to_sobj(sobj, &sobj_ops) != 0)
                return (DCMD_USAGE);

        if (excl_sobj != NULL && text_to_sobj(excl_sobj, &excl_sobj_ops) != 0)
                return (DCMD_USAGE);

        if (sobj_ops != 0 && excl_sobj_ops != 0) {
                mdb_warn("stacks: only one of -s and -S can be specified\n");
                return (DCMD_USAGE);
        }

        if (tstate_str != NULL && text_to_tstate(tstate_str, &tstate) != 0)
                return (DCMD_USAGE);

        if (excl_tstate_str != NULL &&
            text_to_tstate(excl_tstate_str, &excl_tstate) != 0)
                return (DCMD_USAGE);

        if (tstate != -1U && excl_tstate != -1U) {
```

```
702                 mdb_warn("stacks: only one of -t and -T can be specified\n");
703                 return (DCMD_USAGE);
704         }

706         /*
707          * If there's an address specified, we're going to further filter
708          * to only entries which have an address in the input.  To reduce
709          * overhead (and make the sorted output come out right), we
710          * use mdb_get_pipe() to grab the entire pipeline of input, then
711          * use qsort() and bsearch() to speed up the search.
712          */
713         if (addrspec) {
714                 mdb_get_pipe(&p);
715                 if (p.pipe_data == NULL || p.pipe_len == 0) {
716                         p.pipe_data = &addr;
717                         p.pipe_len = 1;
718                 }
719                 qsort(p.pipe_data, p.pipe_len, sizeof (uintptr_t),
720                     uintptrcomp);

722                 /* remove any duplicates in the data */
723                 idx = 0;
724                 while (idx < p.pipe_len - 1) {
725                         uintptr_t *data = &p.pipe_data[idx];
726                         size_t len = p.pipe_len - idx;

728                         if (data[0] == data[1]) {
729                                 memmove(data, data + 1,
730                                     (len - 1) * sizeof (*data));
731                                 p.pipe_len--;
732                                 continue; /* repeat without incrementing idx */
733                         }
734                         idx++;
735                 }

737                 seen = mdb_zalloc(p.pipe_len, UM_SLEEP | UM_GC);
738         }

740         /*
741          * Force a cleanup if we're connected to a live system. Never
742          * do a cleanup after the first invocation around the loop.
743          */
744         force |= (mdb_get_state() == MDB_STATE_RUNNING);
745         if (force && (flags & (DCMD_LOOPFIRST|DCMD_LOOP)) == DCMD_LOOP)
746                 force = 0;

748         stacks_cleanup(force);

750         if (stacks_state == STACKS_STATE_CLEAN) {
751                 int res = stacks_run(verbose, addrspec ? &p : NULL);
752                 if (res != DCMD_OK)
753                         return (res);
754         }

756         for (idx = 0; idx < stacks_array_size; idx++) {
757                 stacks_entry_t *sep = stacks_array[idx];
758                 stacks_entry_t *cur = sep;
759                 int frame;
760                 size_t count = sep->se_count;

762                 if (addrspec) {
763                         stacks_entry_t *head = NULL, *tail = NULL, *sp;
764                         size_t foundcount = 0;
765                         /*
766                          * We use the now-unused hash chain field se_next to
767                          * link together the dups which match our list.
```



```
768                          */
769                         for (sp = sep; sp != NULL; sp = sp->se_dup) {
770                                 uintptr_t *entry = bsearch(&sp->se_thread,
771                                     p.pipe_data, p.pipe_len, sizeof (uintptr_t),
772                                     uintptrcomp);
773                                 if (entry != NULL) {
774                                         foundcount++;
775                                         seen[entry - p.pipe_data]++;
776                                         if (head == NULL)
777                                                 head = sp;
778                                         else
779                                                 tail->se_next = sp;
780                                         tail = sp;
781                                         sp->se_next = NULL;
782                                 }
783                         }
784                         if (head == NULL)
785                                 continue;       /* no match, skip entry */

787                         if (only_matching) {
788                                 cur = sep = head;
789                                 count = foundcount;
790                         }
791                 }

793                 if (caller != 0 && !stacks_has_caller(sep, caller))
794                         continue;

796                 if (excl_caller != 0 && stacks_has_caller(sep, excl_caller))
797                         continue;

799                 if (module.sm_size != 0 && !stacks_has_module(sep, &module))
800                         continue;

802                 if (excl_module.sm_size != 0 &&
803                     stacks_has_module(sep, &excl_module))
804                         continue;

806                 if (tstate != -1U) {
807                         if (tstate == TSTATE_PANIC) {
808                                 if (!sep->se_panic)
809                                         continue;
810                         } else if (sep->se_panic || sep->se_tstate != tstate)
811                                 continue;
812                 }
813                 if (excl_tstate != -1U) {
814                         if (excl_tstate == TSTATE_PANIC) {
815                                 if (sep->se_panic)
816                                         continue;
817                         } else if (!sep->se_panic &&
818                             sep->se_tstate == excl_tstate)
819                                 continue;
820                 }

822                 if (sobj_ops == SOBJ_ALL) {
823                         if (sep->se_sobj_ops == 0)
824                                 continue;
825                 } else if (sobj_ops != 0) {
826                         if (sobj_ops != sep->se_sobj_ops)
827                                 continue;
828                 }

830                 if (!(interesting && sep->se_panic)) {
831                         if (excl_sobj_ops == SOBJ_ALL) {
832                                 if (sep->se_sobj_ops != 0)
833                                         continue;
```

```
834                         } else if (excl_sobj_ops != 0) {
835                                 if (excl_sobj_ops == sep->se_sobj_ops)
836                                         continue;
837                         }
838                 }

840                 if (flags & DCMD_PIPE_OUT) {
841                         while (sep != NULL) {
842                                 mdb_printf("%lr\n", sep->se_thread);
843                                 sep = only_matching ?
844                                     sep->se_next : sep->se_dup;
845                         }
846                         continue;
847                 }

849                 if (all || !printed) {
850                         mdb_printf("%<u>%-?s %-8s %-?s %8s%</u>\n",
851                             "THREAD", "STATE", "SOBJ", "COUNT");
852                         printed = 1;
853                 }

855                 do {
856                         char state[20];
857                         char sobj[100];

859                         tstate_to_text(cur->se_tstate, cur->se_panic,
860                             state, sizeof (state));
861                         sobj_to_text(cur->se_sobj_ops,
862                             sobj, sizeof (sobj));

864                         if (cur == sep)
865                                 mdb_printf("%-?p %-8s %-?s %8d\n",
866                                     cur->se_thread, state, sobj, count);
867                         else
868                                 mdb_printf("%-?p %-8s %-?s %8s\n",
869                                     cur->se_thread, state, sobj, "-");

871                         cur = only_matching ? cur->se_next : cur->se_dup;
872                 } while (all && cur != NULL);

874                 if (sep->se_failed != 0) {
875                         char *reason;
876                         switch (sep->se_failed) {
877                         case FSI_FAIL_NOTINMEMORY:
878                                 reason = "thread not in memory";
879                                 break;
877                         case FSI_FAIL_THREADCORRUPT:
878                                 reason = "thread structure stack info corrupt";
879                                 break;
880                         case FSI_FAIL_STACKNOTFOUND:
881                                 reason = "no consistent stack found";
882                                 break;
883                         default:
884                                 reason = "unknown failure";
885                                 break;
886                         }
887                         mdb_printf("%?s <%s>\n", "", reason);
888                 }

890                 for (frame = 0; frame < sep->se_depth; frame++)
891                         mdb_printf("%?s %a\n", "", sep->se_stack[frame]);
892                 if (sep->se_overflow)
893                         mdb_printf("%?s ... truncated ...\n", "");
894                 mdb_printf("\n");
895         }
```



```
897         if (flags & DCMD_ADDRSPEC) {
898                 for (idx = 0; idx < p.pipe_len; idx++)
899                         if (seen[idx] == 0)
900                                 mdb_warn("stacks: %p not in thread list\n",
901                                     p.pipe_data[idx]);
902         }
903         return (DCMD_OK);
904 }
```

_____unchanged_portion_omitted_

****
2801 Fri May 8 18:10:21 2015
new/usr/src/cmd/mdb/common/modules/genunix/findstack.h
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory. The code is there and in theory it runs when we get *extremely* low
on memory. In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
****
_____***unchanged_portion_omitted_***

```
 49 #define FSI_FAIL_BADTHREAD      1
 50 #define FSI_FAIL_THREADCORRUPT  2
 51 #define FSI_FAIL_STACKNOTFOUND  3
 50 #define FSI_FAIL_NOTINMEMORY    2
 51 #define FSI_FAIL_THREADCORRUPT  3
 52 #define FSI_FAIL_STACKNOTFOUND  4

 53 typedef struct stacks_module {
 54         char            sm_name[MAXPATHLEN]; /* name of module */
 55         uintptr_t       sm_text;        /* base address of text in module */
 56         size_t          sm_size;        /* size of text in module */
 57 } stacks_module_t;
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   11424 Fri May  8 18:10:21 2015
new/usr/src/cmd/mdb/common/modules/genunix/findstack_subr.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

 140 /*ARGSUSED*/
 141 int
 142 stacks_findstack(uintptr_t addr, findstack_info_t *fsip, uint_t print_warnings)
 143 {
 144         mdb_findstack_kthread_t thr;
 145         size_t stksz;
 146         uintptr_t ubase, utop;
 147         uintptr_t kbase, ktop;
 148         uintptr_t win, sp;

 150         fsip->fsi_failed = 0;
 151         fsip->fsi_pc = 0;
 152         fsip->fsi_sp = 0;
 153         fsip->fsi_depth = 0;
 154         fsip->fsi_overflow = 0;

 156         if (mdb_ctf_vread(&thr, "kthread_t", "mdb_findstack_kthread_t",
 157             addr, print_warnings ? 0 : MDB_CTF_VREAD_QUIET) == -1) {
 158                 fsip->fsi_failed = FSI_FAIL_BADTHREAD;
 159                 return (DCMD_ERR);
 160         }

 162         fsip->fsi_sobj_ops = (uintptr_t)thr.t_sobj_ops;
 163         fsip->fsi_tstate = thr.t_state;
 164         fsip->fsi_panic = !!(thr.t_flag & T_PANIC);

 166         if ((thr.t_schedflag & TS_LOAD) == 0) {
 167                 if (print_warnings)
 168                         mdb_warn("thread %p isn't in memory\n", addr);
 169                 fsip->fsi_failed = FSI_FAIL_NOTINMEMORY;
 170                 return (DCMD_ERR);
 171         }

 166         if (thr.t_stk < thr.t_stkbase) {
 167                 if (print_warnings)
 168                         mdb_warn(
 169                             "stack base or stack top corrupt for thread %p\n",
 170                             addr);
 171                 fsip->fsi_failed = FSI_FAIL_THREADCORRUPT;
 172                 return (DCMD_ERR);
 173         }

 175         kbase = (uintptr_t)thr.t_stkbase;
 176         ktop = (uintptr_t)thr.t_stk;
 177         stksz = ktop - kbase;

 179 #ifdef __amd64
 180         /*
 181          * The stack on amd64 is intentionally misaligned, so ignore the top
 182          * half-frame.  See thread_stk_init().  When handling traps, the frame
 183          * is automatically aligned by the hardware, so we only alter ktop if
 184          * needed.
 185          */
```



```
 186         if ((ktop & (STACK_ALIGN - 1)) != 0)
 187                 ktop -= STACK_ENTRY_ALIGN;
 188 #endif

 190         /*
 191          * If the stack size is larger than a meg, assume that it's bogus.
 192          */
 193         if (stksz > TOO_BIG_FOR_A_STACK) {
 194                 if (print_warnings)
 195                         mdb_warn("stack size for thread %p is too big to be "
 196                             "reasonable\n", addr);
 197                 fsip->fsi_failed = FSI_FAIL_THREADCORRUPT;
 198                 return (DCMD_ERR);
 199         }

 201         /*
 202          * This could be (and was) a UM_GC allocation.  Unfortunately,
 203          * stksz tends to be very large.  As currently implemented, dcmds
 204          * invoked as part of pipelines don't have their UM_GC-allocated
 205          * memory freed until the pipeline completes.  With stksz in the
 206          * neighborhood of 20k, the popular ::walk thread |::findstack
 207          * pipeline can easily run memory-constrained debuggers (kmdb) out
 208          * of memory.  This can be changed back to a gc-able allocation when
 209          * the debugger is changed to free UM_GC memory more promptly.
 210          */
 211         ubase = (uintptr_t)mdb_alloc(stksz, UM_SLEEP);
 212         utop = ubase + stksz;
 213         if (mdb_vread((caddr_t)ubase, stksz, kbase) != stksz) {
 214                 mdb_free((void *)ubase, stksz);
 215                 if (print_warnings)
 216                         mdb_warn("couldn't read entire stack for thread %p\n",
 217                             addr);
 218                 fsip->fsi_failed = FSI_FAIL_THREADCORRUPT;
 219                 return (DCMD_ERR);
 220         }

 222         /*
 223          * Try the saved %sp first, if it looks reasonable.
 224          */
 225         sp = KTOU((uintptr_t)thr.t_sp + STACK_BIAS);
 226         if (sp >= ubase && sp <= utop) {
 227                 if (crawl(sp, kbase, ktop, ubase, 0, fsip) == CRAWL_FOUNDALL) {
 228                         fsip->fsi_sp = (uintptr_t)thr.t_sp;
 229 #if !defined(__i386)
 230                         fsip->fsi_pc = (uintptr_t)thr.t_pc;
 231 #endif
 232                         goto found;
 233                 }
 234         }

 236         /*
 237          * Now walk through the whole stack, starting at the base,
 238          * trying every possible "window".
 239          */
 240         for (win = ubase;
 241             win + sizeof (struct rwindow) <= utop;
 242             win += sizeof (struct rwindow *)) {
 243                 if (crawl(win, kbase, ktop, ubase, 1, fsip) == CRAWL_FOUNDALL) {
 244                         fsip->fsi_sp = UTOK(win) - STACK_BIAS;
 245                         goto found;
 246                 }
 247         }

 249         /*
 250          * We didn't conclusively find the stack.  So we'll take another lap,
 251          * and print out anything that looks possible.
```

```
252          */
253         if (print_warnings)
254                 mdb_printf("Possible stack pointers for thread %p:\n", addr);
255         (void) mdb_vread((caddr_t)ubase, stksz, kbase);

257         for (win = ubase;
258             win + sizeof (struct rwindow) <= utop;
259             win += sizeof (struct rwindow *)) {
260                 uintptr_t fp = ((struct rwindow *)win)->rw_fp;
261                 int levels;

263                 if ((levels = crawl(win, kbase, ktop, ubase, 1, fsip)) > 1) {
264                         if (print_warnings)
265                                 mdb_printf("  %p (%d)\n", fp, levels);
266                 } else if (levels == CRAWL_FOUNDALL) {
267                         /*
268                          * If this is a live system, the stack could change
269                          * between the two mdb_vread(ubase, utop, kbase)'s,
270                          * and we could have a fully valid stack here.
271                          */
272                         fsip->fsi_sp = UTOK(win) - STACK_BIAS;
273                         goto found;
274                 }
275         }

277         fsip->fsi_depth = 0;
278         fsip->fsi_overflow = 0;
279         fsip->fsi_failed = FSI_FAIL_STACKNOTFOUND;

281         mdb_free((void *)ubase, stksz);
282         return (DCMD_ERR);
283 found:
284         mdb_free((void *)ubase, stksz);
285         return (DCMD_OK);
286 }
```

*unchanged_portion_omitted_*

****
109418 Fri May 8 18:10:21 2015
new/usr/src/cmd/mdb/common/modules/genunix/kmem.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim memory. The code is there and in theory it runs when we get *extremely* low on memory. In practice, it never runs since the definition of low-on-memory is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
****

_____***unchanged_portion_omitted_***

```
4316 static int
4317 whatthread_walk_thread(uintptr_t addr, const kthread_t *t, whatthread_t *w)
4318 {
4319         uintptr_t current, data;

4321         if (t->t_stkbase == NULL)
4322                 return (WALK_NEXT);

4324         /*
4325          * Warn about swapped out threads, but drive on anyway
4326          */
4327         if (!(t->t_schedflag & TS_LOAD)) {
4328                 mdb_warn("thread %p's stack swapped out\n", addr);
4329                 return (WALK_NEXT);
4330         }

4332         /*
4325          * Search the thread's stack for the given pointer.  Note that it would
4326          * be more efficient to follow ::kgrep's lead and read in page-sized
4327          * chunks, but this routine is already fast and simple.
4328          */
4329         for (current = (uintptr_t)t->t_stkbase; current < (uintptr_t)t->t_stk;
4330             current += sizeof (uintptr_t)) {
4331                 if (mdb_vread(&data, sizeof (data), current) == -1) {
4332                         mdb_warn("couldn't read thread %p's stack at %p",
4333                             addr, current);
4334                         return (WALK_ERR);
4335                 }

4337                 if (data == w->wt_target) {
4338                         if (w->wt_verbose) {
4339                                 mdb_printf("%p in thread %p's stack%s\n",
4340                                     current, addr, stack_active(t, current));
4341                         } else {
4342                                 mdb_printf("%#lr\n", addr);
4343                                 return (WALK_NEXT);
4344                         }
4345                 }
4346         }

4348         return (WALK_NEXT);
4349 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   21512 Fri May  8 18:10:21 2015
new/usr/src/cmd/mdb/common/modules/genunix/leaky_subr.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
```

```
   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, Version 1.0 only
   6  * (the "License").  You may not use this file except in compliance
   7  * with the License.
   8  *
   9  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
  10  * or http://www.opensolaris.org/os/licensing.
  11  * See the License for the specific language governing permissions
  12  * and limitations under the License.
  13  *
  14  * When distributing Covered Code, include this CDDL HEADER in each
  15  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  16  * If applicable, add the following below this CDDL HEADER, with the
  17  * fields enclosed by brackets "[]" replaced with your own identifying
  18  * information: Portions Copyright [yyyy] [name of copyright owner]
  19  *
  20  * CDDL HEADER END
  21  */
  22 /*
  23  * Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
  24  * Use is subject to license terms.
  25  */

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

  27 #include <mdb/mdb_param.h>
  28 #include <mdb/mdb_modapi.h>

  30 #include <sys/fs/ufs_inode.h>
  31 #include <sys/kmem_impl.h>
  32 #include <sys/vmem_impl.h>
  33 #include <sys/modctl.h>
  34 #include <sys/kobj.h>
  35 #include <sys/kobj_impl.h>
  36 #include <vm/seg_vn.h>
  37 #include <vm/as.h>
  38 #include <vm/seg_map.h>
  39 #include <mdb/mdb_ctf.h>

  41 #include "kmem.h"
  42 #include "leaky_impl.h"

  44 /*
  45  * This file defines the genunix target for leaky.c.  There are three types
  46  * of buffers in the kernel's heap:  TYPE_VMEM, for kmem_oversize allocations,
  47  * TYPE_KMEM, for kmem_cache_alloc() allocations bufctl_audit_ts, and
  48  * TYPE_CACHE, for kmem_cache_alloc() allocation without bufctl_audit_ts.
  49  *
  50  * See "leaky_impl.h" for the target interface definition.
  51  */

  53 #define TYPE_VMEM       0               /* lkb_data is the vmem_seg's size */
```



```
  54 #define TYPE_CACHE      1               /* lkb_cid is the bufctl's cache */
  55 #define TYPE_KMEM       2               /* lkb_cid is the bufctl's cache */

  57 #define LKM_CTL_BUFCTL  0       /* normal allocation, PTR is bufctl */
  58 #define LKM_CTL_VMSEG   1       /* oversize allocation, PTR is vmem_seg_t */
  59 #define LKM_CTL_CACHE   2       /* normal alloc, non-debug, PTR is cache */
  60 #define LKM_CTL_MASK    3L

  62 #define LKM_CTL(ptr, type)      (LKM_CTLPTR(ptr) | (type))
  63 #define LKM_CTLPTR(ctl)         ((uintptr_t)(ctl) & ~(LKM_CTL_MASK))
  64 #define LKM_CTLTYPE(ctl)        ((uintptr_t)(ctl) &  (LKM_CTL_MASK))

  66 static int kmem_lite_count = 0; /* cache of the kernel's version */

  68 /*ARGSUSED*/
  69 static int
  70 leaky_mtab(uintptr_t addr, const kmem_bufctl_audit_t *bcp, leak_mtab_t **lmp)
  71 {
  72         leak_mtab_t *lm = (*lmp)++;

  74         lm->lkm_base = (uintptr_t)bcp->bc_addr;
  75         lm->lkm_bufctl = LKM_CTL(addr, LKM_CTL_BUFCTL);

  77         return (WALK_NEXT);
  78 }
_____unchanged_portion_omitted_

 279 /*ARGSUSED*/
 280 #endif /* ! codereview */
 281 static int
 282 leaky_thread(uintptr_t addr, const kthread_t *t, unsigned long *pagesize)
 283 {
 284         uintptr_t size, base = (uintptr_t)t->t_stkbase;
 285         uintptr_t stk = (uintptr_t)t->t_stk;

 282         /*
 283          * If this thread isn't in memory, we can't look at its stack.  This
 284          * may result in false positives, so we print a warning.
 285          */
 286         if (!(t->t_schedflag & TS_LOAD)) {
 287                 mdb_printf("findleaks: thread %p's stack swapped out; "
 288                     "false positives possible\n", addr);
 289                 return (WALK_NEXT);
 290         }

 287         if (t->t_state != TS_FREE)
 288                 leaky_grep(base, stk - base);

 290         /*
 291          * There is always gunk hanging out between t_stk and the page
 292          * boundary.  If this thread structure wasn't kmem allocated,
 293          * this will include the thread structure itself.  If the thread
 294          * _is_ kmem allocated, we'll be able to get to it via allthreads.
 295          */
 296         size = *pagesize - (stk & (*pagesize - 1));

 298         leaky_grep(stk, size);

 300         return (WALK_NEXT);
 301 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   68570 Fri May  8 18:10:22 2015
new/usr/src/pkg/manifests/system-kernel-platform.mf
remove zulu (XVR-4000)
XVR-4000 was a very expensive, very rare graphics card.
**********************************************************
#
# 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) 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright 2012 Nexenta Systems, Inc. All rights reserved.
# Copyright 2014 Gary Mills
#

#
# The default for payload-bearing actions in this package is to appear in the
# global zone only.  See the include file for greater detail, as well as
# information about overriding the defaults.
#
<include global_zone_only_component>
set name=pkg.fmri value=pkg:/system/kernel/platform@$(PKGVERS)
set name=pkg.description \
    value="core kernel software for a specific hardware platform group"
set name=pkg.summary value="Core Solaris Kernel Architecture"
set name=info.classification value=org.opensolaris.category.2008:System/Core
set name=variant.arch value=$(ARCH)
dir path=platform group=sys
$(sparc_ONLY)dir path=platform/SUNW,A70 group=sys
$(sparc_ONLY)dir path=platform/SUNW,A70/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,A70/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/SUNW,A70/kernel/crypto/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,A70/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,A70/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,A70/kernel/misc/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-CP2300 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-CP2300/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-CP2300/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-CP2300/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-CP2300/kernel/tod group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-CP2300/kernel/tod/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-CP3010 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-CP3010/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-CP3010/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-CP3010/kernel/crypto/$(ARCH64) \
    group=sys
```



```
$(sparc_ONLY)dir path=platform/SUNW,Netra-CP3010/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-CP3010/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-T12 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-T12/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-T12/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-T12/kernel/crypto/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-T12/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-T12/kernel/drv/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-T12/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-T12/kernel/misc/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-T4 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-T4/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-T4/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-T4/kernel/crypto/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-T4/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-T4/kernel/drv/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-T4/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Netra-T4/kernel/misc/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,SPARC-Enterprise group=sys
$(sparc_ONLY)dir path=platform/SUNW,SPARC-Enterprise/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,SPARC-Enterprise/kernel/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,SPARC-Enterprise/kernel/cpu group=sys
$(sparc_ONLY)dir path=platform/SUNW,SPARC-Enterprise/kernel/cpu/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,SPARC-Enterprise/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/SUNW,SPARC-Enterprise/kernel/crypto/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,SPARC-Enterprise/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,SPARC-Enterprise/kernel/drv/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,SPARC-Enterprise/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,SPARC-Enterprise/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Serverblade1 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Serverblade1/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Serverblade1/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,Serverblade1/kernel/drv/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Serverblade1/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Serverblade1/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-100 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-100/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-100/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-100/kernel/drv/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-100/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-100/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-1000 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-1000/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-1000/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-1000/kernel/crypto/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-1000/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-1000/kernel/drv/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-1000/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-1000/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-1500 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-1500/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-1500/kernel/crypto group=sys
```

```
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-1500/kernel/crypto/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-1500/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-1500/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-1500/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-2500 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-2500/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-2500/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-2500/kernel/crypto/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-2500/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-2500/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Blade-2500/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-15000 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-15000/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-15000/kernel/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-15000/kernel/cpu group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-15000/kernel/cpu/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-15000/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-15000/kernel/crypto/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-15000/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-15000/kernel/drv/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-15000/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-15000/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-280R group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-280R/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-280R/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-280R/kernel/crypto/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-280R/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-280R/kernel/drv/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-280R/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-280R/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-480R group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-480R/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-480R/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-480R/kernel/crypto/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-480R/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-480R/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-880 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-880/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-880/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-880/kernel/crypto/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-880/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-880/kernel/drv/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-880/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-880/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-T200 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-T200/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-T200/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-T200/kernel/drv/$(ARCH64) \
    group=sys
```



```
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V215 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V215/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V215/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V215/kernel/crypto/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V215/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V215/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V215/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V240 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V240/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V240/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V240/kernel/crypto/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V240/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V240/kernel/drv/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V240/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V240/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V250 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V250/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V250/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V250/kernel/crypto/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V250/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V250/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V440 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V440/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V440/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V440/kernel/crypto/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V440/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V440/kernel/drv/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V440/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V440/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V445 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V445/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V445/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V445/kernel/crypto/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V445/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire-V445/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire/kernel/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire/kernel/cpu group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire/kernel/cpu/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire/kernel/crypto/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire/kernel/drv/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Sun-Fire/kernel/misc/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-250 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-250/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-250/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-250/kernel/drv/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-250/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-250/kernel/misc/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-4 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-4/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-4/kernel/drv group=sys
```

```
$(sparc_ONLY)dir path=platform/SUNW,Ultra-4/kernel/drv/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-4/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-4/kernel/misc/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-5_10 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-5_10/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-5_10/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-5_10/kernel/misc/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-80 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-80/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-80/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-80/kernel/misc/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-Enterprise group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-Enterprise-10000 group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-Enterprise-10000/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-Enterprise-10000/kernel/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-Enterprise-10000/kernel/cpu \
    group=sys
$(sparc_ONLY)dir \
    path=platform/SUNW,Ultra-Enterprise-10000/kernel/cpu/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-Enterprise-10000/kernel/drv \
    group=sys
$(sparc_ONLY)dir \
    path=platform/SUNW,Ultra-Enterprise-10000/kernel/drv/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-Enterprise-10000/kernel/misc \
    group=sys
$(sparc_ONLY)dir \
    path=platform/SUNW,Ultra-Enterprise-10000/kernel/misc/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-Enterprise/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-Enterprise/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-Enterprise/kernel/drv/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-Enterprise/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,Ultra-Enterprise/kernel/misc/$(ARCH64) \
    group=sys
$(sparc_ONLY)dir path=platform/SUNW,UltraAX-i2 group=sys
$(sparc_ONLY)dir path=platform/SUNW,UltraAX-i2/kernel group=sys
$(sparc_ONLY)dir path=platform/SUNW,UltraAX-i2/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/SUNW,UltraAX-i2/kernel/misc/$(ARCH64) group=sys
$(i386_ONLY)dir path=platform/i86pc group=sys
$(i386_ONLY)dir path=platform/i86pc/$(ARCH64) group=sys
$(i386_ONLY)dir path=platform/i86pc/kernel group=sys
$(i386_ONLY)dir path=platform/i86pc/kernel/$(ARCH64) group=sys
$(i386_ONLY)dir path=platform/i86pc/kernel/cpu group=sys
$(i386_ONLY)dir path=platform/i86pc/kernel/cpu/$(ARCH64) group=sys
$(i386_ONLY)dir path=platform/i86pc/kernel/dacf group=sys
$(i386_ONLY)dir path=platform/i86pc/kernel/dacf/$(ARCH64) group=sys
$(i386_ONLY)dir path=platform/i86pc/kernel/drv group=sys
$(i386_ONLY)dir path=platform/i86pc/kernel/drv/$(ARCH64) group=sys
$(i386_ONLY)dir path=platform/i86pc/kernel/mach group=sys
$(i386_ONLY)dir path=platform/i86pc/kernel/mach/$(ARCH64) group=sys
$(i386_ONLY)dir path=platform/i86pc/kernel/misc group=sys
$(i386_ONLY)dir path=platform/i86pc/kernel/misc/$(ARCH64) group=sys
$(i386_ONLY)dir path=platform/i86pc/ucode group=sys
$(i386_ONLY)dir path=platform/i86xpv group=sys
$(i386_ONLY)dir path=platform/i86xpv/kernel group=sys
$(i386_ONLY)dir path=platform/i86xpv/kernel/$(ARCH64) group=sys
$(i386_ONLY)dir path=platform/i86xpv/kernel/cpu group=sys
$(i386_ONLY)dir path=platform/i86xpv/kernel/cpu/$(ARCH64) group=sys
$(i386_ONLY)dir path=platform/i86xpv/kernel/dacf group=sys
$(i386_ONLY)dir path=platform/i86xpv/kernel/dacf/$(ARCH64) group=sys
$(i386_ONLY)dir path=platform/i86xpv/kernel/drv group=sys
$(i386_ONLY)dir path=platform/i86xpv/kernel/drv/$(ARCH64) group=sys
$(i386_ONLY)dir path=platform/i86xpv/kernel/mach group=sys
$(i386_ONLY)dir path=platform/i86xpv/kernel/mach/$(ARCH64) group=sys
$(i386_ONLY)dir path=platform/i86xpv/kernel/misc group=sys
```



```
$(i386_ONLY)dir path=platform/i86xpv/kernel/misc/$(ARCH64) group=sys
$(i386_ONLY)dir path=platform/i86xpv/kernel/tod group=sys
$(i386_ONLY)dir path=platform/i86xpv/kernel/tod/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/sun4u group=sys
$(sparc_ONLY)dir path=platform/sun4u-us3 group=sys
$(sparc_ONLY)dir path=platform/sun4u-us3/kernel group=sys
$(sparc_ONLY)dir path=platform/sun4u-us3/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/sun4u-us3/kernel/crypto/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/sun4u/kernel group=sys
$(sparc_ONLY)dir path=platform/sun4u/kernel/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/sun4u/kernel/cpu group=sys
$(sparc_ONLY)dir path=platform/sun4u/kernel/cpu/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/sun4u/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/sun4u/kernel/crypto/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/sun4u/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/sun4u/kernel/drv/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/sun4u/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/sun4u/kernel/misc/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/sun4u/kernel/strmod group=sys
$(sparc_ONLY)dir path=platform/sun4u/kernel/strmod/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/sun4u/kernel/tod group=sys
$(sparc_ONLY)dir path=platform/sun4u/kernel/tod/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/sun4v group=sys
$(sparc_ONLY)dir path=platform/sun4v/kernel group=sys
$(sparc_ONLY)dir path=platform/sun4v/kernel/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/sun4v/kernel/cpu group=sys
$(sparc_ONLY)dir path=platform/sun4v/kernel/cpu/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/sun4v/kernel/crypto group=sys
$(sparc_ONLY)dir path=platform/sun4v/kernel/crypto/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/sun4v/kernel/drv group=sys
$(sparc_ONLY)dir path=platform/sun4v/kernel/drv/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/sun4v/kernel/misc group=sys
$(sparc_ONLY)dir path=platform/sun4v/kernel/misc/$(ARCH64) group=sys
$(sparc_ONLY)dir path=platform/sun4v/kernel/sys group=sys
$(sparc_ONLY)dir path=platform/sun4v/kernel/sys/$(ARCH64) group=sys
$(sparc_ONLY)dir path=usr/share/man
dir path=usr/share/man/man4
$(sparc_ONLY)dir path=usr/share/man/man7d
$(sparc_ONLY)dir path=usr/share/man/man7m
$(sparc_ONLY)driver name=ac
$(i386_ONLY)driver name=acpinex alias=acpivirtnex
$(i386_ONLY)driver name=acpippm
$(sparc_ONLY)driver name=adm1026 alias=i2c-adm1026
$(sparc_ONLY)driver name=adm1031 alias=i2c-adm1031
$(i386_ONLY)driver name=amd_iommu perms="* 0644 root sys" \
    alias=pci1002,5a23 \
    alias=pci1022,11ff
$(sparc_ONLY)driver name=axq alias=SUNW,axq
$(i386_ONLY)driver name=balloon perms="* 0444 root sys"
$(sparc_ONLY)driver name=bbc_beep alias=SUNW,bbc-beep
$(sparc_ONLY)driver name=central
$(i386_ONLY)driver name=cpudrv alias=cpu
$(sparc_ONLY)driver name=ctsmc alias=nct-ds80ch11-smc
$(sparc_ONLY)driver name=db21554 \
    alias=pci108e,6300 \
    alias=pci108e,6301 \
    alias=pci108e,6302 \
    alias=pci108e,6303 \
    alias=pci108e,6310 \
    alias=pci108e,6311 \
    alias=pci108e,6312 \
    alias=pci108e,6313 \
    alias=pci108e,6320 \
    alias=pci108e,6323 \
    alias=pci108e,6330 \
    alias=pci108e,6331 \
```

```
391     alias=pci108e,6332 \
392     alias=pci108e,6333 \
393     alias=pci108e,6340 \
394     alias=pci108e,6343 \
395     alias=pci108e,6350 \
396     alias=pci108e,6353 \
397     alias=pci108e,6722 \
398     alias=pciclass,060940 \
399     alias=pciclass,060980
400 $(sparc_ONLY)driver name=dman perms="* 0600 root sys"
401 $(i386_ONLY)driver name=domcaps perms="* 0444 root sys"
402 $(sparc_ONLY)driver name=dr
403 $(sparc_ONLY)driver name=ebus class=ebus \
404     alias=SUNW,ebus \
405     alias=SUNW,sun4v-ebus \
406     alias=isa \
407     alias=jbus-ebus
408 $(sparc_ONLY)driver name=envctrl alias=SUNW,envctrl
409 $(sparc_ONLY)driver name=envctrltwo alias=SUNW,envctrltwo \
410     perms="* 0644 root sys" \
411     policy="read_priv_set=sys_config write_priv_set=sys_config"
412 $(sparc_ONLY)driver name=environ alias=environment
413 $(sparc_ONLY)driver name=epic alias=SUNW,ebus-pic18lf65j10-env
414 $(i386_ONLY)driver name=evtchn perms="* 0666 root sys"
415 $(sparc_ONLY)driver name=fd perms="* 0666 root sys" \
416     alias=SUNW,fdtwo \
417     alias=fdthree \
418     alias=pnpALI,1533,0
419 $(sparc_ONLY)driver name=fhc
420 $(sparc_ONLY)driver name=glvc
421 $(sparc_ONLY)driver name=gpio_87317 alias=ns87317-gpio perms="* 0600 root sys"
422 $(sparc_ONLY)driver name=grbeep alias=SUNW,smbus-beep
423 $(sparc_ONLY)driver name=grfans alias=SUNW,smbus-fan-control
424 $(sparc_ONLY)driver name=grppm alias=SUNW,smbus-ppm
425 $(sparc_ONLY)driver name=hpc3130 perms="* 0644 root sys" \
426     alias=FJSV,hpc3130 \
427     alias=i2c-hpc3130
428 $(sparc_ONLY)driver name=i2bsc alias=SUNW,i2bsc
429 $(sparc_ONLY)driver name=ics951601 alias=i2c-ics951601
430 $(sparc_ONLY)driver name=iosram
431 $(i386_ONLY)driver name=isa alias=pciclass,060100 class=sysbus
432 $(sparc_ONLY)driver name=isadma
433 $(sparc_ONLY)driver name=jbusppm alias=jbus-ppm
434 $(sparc_ONLY)driver name=lm75 alias=i2c-lm75 perms="* 0644 root sys"
435 $(sparc_ONLY)driver name=lombus alias=SUNW,lombus perms="* 0644 root sys"
436 $(sparc_ONLY)driver name=ltc1427 alias=i2c-ltc1427 perms="* 0644 root sys"
437 $(sparc_ONLY)driver name=lw8 perms="* 0644 root sys"
438 $(sparc_ONLY)driver name=m1535ppm alias=ali1535d+-ppm
439 $(sparc_ONLY)driver name=max1617 alias=i2c-max1617
440 $(sparc_ONLY)driver name=mc-opl alias=FJSV,oplmc
441 $(sparc_ONLY)driver name=mc-us3 alias=memory-controller
442 $(sparc_ONLY)driver name=mc-us3i alias=SUNW,UltraSPARC-IIIi,mc
443 $(sparc_ONLY)driver name=mdesc perms="* 0666 root sys"
444 $(sparc_ONLY)driver name=mem_cache
445 $(sparc_ONLY)driver name=mi2cv alias=fire-i2c
446 $(sparc_ONLY)driver name=n2rng \
447     alias=SUNW,kt-rng \
448     alias=SUNW,n2-rng \
449     alias=SUNW,vf-rng
450 $(i386_ONLY)driver name=npe alias=pciex_root_complex
451 $(sparc_ONLY)driver name=ntwdt perms="* 0644 root sys"
452 $(sparc_ONLY)driver name=oplmsu alias=FJSV,oplmsu
453 $(sparc_ONLY)driver name=oplpanel alias=FJSV,panel
454 $(sparc_ONLY)driver name=pca9556 \
455     alias=i2c-pca9555 \
456     alias=i2c-pca9556
```



```
457 $(sparc_ONLY)driver name=pcf8574 perms="* 0644 root sys" \
458     alias=i2c-pcf8574 \
459     alias=nct-PHG,pcf8574
460 $(sparc_ONLY)driver name=pcf8584 \
461     alias=SUNW,bbc-i2c \
462     alias=SUNW,i2c-pic16f747 \
463     alias=nct-PHG,pcf8584
464 $(sparc_ONLY)driver name=pcf8591 perms="* 0644 root sys" \
465     alias=i2c-pcf8591 \
466     alias=nct-PHG,pcf8591
467 $(i386_ONLY)driver name=pci class=pci
468 $(sparc_ONLY)driver name=pcicmu \
469     alias=pci10cf,138f \
470     alias=pci10cf,1390
471 $(sparc_ONLY)driver name=pcipsy class=pci \
472     alias=SUNW,pci \
473     alias=pci \
474     alias=pci108e,8000 \
475     alias=pci108e,a000 \
476     alias=pci108e,a001 \
477     alias=pciclass,060000
478 $(sparc_ONLY)driver name=pcisch class=pci \
479     alias=pci108e,8001 \
480     alias=pci108e,8002 \
481     alias=pci108e,a801
482 $(sparc_ONLY)driver name=pic16f747 alias=SUNW,ebus-pic16f747-env
483 $(sparc_ONLY)driver name=pic16f819 alias=SUNW,i2c-auxfan1
484 $(i386_ONLY)driver name=pit_beep alias=SUNW,pit_beep
485 $(sparc_ONLY)driver name=pmc alias=SUNW,pmc
486 $(sparc_ONLY)driver name=pmubus alias=pmu
487 $(sparc_ONLY)driver name=pmugpio
488 driver name=ppm
489 $(i386_ONLY)driver name=privcmd perms="* 0666 root sys"
490 $(sparc_ONLY)driver name=px \
491     alias=SUNW,sun4v-pci \
492     alias=pciex108e,80f0 \
493     alias=pciex108e,80f8
494 $(sparc_ONLY)driver name=qcn alias=SUNW,sun4v-console
495 $(sparc_ONLY)driver name=rmc_comm
496 $(sparc_ONLY)driver name=rmcadm
497 $(sparc_ONLY)driver name=rmclomv
498 $(sparc_ONLY)driver name=rootnex \
499     alias=cpu-unit \
500     alias=io-unit \
501     alias=mem-unit
502 $(i386_ONLY)driver name=rootnex
503 $(sparc_ONLY)driver name=sbbc alias=pci108e,c416
504 $(sparc_ONLY)driver name=sbus class=sbus
505 $(sparc_ONLY)driver name=sbusmem
506 $(sparc_ONLY)driver name=scfd alias=FJSV,scfc
507 $(sparc_ONLY)driver name=schpc
508 $(sparc_ONLY)driver name=schppm alias=gp2-ppm
509 $(sparc_ONLY)driver name=seeprom \
510     alias=i2c-at24c64 \
511     alias=i2c-at34c02
512 $(sparc_ONLY)driver name=sgcn
513 $(sparc_ONLY)driver name=sgenv
514 $(sparc_ONLY)driver name=sgfru
515 $(sparc_ONLY)driver name=sghsc
516 $(sparc_ONLY)driver name=sgsbbc
517 $(sparc_ONLY)driver name=simba alias=pci108e,5000 class=pci
518 $(sparc_ONLY)driver name=simmstat alias=simm-status
519 $(sparc_ONLY)driver name=smbus alias=i2c-smbus
520 $(sparc_ONLY)driver name=smbus_ara alias=i2c-smbus-ara
521 $(sparc_ONLY)driver name=sram
522 $(sparc_ONLY)driver name=ssc050 alias=i2c-ssc050 perms="* 0644 root sys"
```

```
$(sparc_ONLY)driver name=ssc100 alias=i2c-ssc100 perms="* 0644 root sys"
$(sparc_ONLY)driver name=ssm perms="* 0640 root sys"
$(sparc_ONLY)driver name=su perms="[a-z] 0666 root sys" \
    perms="[a-z],cu 0600 uucp uucp" perms="ssp 0600 root sys" \
    perms="sspctl 0600 root sys"
$(sparc_ONLY)driver name=sysctrl alias=clock-board
$(sparc_ONLY)driver name=tda8444 alias=i2c-tda8444
$(sparc_ONLY)driver name=tod perms="* 0600 root sys"
$(sparc_ONLY)driver name=todds1287 alias=ds1287
$(sparc_ONLY)driver name=todds1307
$(sparc_ONLY)driver name=todds1337 alias=i2c-ds1337
$(sparc_ONLY)driver name=trapstat perms="* 0600 root sys"
$(sparc_ONLY)driver name=tsalarm
$(sparc_ONLY)driver name=upa64s \
    alias=SUNW,upa64s \
    alias=jbus-upa64s
$(sparc_ONLY)driver name=vnex \
    alias=SUNW,sun4v-virtual-devices \
    alias=SUNW,virtual-devices
$(sparc_ONLY)driver name=xcalppm alias=ebus-ppm
$(sparc_ONLY)driver name=xcalwd
$(i386_ONLY)driver name=xdb
$(i386_ONLY)driver name=xdf
$(i386_ONLY)driver name=xenbus perms="* 0666 root sys"
$(i386_ONLY)driver name=xencons
$(i386_ONLY)driver name=xnbe alias=xnb,ioemu
$(i386_ONLY)driver name=xnbo \
    alias=xnb \
    alias=xnb,SUNW_mac
$(i386_ONLY)driver name=xnbu alias=xnb,netfront
$(i386_ONLY)driver name=xnf
$(i386_ONLY)driver name=xpvd
$(i386_ONLY)driver name=xpvtap perms="* 0666 root sys"
$(sparc_ONLY)driver name=zs perms="[a-z] 0666 root sys" \
    perms="[a-z],cu 0600 uucp uucp"
$(sparc_ONLY)driver name=zsh perms="* 0666 root sys"
$(sparc_ONLY)file path=platform/SUNW,A70/kernel/drv/ppm.conf group=sys
$(sparc_ONLY)file path=platform/SUNW,A70/kernel/misc/$(ARCH64)/platmod \
    group=sys mode=0755
$(sparc_ONLY)file \
    path=platform/SUNW,Netra-CP2300/kernel/misc/$(ARCH64)/platmod group=sys \
    mode=0755
$(sparc_ONLY)file \
    path=platform/SUNW,Netra-CP2300/kernel/tod/$(ARCH64)/todds1307 group=sys \
    mode=0755
$(sparc_ONLY)file \
    path=platform/SUNW,Netra-CP3010/kernel/misc/$(ARCH64)/platmod group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/SUNW,Netra-T12/kernel/drv/$(ARCH64)/lw8 \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,Netra-T12/kernel/drv/$(ARCH64)/ntwdt \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,Netra-T12/kernel/drv/$(ARCH64)/sgenv \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,Netra-T12/kernel/drv/$(ARCH64)/sgfru \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,Netra-T12/kernel/drv/lw8.conf group=sys
$(sparc_ONLY)file path=platform/SUNW,Netra-T12/kernel/drv/ntwdt.conf group=sys
$(sparc_ONLY)file path=platform/SUNW,Netra-T12/kernel/drv/sgenv.conf group=sys
$(sparc_ONLY)file path=platform/SUNW,Netra-T12/kernel/drv/sgfru.conf group=sys
$(sparc_ONLY)file path=platform/SUNW,Netra-T12/kernel/misc/$(ARCH64)/platmod \
    group=sys mode=0755
$(sparc_ONLY)file path=platform/SUNW,Netra-T4/kernel/drv/$(ARCH64)/lombus \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,Netra-T4/kernel/drv/lombus.conf group=sys
$(sparc_ONLY)file path=platform/SUNW,SPARC-Enterprise/kernel/$(ARCH64)/unix \
```



```
    group=sys mode=0755
$(sparc_ONLY)file \
    path=platform/SUNW,SPARC-Enterprise/kernel/cpu/$(ARCH64)/FJSV,SPARC64-VI \
    group=sys mode=0755
$(sparc_ONLY)file path=platform/SUNW,SPARC-Enterprise/kernel/drv/$(ARCH64)/dr \
    group=sys
$(sparc_ONLY)file \
    path=platform/SUNW,SPARC-Enterprise/kernel/drv/$(ARCH64)/mc-opl group=sys
$(sparc_ONLY)file \
    path=platform/SUNW,SPARC-Enterprise/kernel/drv/$(ARCH64)/oplmsu group=sys
$(sparc_ONLY)file \
    path=platform/SUNW,SPARC-Enterprise/kernel/drv/$(ARCH64)/oplpanel \
    group=sys
$(sparc_ONLY)file \
    path=platform/SUNW,SPARC-Enterprise/kernel/drv/$(ARCH64)/pcicmu group=sys
$(sparc_ONLY)file \
    path=platform/SUNW,SPARC-Enterprise/kernel/drv/$(ARCH64)/scfd group=sys
$(sparc_ONLY)file path=platform/SUNW,SPARC-Enterprise/kernel/drv/dr.conf \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,SPARC-Enterprise/kernel/drv/mc-opl.conf \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,SPARC-Enterprise/kernel/drv/oplpanel.conf \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,SPARC-Enterprise/kernel/drv/options.conf \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,SPARC-Enterprise/kernel/drv/scfd.conf \
    group=sys
$(sparc_ONLY)file \
    path=platform/SUNW,SPARC-Enterprise/kernel/misc/$(ARCH64)/drmach \
    group=sys mode=0755
$(sparc_ONLY)file \
    path=platform/SUNW,SPARC-Enterprise/kernel/misc/$(ARCH64)/platmod \
    group=sys mode=0755
$(sparc_ONLY)file path=platform/SUNW,Serverblade1/kernel/drv/$(ARCH64)/bscbus \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,Serverblade1/kernel/drv/$(ARCH64)/bscv \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,Serverblade1/kernel/drv/bscbus.conf \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,Serverblade1/kernel/drv/bscv.conf \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,Serverblade1/kernel/drv/options.conf \
    group=sys
$(sparc_ONLY)file \
    path=platform/SUNW,Serverblade1/kernel/misc/$(ARCH64)/platmod group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/SUNW,Sun-Blade-100/kernel/drv/$(ARCH64)/grfans \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,Sun-Blade-100/kernel/drv/$(ARCH64)/grppm \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,Sun-Blade-100/kernel/drv/grppm.conf \
    group=sys
$(sparc_ONLY)file \
    path=platform/SUNW,Sun-Blade-100/kernel/misc/$(ARCH64)/platmod group=sys \
    mode=0755
$(sparc_ONLY)file \
    path=platform/SUNW,Sun-Blade-1000/kernel/drv/$(ARCH64)/xcalppm group=sys
$(sparc_ONLY)file \
    path=platform/SUNW,Sun-Blade-1000/kernel/drv/$(ARCH64)/xcalwd group=sys
$(sparc_ONLY)file path=platform/SUNW,Sun-Blade-1000/kernel/drv/xcalppm.conf \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,Sun-Blade-1000/kernel/drv/xcalwd.conf \
    group=sys
$(sparc_ONLY)file \
    path=platform/SUNW,Sun-Blade-1000/kernel/misc/$(ARCH64)/platmod group=sys \
    mode=0755
```

```
655 $(sparc_ONLY)file path=platform/SUNW,Sun-Blade-1500/kernel/drv/ppm.conf \
656     group=sys
657 $(sparc_ONLY)file \
658     path=platform/SUNW,Sun-Blade-1500/kernel/misc/$(ARCH64)/platmod group=sys \
659     mode=0755
660 $(sparc_ONLY)file path=platform/SUNW,Sun-Blade-2500/kernel/drv/ppm.conf \
661     group=sys
662 $(sparc_ONLY)file \
663     path=platform/SUNW,Sun-Blade-2500/kernel/misc/$(ARCH64)/platmod group=sys \
664     mode=0755
665 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire-15000/kernel/$(ARCH64)/unix \
666     group=sys mode=0755
667 $(sparc_ONLY)file \
668     path=platform/SUNW,Sun-Fire-15000/kernel/cpu/$(ARCH64)/SUNW,UltraSPARC-III \
669     group=sys mode=0755
670 $(sparc_ONLY)file \
671     path=platform/SUNW,Sun-Fire-15000/kernel/cpu/$(ARCH64)/SUNW,UltraSPARC-III+
672     group=sys mode=0755
673 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire-15000/kernel/drv/$(ARCH64)/axq \
674     group=sys
675 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire-15000/kernel/drv/$(ARCH64)/dman \
676     group=sys
677 $(sparc_ONLY)file \
678     path=platform/SUNW,Sun-Fire-15000/kernel/drv/$(ARCH64)/iosram group=sys
679 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire-15000/kernel/drv/$(ARCH64)/schpc \
680     group=sys
681 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire-15000/kernel/drv/dman.conf \
682     group=sys
683 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire-15000/kernel/drv/schpc.conf \
684     group=sys
685 $(sparc_ONLY)file \
686     path=platform/SUNW,Sun-Fire-15000/kernel/misc/$(ARCH64)/mboxsc group=sys \
687     mode=0755
688 $(sparc_ONLY)file \
689     path=platform/SUNW,Sun-Fire-15000/kernel/misc/$(ARCH64)/platmod group=sys \
690     mode=0755
691 $(sparc_ONLY)file \
692     path=platform/SUNW,Sun-Fire-15000/kernel/misc/$(ARCH64)/scosmb group=sys \
693     mode=0755
694 $(sparc_ONLY)file \
695     path=platform/SUNW,Sun-Fire-280R/kernel/drv/$(ARCH64)/pcf8574 group=sys
696 $(sparc_ONLY)file \
697     path=platform/SUNW,Sun-Fire-280R/kernel/misc/$(ARCH64)/platmod group=sys \
698     mode=0755
699 $(sparc_ONLY)file \
700     path=platform/SUNW,Sun-Fire-480R/kernel/misc/$(ARCH64)/platmod group=sys \
701     mode=0755
702 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire-880/kernel/drv/$(ARCH64)/hpc3130 \
703     group=sys
704 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire-880/kernel/drv/hpc3130.conf \
705     group=sys
706 $(sparc_ONLY)file \
707     path=platform/SUNW,Sun-Fire-880/kernel/misc/$(ARCH64)/platmod group=sys \
708     mode=0755
709 $(sparc_ONLY)file \
710     path=platform/SUNW,Sun-Fire-T200/kernel/drv/$(ARCH64)/tsalarm group=sys
711 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire-T200/kernel/drv/tsalarm.conf \
712     group=sys
713 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire-V215/kernel/drv/su.conf \
714     group=sys
715 $(sparc_ONLY)file \
716     path=platform/SUNW,Sun-Fire-V215/kernel/misc/$(ARCH64)/platmod group=sys \
717     mode=0755
718 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire-V240/kernel/drv/$(ARCH64)/ntwdt \
719     group=sys
720 $(sparc_ONLY)file \
```



```
721     path=platform/SUNW,Sun-Fire-V240/kernel/drv/$(ARCH64)/tsalarm group=sys
722 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire-V240/kernel/drv/ntwdt.conf \
723     group=sys
724 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire-V240/kernel/drv/tsalarm.conf \
725     group=sys
726 $(sparc_ONLY)file \
727     path=platform/SUNW,Sun-Fire-V240/kernel/misc/$(ARCH64)/platmod group=sys \
728     mode=0755
729 $(sparc_ONLY)file \
730     path=platform/SUNW,Sun-Fire-V250/kernel/misc/$(ARCH64)/platmod group=sys \
731     mode=0755
732 $(sparc_ONLY)file \
733     path=platform/SUNW,Sun-Fire-V440/kernel/misc/$(ARCH64)/platmod group=sys \
734     mode=0755
735 $(sparc_ONLY)file \
736     path=platform/SUNW,Sun-Fire-V445/kernel/misc/$(ARCH64)/platmod group=sys \
737     mode=0755
738 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire/kernel/$(ARCH64)/unix group=sys \
739     mode=0755
740 $(sparc_ONLY)file \
741     path=platform/SUNW,Sun-Fire/kernel/cpu/$(ARCH64)/SUNW,UltraSPARC-III \
742     group=sys mode=0755
743 $(sparc_ONLY)file \
744     path=platform/SUNW,Sun-Fire/kernel/cpu/$(ARCH64)/SUNW,UltraSPARC-III+ \
745     group=sys mode=0755
746 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire/kernel/drv/$(ARCH64)/sgcn \
747     group=sys
748 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire/kernel/drv/$(ARCH64)/sghsc \
749     group=sys
750 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire/kernel/drv/$(ARCH64)/sgsbbc \
751     group=sys
752 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire/kernel/drv/$(ARCH64)/ssm \
753     group=sys
754 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire/kernel/drv/sghsc.conf group=sys
755 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire/kernel/misc/$(ARCH64)/platmod \
756     group=sys mode=0755
757 $(sparc_ONLY)file path=platform/SUNW,Sun-Fire/kernel/misc/$(ARCH64)/sbdp \
758     group=sys mode=0755
759 $(sparc_ONLY)file path=platform/SUNW,Ultra-250/kernel/drv/$(ARCH64)/envctrltwo \
760     group=sys
761 $(sparc_ONLY)file path=platform/SUNW,Ultra-250/kernel/misc/$(ARCH64)/platmod \
762     group=sys mode=0755
763 $(sparc_ONLY)file path=platform/SUNW,Ultra-4/kernel/drv/$(ARCH64)/envctrl \
764     group=sys
765 $(sparc_ONLY)file path=platform/SUNW,Ultra-4/kernel/misc/$(ARCH64)/platmod \
766     group=sys mode=0755
767 $(sparc_ONLY)file path=platform/SUNW,Ultra-5_10/kernel/misc/$(ARCH64)/platmod \
768     group=sys mode=0755
769 $(sparc_ONLY)file \
770     path=platform/SUNW,Ultra-Enterprise-10000/kernel/$(ARCH64)/unix group=sys \
771     mode=0755
772 $(sparc_ONLY)file \
773     path=platform/SUNW,Ultra-Enterprise-10000/kernel/cpu/$(ARCH64)/SUNW,UltraSPA
774     group=sys mode=0755
775 $(sparc_ONLY)file \
776     path=platform/SUNW,Ultra-Enterprise-10000/kernel/drv/$(ARCH64)/pcipsy \
777     group=sys
778 $(sparc_ONLY)file \
779     path=platform/SUNW,Ultra-Enterprise-10000/kernel/drv/$(ARCH64)/rootnex \
780     group=sys
781 $(sparc_ONLY)file \
782     path=platform/SUNW,Ultra-Enterprise-10000/kernel/drv/$(ARCH64)/sbus \
783     group=sys
784 $(sparc_ONLY)file \
785     path=platform/SUNW,Ultra-Enterprise-10000/kernel/misc/$(ARCH64)/platmod \
786     group=sys mode=0755
```

```
$(sparc_ONLY)file path=platform/SUNW,Ultra-Enterprise/kernel/drv/$(ARCH64)/ac \
    group=sys
$(sparc_ONLY)file \
    path=platform/SUNW,Ultra-Enterprise/kernel/drv/$(ARCH64)/central \
    group=sys
$(sparc_ONLY)file \
    path=platform/SUNW,Ultra-Enterprise/kernel/drv/$(ARCH64)/environ \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,Ultra-Enterprise/kernel/drv/$(ARCH64)/fhc \
    group=sys
$(sparc_ONLY)file \
    path=platform/SUNW,Ultra-Enterprise/kernel/drv/$(ARCH64)/simmstat \
    group=sys
$(sparc_ONLY)file \
    path=platform/SUNW,Ultra-Enterprise/kernel/drv/$(ARCH64)/sram group=sys
$(sparc_ONLY)file \
    path=platform/SUNW,Ultra-Enterprise/kernel/drv/$(ARCH64)/sysctrl \
    group=sys
$(sparc_ONLY)file path=platform/SUNW,Ultra-Enterprise/kernel/drv/fhc.conf \
    group=sys
$(sparc_ONLY)file \
    path=platform/SUNW,Ultra-Enterprise/kernel/misc/$(ARCH64)/platmod \
    group=sys mode=0755
$(sparc_ONLY)file path=platform/SUNW,UltraAX-i2/kernel/misc/$(ARCH64)/platmod \
    group=sys mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/$(ARCH64)/unix group=sys mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/cpu/$(ARCH64)/cpu.generic \
    group=sys mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/cpu/$(ARCH64)/cpu_ms.AuthenticAMD \
    group=sys mode=0755
$(i386_ONLY)file \
    path=platform/i86pc/kernel/cpu/$(ARCH64)/cpu_ms.AuthenticAMD.15 group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/cpu/$(ARCH64)/cpu_ms.GenuineIntel \
    group=sys mode=0755
$(i386_ONLY)file \
    path=platform/i86pc/kernel/cpu/$(ARCH64)/cpu_ms.GenuineIntel.6.46 \
    group=sys mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/cpu/cpu.generic group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/cpu/cpu_ms.AuthenticAMD group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/cpu/cpu_ms.AuthenticAMD.15 \
    group=sys mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/cpu/cpu_ms.GenuineIntel group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/cpu/cpu_ms.GenuineIntel.6.46 \
    group=sys mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/dacf/$(ARCH64)/consconfig_dacf \
    group=sys mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/dacf/consconfig_dacf group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/drv/$(ARCH64)/acpinex group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/$(ARCH64)/acpippm group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/$(ARCH64)/amd_iommu group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/$(ARCH64)/cpudrv group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/$(ARCH64)/isa group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/$(ARCH64)/npe group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/$(ARCH64)/pci group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/$(ARCH64)/pit_beep group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/$(ARCH64)/ppm group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/$(ARCH64)/rootnex group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/acpinex group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/acpippm group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/acpippm.conf group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/amd_iommu group=sys
```



```
$(i386_ONLY)file path=platform/i86pc/kernel/drv/amd_iommu.conf group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/cpudrv group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/isa group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/npe group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/pci group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/pit_beep group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/pit_beep.conf group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/ppm group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/ppm.conf group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/rootnex group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/drv/rootnex.conf group=sys
$(i386_ONLY)file path=platform/i86pc/kernel/mach/$(ARCH64)/apix group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/mach/$(ARCH64)/pcplusmp group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/mach/$(ARCH64)/uppc group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/mach/apix group=sys mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/mach/pcplusmp group=sys mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/mach/uppc group=sys mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/misc/$(ARCH64)/acpidev group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/misc/$(ARCH64)/gfx_private \
    group=sys mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/misc/acpidev group=sys mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/misc/gfx_private group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86pc/kernel/unix group=sys mode=0755
$(i386_ONLY)file path=platform/i86pc/multiboot group=sys mode=0755 \
    reboot-needed=true
$(i386_ONLY)file path=platform/i86pc/ucode/amd-ucode.bin group=sys mode=0444 \
    original_name=SUNWcakr:platform/i86pc/ucode/amd-ucode.bin preserve=true \
    reboot-needed=true
$(i386_ONLY)file path=platform/i86pc/ucode/intel-ucode.txt group=sys mode=0444 \
    original_name=SUNWcakr:platform/i86pc/ucode/intel-ucode.txt preserve=true \
    reboot-needed=true
$(i386_ONLY)file path=platform/i86xpv/kernel/$(ARCH64)/unix group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/cpu/$(ARCH64)/cpu.generic \
    group=sys mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/cpu/$(ARCH64)/cpu_ms.AuthenticAMD \
    group=sys mode=0755
$(i386_ONLY)file \
    path=platform/i86xpv/kernel/cpu/$(ARCH64)/cpu_ms.AuthenticAMD.15 \
    group=sys mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/cpu/$(ARCH64)/cpu_ms.GenuineIntel \
    group=sys mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/cpu/cpu.generic group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/cpu/cpu_ms.AuthenticAMD group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/cpu/cpu_ms.AuthenticAMD.15 \
    group=sys mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/cpu/cpu_ms.GenuineIntel group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/dacf/$(ARCH64)/consconfig_dacf \
    group=sys mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/dacf/consconfig_dacf group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/balloon group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/domcaps group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/evtchn group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/isa group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/npe group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/pci group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/pit_beep group=sys
```

```
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/privcmd group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/rootnex group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/xdb group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/xdf group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/xenbus group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/xencons group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/xnbe group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/xnbo group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/xnbu group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/xnf group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/xpvd group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/$(ARCH64)/xpvtap group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/balloon group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/domcaps group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/evtchn group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/isa group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/npe group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/pci group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/pit_beep group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/pit_beep.conf group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/privcmd group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/rootnex group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/xdf group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/xenbus group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/xencons group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/xencons.conf group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/xnf group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/drv/xpvd group=sys
$(i386_ONLY)file path=platform/i86xpv/kernel/mach/$(ARCH64)/xpv_psm group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/mach/$(ARCH64)/xpv_uppc group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/mach/xpv_psm group=sys mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/mach/xpv_uppc group=sys mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/misc/$(ARCH64)/gfx_private \
    group=sys mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/misc/$(ARCH64)/xnb group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/misc/$(ARCH64)/xpv_autoconfig \
    group=sys mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/misc/gfx_private group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/misc/xpv_autoconfig group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/tod/$(ARCH64)/xpvtod group=sys \
    mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/tod/xpvtod group=sys mode=0755
$(i386_ONLY)file path=platform/i86xpv/kernel/unix group=sys mode=0755
$(sparc_ONLY)file path=platform/sun4u-us3/kernel/crypto/$(ARCH64)/aes \
    group=sys mode=0755
$(sparc_ONLY)file path=platform/sun4u/bootlst group=sys reboot-needed=true
$(sparc_ONLY)file path=platform/sun4u/kernel/$(ARCH64)/genunix group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/$(ARCH64)/unix group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/cpu/$(ARCH64)/SUNW,UltraSPARC-II \
    group=sys mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/cpu/$(ARCH64)/SUNW,UltraSPARC-III \
    group=sys mode=0755
$(sparc_ONLY)file \
    path=platform/sun4u/kernel/cpu/$(ARCH64)/SUNW,UltraSPARC-III+ group=sys \
    mode=0755
$(sparc_ONLY)file \
    path=platform/sun4u/kernel/cpu/$(ARCH64)/SUNW,UltraSPARC-IIIi group=sys \
    mode=0755
$(sparc_ONLY)file \
```



```
    path=platform/sun4u/kernel/cpu/$(ARCH64)/SUNW,UltraSPARC-IIIi+ group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/cpu/$(ARCH64)/SUNW,UltraSPARC-IIe \
    group=sys mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/crypto/$(ARCH64)/arcfour \
    group=sys mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/crypto/$(ARCH64)/des group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/crypto/$(ARCH64)/md5 group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/crypto/$(ARCH64)/sha1 group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/adm1026 group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/adm1031 group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/bbc_beep group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/ctsmc group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/db21554 group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/ebus group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/epic group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/fd group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/gpio_87317 \
    group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/grbeep group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/i2bsc group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/ics951601 group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/isadma group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/jbusppm group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/lm75 group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/ltc1427 group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/m1535ppm group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/max1617 group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/mc-us3 group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/mc-us3i group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/mem_cache group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/mi2cv group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/pca9556 group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/pcf8574 group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/pcf8584 group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/pcf8591 group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/pcipsy group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/pcisch group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/pic16f747 group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/pic16f819 group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/pmc group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/pmubus group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/pmugpio group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/power group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/ppm group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/px group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/rmc_comm group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/rmcadm group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/rmclomv group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/rootnex group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/sbbc group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/sbus group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/sbusmem group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/schppm group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/seeprom group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/simba group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/smbus group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/smbus_ara group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/ssc050 group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/ssc100 group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/su group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/tda8444 group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/trapstat group=sys
```

```
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/upa64s group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/zs group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/$(ARCH64)/zsh group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/i2bsc.conf group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/mem_cache.conf group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/rmc_comm.conf group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/rmcadm.conf group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/rmclomv.conf group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/sbusmem.conf group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/trapstat.conf group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/drv/zsh.conf group=sys
$(sparc_ONLY)file path=platform/sun4u/kernel/misc/$(ARCH64)/bignum group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/misc/$(ARCH64)/bootdev group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/misc/$(ARCH64)/forthdebug \
    group=sys mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/misc/$(ARCH64)/i2c_svc group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/misc/$(ARCH64)/kmdbmod group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/misc/$(ARCH64)/obpsym group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/misc/$(ARCH64)/opl_cfg group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/misc/$(ARCH64)/pcie group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/misc/$(ARCH64)/platmod group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/misc/$(ARCH64)/sbd group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/misc/$(ARCH64)/vis group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/misc/$(ARCH64)/zuluvm group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/strmod/$(ARCH64)/kb group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/tod/$(ARCH64)/todblade group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/tod/$(ARCH64)/todbq4802 group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/tod/$(ARCH64)/todds1287 group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/tod/$(ARCH64)/todds1337 group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/tod/$(ARCH64)/todm5819 group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/tod/$(ARCH64)/todm5819p_rmc \
    group=sys mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/tod/$(ARCH64)/todm5823 group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/tod/$(ARCH64)/todmostek group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/tod/$(ARCH64)/todopl group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/tod/$(ARCH64)/todsg group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/tod/$(ARCH64)/todstarcat \
    group=sys mode=0755
$(sparc_ONLY)file path=platform/sun4u/kernel/tod/$(ARCH64)/todstarfire \
    group=sys mode=0755
$(sparc_ONLY)file path=platform/sun4v/bootlst group=sys reboot-needed=true
$(sparc_ONLY)file path=platform/sun4v/kernel/$(ARCH64)/genunix group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4v/kernel/$(ARCH64)/unix group=sys \
    mode=0755
```



```
$(sparc_ONLY)file path=platform/sun4v/kernel/cpu/$(ARCH64)/generic group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4v/kernel/crypto/$(ARCH64)/arcfour \
    group=sys mode=0755
$(sparc_ONLY)file path=platform/sun4v/kernel/crypto/$(ARCH64)/md5 group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/$(ARCH64)/bge group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/$(ARCH64)/bmc group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/$(ARCH64)/ebus group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/$(ARCH64)/glvc group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/$(ARCH64)/mdesc group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/$(ARCH64)/n2rng group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/$(ARCH64)/ntwdt group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/$(ARCH64)/px group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/$(ARCH64)/qcn group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/$(ARCH64)/rootnex group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/$(ARCH64)/su group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/$(ARCH64)/trapstat group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/$(ARCH64)/vnex group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/bge.conf group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/bmc.conf group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/mdesc.conf group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/n2rng.conf group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/ntwdt.conf group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/drv/trapstat.conf group=sys
$(sparc_ONLY)file path=platform/sun4v/kernel/misc/$(ARCH64)/bootdev group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4v/kernel/misc/$(ARCH64)/forthdebug \
    group=sys mode=0755
$(sparc_ONLY)file path=platform/sun4v/kernel/misc/$(ARCH64)/kmdbmod group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4v/kernel/misc/$(ARCH64)/obpsym group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4v/kernel/misc/$(ARCH64)/pcie group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4v/kernel/misc/$(ARCH64)/platmod group=sys \
    mode=0755
$(sparc_ONLY)file path=platform/sun4v/kernel/misc/$(ARCH64)/vis group=sys \
    mode=0755
$(sparc_ONLY)file path=usr/share/man/man4/sbus.4
$(i386_ONLY)file path=usr/share/man/man4/sysbus.4
$(sparc_ONLY)file path=usr/share/man/man7d/bbc_beep.7d
$(sparc_ONLY)file path=usr/share/man/man7d/ctsmc.7d
$(sparc_ONLY)file path=usr/share/man/man7d/dr.7d
$(sparc_ONLY)file path=usr/share/man/man7d/fd.7d
$(sparc_ONLY)file path=usr/share/man/man7d/gpio_87317.7d
$(sparc_ONLY)file path=usr/share/man/man7d/grbeep.7d
$(sparc_ONLY)file path=usr/share/man/man7d/mc-opl.7d
$(sparc_ONLY)file path=usr/share/man/man7d/n2rng.7d
$(sparc_ONLY)file path=usr/share/man/man7d/ncp.7d
$(i386_ONLY)file path=usr/share/man/man7d/npe.7d
$(sparc_ONLY)file path=usr/share/man/man7d/ntwdt.7d
$(sparc_ONLY)file path=usr/share/man/man7d/oplmsu.7d
$(sparc_ONLY)file path=usr/share/man/man7d/oplpanel.7d
$(sparc_ONLY)file path=usr/share/man/man7d/pcicmu.7d
$(sparc_ONLY)file path=usr/share/man/man7d/pcipsy.7d
$(sparc_ONLY)file path=usr/share/man/man7d/pcisch.7d
$(sparc_ONLY)file path=usr/share/man/man7d/schpc.7d
$(sparc_ONLY)file path=usr/share/man/man7d/smbus.7d
$(sparc_ONLY)file path=usr/share/man/man7d/su.7d
$(sparc_ONLY)file path=usr/share/man/man7d/todopl.7d
$(sparc_ONLY)file path=usr/share/man/man7d/tsalarm.7d
$(sparc_ONLY)file path=usr/share/man/man7d/zs.7d
$(sparc_ONLY)file path=usr/share/man/man7d/zsh.7d
$(sparc_ONLY)file path=usr/share/man/man7m/kb.7m
$(i386_ONLY)hardlink \
```

```
1181     path=platform/i86pc/kernel/cpu/$(ARCH64)/cpu_ms.GenuineIntel.6.47 \
1182     target=cpu_ms.GenuineIntel.6.46
1183 $(i386_ONLY)hardlink path=platform/i86pc/kernel/cpu/cpu_ms.GenuineIntel.6.47 \
1184     target=cpu_ms.GenuineIntel.6.46
1185 $(sparc_ONLY)hardlink path=platform/sun4u/kernel/misc/$(ARCH64)/md5 \
1186     target=../../../kernel/crypto/$(ARCH64)/md5
1187 $(sparc_ONLY)hardlink path=platform/sun4u/kernel/misc/$(ARCH64)/sha1 \
1188     target=../../../kernel/crypto/$(ARCH64)/sha1
1189 $(sparc_ONLY)hardlink path=platform/sun4v/kernel/misc/$(ARCH64)/md5 \
1190     target=../../../kernel/crypto/$(ARCH64)/md5
1191 $(i386_ONLY)legacy pkg=SUNWcakr.i arch=$(ARCH).i86pc \
1192     desc="core kernel software for a specific hardware platform group" \
1193     name="Core Solaris Kernel Architecture (Root)"
1194 $(sparc_ONLY)legacy pkg=SUNWcakr.u arch=$(ARCH).sun4u \
1195     desc="core kernel software for a specific hardware platform group" \
1196     name="Core Solaris Kernel Architecture (Root)"
1197 $(sparc_ONLY)legacy pkg=SUNWcakr.v arch=$(ARCH).sun4v \
1198     desc="core kernel software for a specific hardware platform group" \
1199     name="Core Solaris Kernel Architecture (Root)"
1200 $(sparc_ONLY)legacy pkg=SUNWcakrnt2000.v arch=$(ARCH).sun4v \
1201     desc="driver software for the Netra-T2000 hardware platform" \
1202     name="Platform Specific Drivers for Netra-T2000, (Root)"
1203 $(i386_ONLY)legacy pkg=SUNWcakrx.i arch=$(ARCH).i86pc \
1204     desc="core kernel software for the i86xpv virtual hardware platform" \
1205     name="Core Kernel Architecture i86xpv, (Root)"
1206 license cr_Sun license=cr_Sun
1207 license lic_CDDL license=lic_CDDL
1208 license usr/src/cmd/mdb/common/libstand/THIRDPARTYLICENSE \
1209     license=usr/src/cmd/mdb/common/libstand/THIRDPARTYLICENSE
1210 license usr/src/common/bzip2/LICENSE license=usr/src/common/bzip2/LICENSE
1211 $(sparc_ONLY)license usr/src/stand/lib/tcp/THIRDPARTYLICENSE \
1212     license=usr/src/stand/lib/tcp/THIRDPARTYLICENSE
1213 license usr/src/uts/common/sys/THIRDPARTYLICENSE.unicode \
1214     license=usr/src/uts/common/sys/THIRDPARTYLICENSE.unicode
1215 license usr/src/uts/common/zmod/THIRDPARTYLICENSE \
1216     license=usr/src/uts/common/zmod/THIRDPARTYLICENSE
1217 $(i386_ONLY)license usr/src/uts/intel/THIRDPARTYLICENSE \
1218     license=usr/src/uts/intel/THIRDPARTYLICENSE
1219 $(sparc_ONLY)link path=platform/SUNW,A70/kernel/crypto/$(ARCH64)/aes \
1220     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
1221 $(sparc_ONLY)link path=platform/SUNW,Netra-CP3010/kernel/crypto/$(ARCH64)/aes \
1222     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
1223 $(sparc_ONLY)link path=platform/SUNW,Netra-T12/kernel/$(ARCH64) \
1224     target=../../SUNW,Sun-Fire/kernel/$(ARCH64)
1225 $(sparc_ONLY)link path=platform/SUNW,Netra-T12/kernel/cpu \
1226     target=../../SUNW,Sun-Fire/kernel/cpu
1227 $(sparc_ONLY)link path=platform/SUNW,Netra-T12/kernel/crypto/$(ARCH64)/aes \
1228     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
1229 $(sparc_ONLY)link path=platform/SUNW,Netra-T12/kernel/drv/$(ARCH64)/sgcn \
1230     target=../../../../SUNW,Sun-Fire/kernel/drv/$(ARCH64)/sgcn
1231 $(sparc_ONLY)link path=platform/SUNW,Netra-T12/kernel/drv/$(ARCH64)/sgsbbc \
1232     target=../../../../SUNW,Sun-Fire/kernel/drv/$(ARCH64)/sgsbbc
1233 $(sparc_ONLY)link path=platform/SUNW,Netra-T12/kernel/drv/$(ARCH64)/ssm \
1234     target=../../../../SUNW,Sun-Fire/kernel/drv/$(ARCH64)/ssm
1235 $(sparc_ONLY)link path=platform/SUNW,Netra-T12/kernel/misc/$(ARCH64)/sbdp \
1236     target=../../../../SUNW,Sun-Fire/kernel/misc/$(ARCH64)/sbdp
1237 $(sparc_ONLY)link path=platform/SUNW,Netra-T4/kernel/crypto/$(ARCH64)/aes \
1238     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
1239 $(sparc_ONLY)link path=platform/SUNW,Netra-T4/kernel/misc/$(ARCH64)/platmod \
1240     target=../../../../SUNW,Sun-Fire-280R/kernel/misc/$(ARCH64)/platmod
1241 $(sparc_ONLY)link \
1242     path=platform/SUNW,SPARC-Enterprise/kernel/cpu/$(ARCH64)/FJSV,SPARC64-VII \
1243     target=FJSV,SPARC64-VI
1244 $(sparc_ONLY)link \
1245     path=platform/SUNW,SPARC-Enterprise/kernel/crypto/$(ARCH64)/aes \
1246     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
```



```
1247 $(sparc_ONLY)link \
1248     path=platform/SUNW,Sun-Blade-1000/kernel/crypto/$(ARCH64)/aes \
1249     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
1250 $(sparc_ONLY)link \
1251     path=platform/SUNW,Sun-Blade-1500/kernel/crypto/$(ARCH64)/aes \
1252     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
1253 $(sparc_ONLY)link \
1254     path=platform/SUNW,Sun-Blade-2500/kernel/crypto/$(ARCH64)/aes \
1255     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
1256 $(sparc_ONLY)link \
1257     path=platform/SUNW,Sun-Fire-15000/kernel/cpu/$(ARCH64)/SUNW,UltraSPARC-IV \
1258     target=SUNW,UltraSPARC-III+
1259 $(sparc_ONLY)link \
1260     path=platform/SUNW,Sun-Fire-15000/kernel/cpu/$(ARCH64)/SUNW,UltraSPARC-IV+ \
1261     target=SUNW,UltraSPARC-III+
1262 $(sparc_ONLY)link \
1263     path=platform/SUNW,Sun-Fire-15000/kernel/crypto/$(ARCH64)/aes \
1264     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
1265 $(sparc_ONLY)link path=platform/SUNW,Sun-Fire-280R/kernel/crypto/$(ARCH64)/aes \
1266     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
1267 $(sparc_ONLY)link path=platform/SUNW,Sun-Fire-480R/kernel/crypto/$(ARCH64)/aes \
1268     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
1269 $(sparc_ONLY)link path=platform/SUNW,Sun-Fire-880/kernel/crypto/$(ARCH64)/aes \
1270     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
1271 $(sparc_ONLY)link path=platform/SUNW,Sun-Fire-V215/kernel/crypto/$(ARCH64)/aes \
1272     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
1273 $(sparc_ONLY)link path=platform/SUNW,Sun-Fire-V240/kernel/crypto/$(ARCH64)/aes \
1274     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
1275 $(sparc_ONLY)link path=platform/SUNW,Sun-Fire-V250/kernel/crypto/$(ARCH64)/aes \
1276     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
1277 $(sparc_ONLY)link path=platform/SUNW,Sun-Fire-V440/kernel/crypto/$(ARCH64)/aes \
1278     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
1279 $(sparc_ONLY)link path=platform/SUNW,Sun-Fire-V440/kernel/drv/$(ARCH64)/ntwdt \
1280     target=../../../../SUNW,Sun-Fire-V240/kernel/drv/$(ARCH64)/ntwdt
1281 $(sparc_ONLY)link \
1282     path=platform/SUNW,Sun-Fire-V440/kernel/drv/$(ARCH64)/tsalarm \
1283     target=../../../../SUNW,Sun-Fire-V240/kernel/drv/$(ARCH64)/tsalarm
1284 $(sparc_ONLY)link path=platform/SUNW,Sun-Fire-V440/kernel/drv/ntwdt.conf \
1285     target=../../../SUNW,Sun-Fire-V240/kernel/drv/ntwdt.conf
1286 $(sparc_ONLY)link path=platform/SUNW,Sun-Fire-V440/kernel/drv/tsalarm.conf \
1287     target=../../../SUNW,Sun-Fire-V240/kernel/drv/tsalarm.conf
1288 $(sparc_ONLY)link path=platform/SUNW,Sun-Fire-V445/kernel/crypto/$(ARCH64)/aes \
1289     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
1290 $(sparc_ONLY)link \
1291     path=platform/SUNW,Sun-Fire/kernel/cpu/$(ARCH64)/SUNW,UltraSPARC-IV \
1292     target=SUNW,UltraSPARC-III+
1293 $(sparc_ONLY)link \
1294     path=platform/SUNW,Sun-Fire/kernel/cpu/$(ARCH64)/SUNW,UltraSPARC-IV+ \
1295     target=SUNW,UltraSPARC-III+
1296 $(sparc_ONLY)link path=platform/SUNW,Sun-Fire/kernel/crypto/$(ARCH64)/aes \
1297     target=../../../../sun4u-us3/kernel/crypto/$(ARCH64)/aes
1298 $(sparc_ONLY)link path=platform/SUNW,Ultra-80/kernel/misc/$(ARCH64)/platmod \
1299     target=../../../../SUNW,Ultra-5_10/kernel/misc/$(ARCH64)/platmod
1300 $(sparc_ONLY)link path=platform/SUNW,Ultra-Enterprise-10000/kernel/unix \
1301     target=$(ARCH64)/unix
1302 $(sparc_ONLY)link path=platform/sun4u/kernel/cpu/$(ARCH64)/SUNW,UltraSPARC-IIi \
1303     target=SUNW,UltraSPARC-II
1304 $(sparc_ONLY)link path=platform/sun4u/kernel/cpu/$(ARCH64)/SUNW,UltraSPARC-IV \
1305     target=SUNW,UltraSPARC-III+
1306 $(sparc_ONLY)link path=platform/sun4u/kernel/cpu/$(ARCH64)/SUNW,UltraSPARC-IV+ \
1307     target=SUNW,UltraSPARC-III+
1308 $(sparc_ONLY)link path=platform/sun4u/kernel/misc/$(ARCH64)/des \
1309     target=../../../kernel/crypto/$(ARCH64)/des
1310 $(sparc_ONLY)link path=platform/sun4u/kernel/unix target=$(ARCH64)/unix
1311 $(sparc_ONLY)link path=platform/sun4v/kernel/cpu/$(ARCH64)/sun4v \
1312     target=generic
```

```
1313 $(sparc_ONLY)link path=platform/sun4v/kernel/unix target=$(ARCH64)/unix
1314 $(i386_ONLY)link path=usr/share/man/man4/isa.4 target=sysbus.4
1315 $(sparc_ONLY)link path=usr/share/man/man7d/drmach.7d target=dr.7d
1316 link path=usr/share/man/man7d/fdc.7d target=fd.7d
1317 $(sparc_ONLY)link path=usr/share/man/man7d/ngdr.7d target=dr.7d
1318 $(sparc_ONLY)link path=usr/share/man/man7d/ngdrmach.7d target=dr.7d
```

```
**********************************************************
   44750 Fri May  8 18:10:22 2015
new/usr/src/uts/common/Makefile.files
remove xhat
The xhat infrastructure was added to support hardware such as the zulu
graphics card - hardware which had on-board MMUs.  The VM used the xhat code
to keep the CPU's and Zulu's page tables in-sync.  Since the only xhat user
was zulu (which is gone), we can safely remove it simplifying the whole VM
subsystem.
Assorted notes:
- AS_BUSY flag was used solely by xhat
**********************************************************
#
# 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) 2013 by Delphix. All rights reserved.
# Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
# Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
#

#
# This Makefile defines all file modules for the directory uts/common
# and its children. These are the source files which may be considered
# common to all SunOS systems.

i386_CORE_OBJS += \
                atomic.o        \
                avintr.o        \
                pic.o

sparc_CORE_OBJS +=

COMMON_CORE_OBJS +=             \
                beep.o          \
                bitset.o        \
                bp_map.o        \
                brand.o         \
                cpucaps.o       \
                cmt.o           \
                cmt_policy.o    \
                cpu.o           \
                cpu_event.o     \
                cpu_intr.o      \
                cpu_pm.o        \
                cpupart.o       \
                cap_util.o      \
```



```
                disp.o          \
                group.o         \
                kstat_fr.o      \
                iscsiboot_prop.o        \
                lgrp.o          \
                lgrp_topo.o     \
                mmapobj.o       \
                mutex.o         \
                page_lock.o     \
                page_retire.o   \
                panic.o         \
                param.o         \
                pg.o            \
                pghw.o          \
                putnext.o       \
                rctl_proc.o     \
                rwlock.o        \
                seg_kmem.o      \
                softint.o       \
                string.o        \
                strtol.o        \
                strtoul.o       \
                strtoll.o       \
                strtoull.o      \
                thread_intr.o   \
                vm_page.o       \
                vm_pagelist.o   \
                zlib_obj.o      \
                clock_tick.o

CORE_OBJS +=    $(COMMON_CORE_OBJS) $($(MACH)_CORE_OBJS)

ZLIB_OBJS =     zutil.o zmod.o zmod_subr.o \
                adler32.o crc32.o deflate.o inffast.o \
                inflate.o inftrees.o trees.o

GENUNIX_OBJS += \
                access.o        \
                acl.o           \
                acl_common.o    \
                adjtime.o       \
                alarm.o         \
                aio_subr.o      \
                auditsys.o      \
                audit_core.o    \
                audit_zone.o    \
                audit_memory.o  \
                autoconf.o      \
                avl.o           \
                bdev_dsort.o    \
                bio.o           \
                bitmap.o        \
                blabel.o        \
                brandsys.o      \
                bz2blocksort.o  \
                bz2compress.o   \
                bz2decompress.o \
                bz2randtable.o  \
                bz2bzlib.o      \
                bz2crctable.o   \
                bz2huffman.o    \
                callb.o         \
                callout.o       \
                chdir.o         \
                chmod.o         \
                chown.o         \
```

```
121                 cladm.o         \
122                 class.o         \
123                 clock.o         \
124                 clock_highres.o \
125                 clock_realtime.o\
126                 close.o         \
127                 compress.o      \
128                 condvar.o       \
129                 conf.o          \
130                 console.o       \
131                 contract.o      \
132                 copyops.o       \
133                 core.o          \
134                 corectl.o       \
135                 cred.o          \
136                 cs_stubs.o      \
137                 dacf.o          \
138                 dacf_clnt.o     \
139                 damap.o \
140                 cyclic.o        \
141                 ddi.o           \
142                 ddifm.o         \
143                 ddi_hp_impl.o   \
144                 ddi_hp_ndi.o    \
145                 ddi_intr.o      \
146                 ddi_intr_impl.o \
147                 ddi_intr_irm.o  \
148                 ddi_nodeid.o    \
149                 ddi_periodic.o  \
150                 devcfg.o        \
151                 devcache.o      \
152                 device.o        \
153                 devid.o         \
154                 devid_cache.o   \
155                 devid_scsi.o    \
156                 devid_smp.o     \
157                 devpolicy.o     \
158                 disp_lock.o     \
159                 dnlc.o          \
160                 driver.o        \
161                 dumpsubr.o      \
162                 driver_lyr.o    \
163                 dtrace_subr.o   \
164                 errorq.o        \
165                 etheraddr.o     \
166                 evchannels.o    \
167                 exacct.o        \
168                 exacct_core.o   \
169                 exec.o          \
170                 exit.o          \
171                 fbio.o          \
172                 fcntl.o         \
173                 fdbuffer.o      \
174                 fdsync.o        \
175                 fem.o           \
176                 ffs.o           \
177                 fio.o           \
178                 flock.o         \
179                 fm.o            \
180                 fork.o          \
181                 vpm.o           \
182                 fs_reparse.o    \
183                 fs_subr.o       \
184                 fsflush.o       \
185                 ftrace.o        \
186                 getcwd.o        \
```



```
187                 getdents.o      \
188                 getloadavg.o    \
189                 getpagesizes.o  \
190                 getpid.o        \
191                 gfs.o           \
192                 rusagesys.o     \
193                 gid.o           \
194                 groups.o        \
195                 grow.o          \
196                 hat_refmod.o    \
197                 id32.o          \
198                 id_space.o      \
199                 inet_ntop.o     \
200                 instance.o      \
201                 ioctl.o         \
202                 ip_cksum.o      \
203                 issetugid.o     \
204                 ippconf.o       \
205                 kcpc.o          \
206                 kdi.o           \
207                 kiconv.o        \
208                 klpd.o          \
209                 kmem.o          \
210                 ksyms_snapshot.o        \
211                 l_strplumb.o    \
212                 labelsys.o      \
213                 link.o          \
214                 list.o          \
215                 lockstat_subr.o \
216                 log_sysevent.o  \
217                 logsubr.o       \
218                 lookup.o        \
219                 lseek.o         \
220                 ltos.o          \
221                 lwp.o           \
222                 lwp_create.o    \
223                 lwp_info.o      \
224                 lwp_self.o      \
225                 lwp_sobj.o      \
226                 lwp_timer.o     \
227                 lwpsys.o        \
228                 main.o          \
229                 mmapobjsys.o    \
230                 memcntl.o       \
231                 memstr.o        \
232                 lgrpsys.o       \
233                 mkdir.o         \
234                 mknod.o         \
235                 mount.o         \
236                 move.o          \
237                 msacct.o        \
238                 multidata.o     \
239                 nbmlock.o       \
240                 ndifm.o         \
241                 nice.o          \
242                 netstack.o      \
243                 ntptime.o       \
244                 nvpair.o        \
245                 nvpair_alloc_system.o   \
246                 nvpair_alloc_fixed.o    \
247                 fnvpair.o       \
248                 octet.o         \
249                 open.o          \
250                 p_online.o      \
251                 pathconf.o      \
252                 pathname.o      \
```

```
253                 pause.o         \
254                 serializer.o    \
255                 pci_intr_lib.o  \
256                 pci_cap.o       \
257                 pcifm.o         \
258                 pgrp.o          \
259                 pgrpsys.o       \
260                 pid.o           \
261                 pkp_hash.o      \
262                 policy.o        \
263                 poll.o          \
264                 pool.o          \
265                 pool_pset.o     \
266                 port_subr.o     \
267                 ppriv.o         \
268                 printf.o        \
269                 priocntl.o      \
270                 priv.o          \
271                 priv_const.o    \
272                 proc.o          \
273                 procset.o       \
274                 processor_bind.o        \
275                 processor_info.o        \
276                 profil.o        \
277                 project.o       \
278                 qsort.o         \
279                 getrandom.o     \
280                 rctl.o          \
281                 rctlsys.o       \
282                 readlink.o      \
283                 refstr.o        \
284                 rename.o        \
285                 resolvepath.o   \
286                 retire_store.o  \
287                 process.o       \
288                 rlimit.o        \
289                 rmap.o          \
290                 rw.o            \
291                 rwstlock.o      \
292                 sad_conf.o      \
293                 sid.o           \
294                 sidsys.o        \
295                 sched.o         \
296                 schedctl.o      \
297                 sctp_crc32.o    \
298                 seg_dev.o       \
299                 seg_kp.o        \
300                 seg_kpm.o       \
301                 seg_map.o       \
302                 seg_vn.o        \
303                 seg_spt.o       \
304                 semaphore.o     \
305                 sendfile.o      \
306                 session.o       \
307                 share.o         \
308                 shuttle.o       \
309                 sig.o           \
310                 sigaction.o     \
311                 sigaltstack.o   \
312                 signotify.o     \
313                 sigpending.o    \
314                 sigprocmask.o   \
315                 sigqueue.o      \
316                 sigsendset.o    \
317                 sigsuspend.o    \
318                 sigtimedwait.o  \
```



```
319                 sleepq.o        \
320                 sock_conf.o     \
321                 space.o         \
322                 sscanf.o        \
323                 stat.o          \
324                 statfs.o        \
325                 statvfs.o       \
326                 stol.o          \
327                 str_conf.o      \
328                 strcalls.o      \
329                 stream.o        \
330                 streamio.o      \
331                 strext.o        \
332                 strsubr.o       \
333                 strsun.o        \
334                 subr.o          \
335                 sunddi.o        \
336                 sunmdi.o        \
337                 sunndi.o        \
338                 sunpci.o        \
339                 sunpm.o         \
340                 sundlpi.o       \
341                 suntpi.o        \
342                 swap_subr.o     \
343                 swap_vnops.o    \
344                 symlink.o       \
345                 sync.o          \
346                 sysclass.o      \
347                 sysconfig.o     \
348                 sysent.o        \
349                 sysfs.o         \
350                 systeminfo.o    \
351                 task.o          \
352                 taskq.o         \
353                 tasksys.o       \
354                 time.o          \
355                 timer.o         \
356                 times.o         \
357                 timers.o        \
358                 thread.o        \
359                 tlabel.o        \
360                 tnf_res.o       \
361                 turnstile.o     \
362                 tty_common.o    \
363                 u8_textprep.o   \
364                 uadmin.o        \
365                 uconv.o         \
366                 ucredsys.o      \
367                 uid.o           \
368                 umask.o         \
369                 umount.o        \
370                 uname.o         \
371                 unix_bb.o       \
372                 unlink.o        \
373                 urw.o           \
374                 utime.o         \
375                 utssys.o        \
376                 uucopy.o        \
377                 vfs.o           \
378                 vfs_conf.o      \
379                 vmem.o          \
380                 vm_anon.o       \
381                 vm_as.o         \
382                 vm_meter.o      \
383                 vm_pageout.o    \
384                 vm_pvn.o        \
```

```
385                 vm_rm.o         \
386                 vm_seg.o        \
387                 vm_subr.o       \
388                 vm_swap.o       \
389                 vm_usage.o      \
390                 vnode.o         \
391                 vuid_queue.o    \
392                 vuid_store.o    \
393                 waitq.o         \
394                 watchpoint.o    \
395                 yield.o         \
396                 scsi_confdata.o \
397                 xattr.o         \
398                 xattr_common.o  \
399                 xdr_mblk.o      \
400                 xdr_mem.o       \
401                 xdr.o           \
402                 xdr_array.o     \
403                 xdr_refer.o     \
404                 xhat.o          \
404                 zone.o

406 #
407 #       Stubs for the stand-alone linker/loader
408 #
409 sparc_GENSTUBS_OBJS =   \
410         kobj_stubs.o

412 i386_GENSTUBS_OBJS =

414 COMMON_GENSTUBS_OBJS =

416 GENSTUBS_OBJS += $(COMMON_GENSTUBS_OBJS) $($(MACH)_GENSTUBS_OBJS)

418 #
419 #       DTrace and DTrace Providers
420 #
421 DTRACE_OBJS += dtrace.o dtrace_isa.o dtrace_asm.o

423 SDT_OBJS += sdt_subr.o

425 PROFILE_OBJS += profile.o

427 SYSTRACE_OBJS += systrace.o

429 LOCKSTAT_OBJS += lockstat.o

431 FASTTRAP_OBJS += fasttrap.o fasttrap_isa.o

433 DCPC_OBJS += dcpc.o

435 #
436 #       Driver (pseudo-driver) Modules
437 #
438 IPP_OBJS +=     ippctl.o

440 AUDIO_OBJS += audio_client.o audio_ddi.o audio_engine.o \
441             audio_fltdata.o audio_format.o audio_ctrl.o \
442             audio_grc3.o audio_output.o audio_input.o \
443             audio_oss.o audio_sun.o

445 AUDIOEMU10K_OBJS += audioemu10k.o

447 AUDIOENS_OBJS += audioens.o

449 AUDIOVIA823X_OBJS += audiovia823x.o
```



```
451 AUDIOVIA97_OBJS += audiovia97.o

453 AUDIO1575_OBJS += audio1575.o

455 AUDIO810_OBJS += audio810.o

457 AUDIOCMI_OBJS += audiocmi.o

459 AUDIOCMIHD_OBJS += audiocmihd.o

461 AUDIOHD_OBJS += audiohd.o

463 AUDIOIXP_OBJS += audioixp.o

465 AUDIOLS_OBJS += audiols.o

467 AUDIOP16X_OBJS += audiop16x.o

469 AUDIOPCI_OBJS += audiopci.o

471 AUDIOSOLO_OBJS += audiosolo.o

473 AUDIOTS_OBJS += audiots.o

475 AC97_OBJS += ac97.o ac97_ad.o ac97_alc.o ac97_cmi.o

477 BLKDEV_OBJS += blkdev.o

479 CARDBUS_OBJS += cardbus.o cardbus_hp.o cardbus_cfg.o

481 CONSKBD_OBJS += conskbd.o

483 CONSMS_OBJS +=  consms.o

485 OLDPTY_OBJS +=  tty_ptyconf.o

487 PTC_OBJS +=     tty_pty.o

489 PTSL_OBJS +=    tty_pts.o

491 PTM_OBJS +=     ptm.o

493 MII_OBJS +=     mii.o mii_cicada.o mii_natsemi.o mii_intel.o mii_qualsemi.o \
494                 mii_marvell.o mii_realtek.o mii_other.o

496 PTS_OBJS +=     pts.o

498 PTY_OBJS +=     ptms_conf.o

500 SAD_OBJS +=     sad.o

502 MD4_OBJS +=     md4.o md4_mod.o

504 MD5_OBJS +=     md5.o md5_mod.o

506 SHA1_OBJS +=    sha1.o sha1_mod.o

508 SHA2_OBJS +=    sha2.o sha2_mod.o

510 IPGPC_OBJS +=   classifierddi.o classifier.o filters.o trie.o table.o \
511                 ba_table.o

513 DSCPMK_OBJS +=  dscpmk.o dscpmkddi.o

515 DLCOSMK_OBJS += dlcosmk.o dlcosmkddi.o
```

```
517 FLOWACCT_OBJS +=        flowacctddi.o flowacct.o

519 TOKENMT_OBJS += tokenmt.o tokenmtddi.o

521 TSWTCL_OBJS +=  tswtcl.o tswtclddi.o

523 ARP_OBJS +=     arpddi.o

525 ICMP_OBJS +=    icmpddi.o

527 ICMP6_OBJS +=   icmp6ddi.o

529 RTS_OBJS +=     rtsddi.o

531 IP_ICMP_OBJS =  icmp.o icmp_opt_data.o
532 IP_RTS_OBJS =   rts.o rts_opt_data.o
533 IP_TCP_OBJS =   tcp.o tcp_fusion.o tcp_opt_data.o tcp_sack.o tcp_stats.o \
534                 tcp_misc.o tcp_timers.o tcp_time_wait.o tcp_tpi.o tcp_output.o \
535                 tcp_input.o tcp_socket.o tcp_bind.o tcp_cluster.o tcp_tunables.o
536 IP_UDP_OBJS =   udp.o udp_opt_data.o udp_tunables.o udp_stats.o
537 IP_SCTP_OBJS =  sctp.o sctp_opt_data.o sctp_output.o \
538                 sctp_init.o sctp_input.o sctp_cookie.o \
539                 sctp_conn.o sctp_error.o sctp_snmp.o \
540                 sctp_tunables.o sctp_shutdown.o sctp_common.o \
541                 sctp_timer.o sctp_heartbeat.o sctp_hash.o \
542                 sctp_bind.o sctp_notify.o sctp_asconf.o \
543                 sctp_addr.o tn_ipopt.o tnet.o ip_netinfo.o \
544                 sctp_misc.o
545 IP_ILB_OBJS =   ilb.o ilb_nat.o ilb_conn.o ilb_alg_hash.o ilb_alg_rr.o

547 IP_OBJS +=      igmp.o ipmp.o ip.o ip6.o ip6_asp.o ip6_if.o ip6_ire.o \
548                 ip6_rts.o ip_if.o ip_ire.o ip_listutils.o ip_mroute.o \
549                 ip_multi.o ip2mac.o ip_ndp.o ip_rts.o ip_srcid.o \
550                 ipddi.o ipdrop.o mi.o nd.o tunables.o optcom.o snmpcom.o \
551                 ipsec_loader.o spd.o ipclassifier.o inet_common.o ip_squeue.o \
552                 squeue.o ip_sadb.o ip_ftable.o proto_set.o radix.o ip_dummy.o \
553                 ip_helper_stream.o ip_tunables.o \
554                 ip_output.o ip_input.o ip6_input.o ip6_output.o ip_arp.o \
555                 conn_opt.o ip_attr.o ip_dce.o \
556                 $(IP_ICMP_OBJS) \
557                 $(IP_RTS_OBJS) \
558                 $(IP_TCP_OBJS) \
559                 $(IP_UDP_OBJS) \
560                 $(IP_SCTP_OBJS) \
561                 $(IP_ILB_OBJS)

563 IP6_OBJS +=     ip6ddi.o

565 HOOK_OBJS +=    hook.o

567 NETI_OBJS +=    neti_impl.o neti_mod.o neti_stack.o

569 KEYSOCK_OBJS += keysockddi.o keysock.o keysock_opt_data.o

571 IPNET_OBJS +=   ipnet.o ipnet_bpf.o

573 SPDSOCK_OBJS += spdsockddi.o spdsock.o spdsock_opt_data.o

575 IPSECESP_OBJS += ipsecespddi.o ipsecesp.o

577 IPSECAH_OBJS += ipsecahddi.o ipsecah.o sadb.o

579 SPPP_OBJS +=    sppp.o sppp_dlpi.o sppp_mod.o s_common.o

581 SPPPTUN_OBJS += sppptun.o sppptun_mod.o
```



```
583 SPPPASYN_OBJS += spppasyn.o spppasyn_mod.o

585 SPPPCOMP_OBJS += spppcomp.o spppcomp_mod.o deflate.o bsd-comp.o vjcompress.o \
586                 zlib.o

588 TCP_OBJS +=     tcpddi.o

590 TCP6_OBJS +=    tcp6ddi.o

592 NCA_OBJS +=     ncaddi.o

594 SDP_SOCK_MOD_OBJS += sockmod_sdp.o socksdp.o socksdpsubr.o

596 SCTP_SOCK_MOD_OBJS += sockmod_sctp.o socksctp.o socksctpsubr.o

598 PFP_SOCK_MOD_OBJS += sockmod_pfp.o

600 RDS_SOCK_MOD_OBJS += sockmod_rds.o

602 RDS_OBJS +=     rdsddi.o rdssubr.o rds_opt.o rds_ioctl.o

604 RDSIB_OBJS +=   rdsib.o rdsib_ib.o rdsib_cm.o rdsib_ep.o rdsib_buf.o \
605                 rdsib_debug.o rdsib_sc.o

607 RDSV3_OBJS +=   af_rds.o rdsv3_ddi.o bind.o loop.o threads.o connection.o \
608                 transport.o cong.o sysctl.o message.o rds_recv.o send.o \
609                 stats.o info.o page.o rdma_transport.o ib_ring.o ib_rdma.o \
610                 ib_recv.o ib.o ib_send.o ib_sysctl.o ib_stats.o ib_cm.o \
611                 rdsv3_sc.o rdsv3_debug.o rdsv3_impl.o rdma.o rdsv3_af_thr.o

613 ISER_OBJS +=    iser.o iser_cm.o iser_cq.o iser_ib.o iser_idm.o \
614                 iser_resource.o iser_xfer.o

616 UDP_OBJS +=     udpddi.o

618 UDP6_OBJS +=    udp6ddi.o

620 SY_OBJS +=      gentty.o

622 TCO_OBJS +=     ticots.o

624 TCOO_OBJS +=    ticotsord.o

626 TCL_OBJS +=     ticlts.o

628 TL_OBJS +=      tl.o

630 DUMP_OBJS +=    dump.o

632 BPF_OBJS +=     bpf.o bpf_filter.o bpf_mod.o bpf_dlt.o bpf_mac.o

634 CLONE_OBJS +=   clone.o

636 CN_OBJS +=      cons.o

638 DLD_OBJS +=     dld_drv.o dld_proto.o dld_str.o dld_flow.o

640 DLS_OBJS +=     dls.o dls_link.o dls_mod.o dls_stat.o dls_mgmt.o

642 GLD_OBJS +=     gld.o gldutil.o

644 MAC_OBJS +=     mac.o  mac_bcast.o mac_client.o mac_datapath_setup.o mac_flow.o
645                 mac_hio.o mac_mod.o mac_ndd.o mac_provider.o mac_sched.o \
646                 mac_protect.o mac_soft_ring.o mac_stat.o mac_util.o
```

```
 648 MAC_6TO4_OBJS +=        mac_6to4.o

 650 MAC_ETHER_OBJS +=       mac_ether.o

 652 MAC_IPV4_OBJS +=        mac_ipv4.o

 654 MAC_IPV6_OBJS +=        mac_ipv6.o

 656 MAC_WIFI_OBJS +=        mac_wifi.o

 658 MAC_IB_OBJS +=          mac_ib.o

 660 IPTUN_OBJS +=   iptun_dev.o iptun_ctl.o iptun.o

 662 AGGR_OBJS +=    aggr_dev.o aggr_ctl.o aggr_grp.o aggr_port.o \
 663                 aggr_send.o aggr_recv.o aggr_lacp.o

 665 SOFTMAC_OBJS += softmac_main.o softmac_ctl.o softmac_capab.o \
 666                 softmac_dev.o softmac_stat.o softmac_pkt.o softmac_fp.o

 668 NET80211_OBJS += net80211.o net80211_proto.o net80211_input.o \
 669                  net80211_output.o net80211_node.o net80211_crypto.o \
 670                  net80211_crypto_none.o net80211_crypto_wep.o net80211_ioctl.o \
 671                  net80211_crypto_tkip.o net80211_crypto_ccmp.o  \
 672                  net80211_ht.o

 674 VNIC_OBJS +=    vnic_ctl.o vnic_dev.o

 676 SIMNET_OBJS +=  simnet.o

 678 IB_OBJS +=      ibnex.o ibnex_ioctl.o ibnex_hca.o

 680 IBCM_OBJS +=    ibcm_impl.o ibcm_sm.o ibcm_ti.o ibcm_utils.o ibcm_path.o \
 681                 ibcm_arp.o ibcm_arp_link.o

 683 IBDM_OBJS +=    ibdm.o

 685 IBDMA_OBJS +=   ibdma.o

 687 IBMF_OBJS +=    ibmf.o ibmf_impl.o ibmf_dr.o ibmf_wqe.o ibmf_ud_dest.o ibmf_mod.
 688                 ibmf_send.o ibmf_recv.o ibmf_handlers.o ibmf_trans.o \
 689                 ibmf_timers.o ibmf_msg.o ibmf_utils.o ibmf_rmpp.o \
 690                 ibmf_saa.o ibmf_saa_impl.o ibmf_saa_utils.o ibmf_saa_events.o

 692 IBTL_OBJS +=    ibtl_impl.o ibtl_util.o ibtl_mem.o ibtl_handlers.o ibtl_qp.o \
 693                 ibtl_cq.o ibtl_wr.o ibtl_hca.o ibtl_chan.o ibtl_cm.o \
 694                 ibtl_mcg.o ibtl_ibnex.o ibtl_srq.o ibtl_part.o

 696 TAVOR_OBJS +=   tavor.o tavor_agents.o tavor_cfg.o tavor_ci.o tavor_cmd.o \
 697                 tavor_cq.o tavor_event.o tavor_ioctl.o tavor_misc.o \
 698                 tavor_mr.o tavor_qp.o tavor_qpmod.o tavor_rsrc.o \
 699                 tavor_srq.o tavor_stats.o tavor_umap.o tavor_wr.o

 701 HERMON_OBJS +=  hermon.o hermon_agents.o hermon_cfg.o hermon_ci.o hermon_cmd.o \
 702                 hermon_cq.o hermon_event.o hermon_ioctl.o hermon_misc.o \
 703                 hermon_mr.o hermon_qp.o hermon_qpmod.o hermon_rsrc.o \
 704                 hermon_srq.o hermon_stats.o hermon_umap.o hermon_wr.o \
 705                 hermon_fcoib.o hermon_fm.o

 707 DAPLT_OBJS +=   daplt.o

 709 SOL_OFS_OBJS += sol_cma.o sol_ib_cma.o sol_uobj.o \
 710                 sol_ofs_debug_util.o sol_ofs_gen_util.o \
 711                 sol_kverbs.o

 713 SOL_UCMA_OBJS +=        sol_ucma.o
```



```
 715 SOL_UVERBS_OBJS +=      sol_uverbs.o sol_uverbs_comp.o sol_uverbs_event.o \
 716                         sol_uverbs_hca.o sol_uverbs_qp.o

 718 SOL_UMAD_OBJS += sol_umad.o

 720 KSTAT_OBJS +=   kstat.o

 722 KSYMS_OBJS +=   ksyms.o

 724 INSTANCE_OBJS += inst_sync.o

 726 IWSCN_OBJS +=   iwscons.o

 728 LOFI_OBJS +=    lofi.o LzmaDec.o

 730 FSSNAP_OBJS +=  fssnap.o

 732 FSSNAPIF_OBJS += fssnap_if.o

 734 MM_OBJS +=      mem.o

 736 PHYSMEM_OBJS += physmem.o

 738 OPTIONS_OBJS += options.o

 740 WINLOCK_OBJS += winlockio.o

 742 PM_OBJS +=      pm.o
 743 SRN_OBJS +=     srn.o

 745 PSEUDO_OBJS +=  pseudonex.o

 747 RAMDISK_OBJS += ramdisk.o

 749 LLC1_OBJS += llc1.o

 751 USBKBM_OBJS += usbkbm.o

 753 USBWCM_OBJS += usbwcm.o

 755 BOFI_OBJS += bofi.o

 757 HID_OBJS += hid.o

 759 USBSKEL_OBJS += usbskel.o

 761 USBVC_OBJS += usbvc.o usbvc_v4l2.o

 763 HIDPARSER_OBJS += hidparser.o

 765 USB_AC_OBJS += usb_ac.o

 767 USB_AS_OBJS += usb_as.o

 769 USB_AH_OBJS += usb_ah.o

 771 USBMS_OBJS += usbms.o

 773 USBPRN_OBJS += usbprn.o

 775 UGEN_OBJS += ugen.o

 777 USBSER_OBJS += usbser.o usbser_rseq.o

 779 USBSACM_OBJS += usbsacm.o
```

```
781 USBSER_KEYSPAN_OBJS += usbser_keyspan.o keyspan_dsd.o keyspan_pipe.o

783 USBS49_FW_OBJS += keyspan_49fw.o

785 USBSPRL_OBJS += usbser_pl2303.o pl2303_dsd.o

787 USBFTDI_OBJS += usbser_uftdi.o uftdi_dsd.o

789 USBECM_OBJS += usbecm.o

791 WC_OBJS += wscons.o vcons.o

793 VCONS_CONF_OBJS += vcons_conf.o

795 SCSI_OBJS +=    scsi_capabilities.o scsi_confsubr.o scsi_control.o \
796                 scsi_data.o scsi_fm.o scsi_hba.o scsi_reset_notify.o \
797                 scsi_resource.o scsi_subr.o scsi_transport.o scsi_watch.o \
798                 smp_transport.o

800 SCSI_VHCI_OBJS +=              scsi_vhci.o mpapi_impl.o scsi_vhci_tpgs.o

802 SCSI_VHCI_F_SYM_OBJS +=        sym.o

804 SCSI_VHCI_F_TPGS_OBJS +=       tpgs.o

806 SCSI_VHCI_F_ASYM_SUN_OBJS +=   asym_sun.o

808 SCSI_VHCI_F_SYM_HDS_OBJS +=    sym_hds.o

810 SCSI_VHCI_F_TAPE_OBJS +=       tape.o

812 SCSI_VHCI_F_TPGS_TAPE_OBJS +=  tpgs_tape.o

814 SGEN_OBJS +=    sgen.o

816 SMP_OBJS +=     smp.o

818 SATA_OBJS +=    sata.o

820 USBA_OBJS +=    hcdi.o  usba.o  usbai.o hubdi.o parser.o genconsole.o \
821                 usbai_pipe_mgmt.o usbai_req.o usbai_util.o usbai_register.o \
822                 usba_devdb.o usba10_calls.o usba_ugen.o

824 USBA10_OBJS +=  usba10.o

826 RSM_OBJS +=     rsm.o   rsmka_pathmanager.o     rsmka_util.o

828 RSMOPS_OBJS +=  rsmops.o

830 S1394_OBJS +=   t1394.o t1394_errmsg.o s1394.o s1394_addr.o s1394_asynch.o \
831                 s1394_bus_reset.o s1394_cmp.o s1394_csr.o s1394_dev_disc.o \
832                 s1394_fa.o s1394_fcp.o \
833                 s1394_hotplug.o s1394_isoch.o s1394_misc.o h1394.o nx1394.o

835 HCI1394_OBJS += hci1394.o hci1394_async.o hci1394_attach.o hci1394_buf.o \
836                 hci1394_csr.o hci1394_detach.o hci1394_extern.o \
837                 hci1394_ioctl.o hci1394_isoch.o hci1394_isr.o \
838                 hci1394_ixl_comp.o hci1394_ixl_isr.o hci1394_ixl_misc.o \
839                 hci1394_ixl_update.o hci1394_misc.o hci1394_ohci.o \
840                 hci1394_q.o hci1394_s1394if.o hci1394_tlabel.o \
841                 hci1394_tlist.o hci1394_vendor.o

843 AV1394_OBJS +=  av1394.o av1394_as.o av1394_async.o av1394_cfgrom.o \
844                 av1394_cmp.o av1394_fcp.o av1394_isoch.o av1394_isoch_chan.o \
845                 av1394_isoch_recv.o av1394_isoch_xmit.o av1394_list.o \
```



```
846                 av1394_queue.o

848 DCAM1394_OBJS += dcam.o dcam_frame.o dcam_param.o dcam_reg.o \
849                 dcam_ring_buff.o

851 SCSA1394_OBJS += hba.o sbp2_driver.o sbp2_bus.o

853 SBP2_OBJS += cfgrom.o sbp2.o

855 PMODEM_OBJS += pmodem.o pmodem_cis.o cis.o cis_callout.o cis_handlers.o cis_para

857 DSW_OBJS +=     dsw.o dsw_dev.o ii_tree.o

859 NCALL_OBJS +=   ncall.o \
860                 ncall_stub.o

862 RDC_OBJS +=     rdc.o \
863                 rdc_dev.o \
864                 rdc_io.o \
865                 rdc_clnt.o \
866                 rdc_prot_xdr.o \
867                 rdc_svc.o \
868                 rdc_bitmap.o \
869                 rdc_health.o \
870                 rdc_subr.o \
871                 rdc_diskq.o

873 RDCSRV_OBJS +=  rdcsrv.o

875 RDCSTUB_OBJS += rdc_stub.o

877 SDBC_OBJS +=    sd_bcache.o \
878                 sd_bio.o \
879                 sd_conf.o \
880                 sd_ft.o \
881                 sd_hash.o \
882                 sd_io.o \
883                 sd_misc.o \
884                 sd_pcu.o \
885                 sd_tdaemon.o \
886                 sd_trace.o \
887                 sd_iob_impl0.o \
888                 sd_iob_impl1.o \
889                 sd_iob_impl2.o \
890                 sd_iob_impl3.o \
891                 sd_iob_impl4.o \
892                 sd_iob_impl5.o \
893                 sd_iob_impl6.o \
894                 sd_iob_impl7.o \
895                 safestore.o \
896                 safestore_ram.o

898 NSCTL_OBJS +=   nsctl.o \
899                 nsc_cache.o \
900                 nsc_disk.o \
901                 nsc_dev.o \
902                 nsc_freeze.o \
903                 nsc_gen.o \
904                 nsc_mem.o \
905                 nsc_ncallio.o \
906                 nsc_power.o \
907                 nsc_resv.o \
908                 nsc_rmspin.o \
909                 nsc_solaris.o \
910                 nsc_trap.o \
911                 nsc_list.o
```

```
912 UNISTAT_OBJS += spuni.o \
913                 spcs_s_k.o

915 NSKERN_OBJS +=  nsc_ddi.o \
916                 nsc_proc.o \
917                 nsc_raw.o \
918                 nsc_thread.o \
919                 nskernd.o

921 SV_OBJS +=      sv.o

923 PMCS_OBJS += pmcs_attach.o pmcs_ds.o pmcs_intr.o pmcs_nvram.o pmcs_sata.o \
924                 pmcs_scsa.o pmcs_smhba.o pmcs_subr.o pmcs_fwlog.o

926 PMCS8001FW_C_OBJS +=    pmcs_fw_hdr.o
927 PMCS8001FW_OBJS +=              $(PMCS8001FW_C_OBJS) SPCBoot.o ila.o firmware.o

929 #
930 #       Build up defines and paths.

932 ST_OBJS +=      st.o    st_conf.o

934 EMLXS_OBJS +=   emlxs_clock.o emlxs_dfc.o emlxs_dhchap.o emlxs_diag.o \
935                 emlxs_download.o emlxs_dump.o emlxs_els.o emlxs_event.o \
936                 emlxs_fcf.o emlxs_fcp.o emlxs_fct.o emlxs_hba.o emlxs_ip.o \
937                 emlxs_mbox.o emlxs_mem.o emlxs_msg.o emlxs_node.o \
938                 emlxs_pkt.o emlxs_sli3.o emlxs_sli4.o emlxs_solaris.o \
939                 emlxs_thread.o

941 EMLXS_FW_OBJS +=        emlxs_fw.o

943 OCE_OBJS +=     oce_buf.o oce_fm.o oce_gld.o oce_hw.o oce_intr.o oce_main.o \
944                 oce_mbx.o oce_mq.o oce_queue.o oce_rx.o oce_stat.o oce_tx.o \
945                 oce_utils.o

947 FCT_OBJS += discovery.o fct.o

949 QLT_OBJS += 2400.o 2500.o 8100.o qlt.o qlt_dma.o

951 SRPT_OBJS += srpt_mod.o srpt_ch.o srpt_cm.o srpt_ioc.o srpt_stp.o

953 FCOE_OBJS += fcoe.o fcoe_eth.o fcoe_fc.o

955 FCOET_OBJS += fcoet.o fcoet_eth.o fcoet_fc.o

957 FCOEI_OBJS += fcoei.o fcoei_eth.o fcoei_lv.o

959 ISCSIT_SHARED_OBJS += \
960                 iscsit_common.o

962 ISCSIT_OBJS +=  $(ISCSIT_SHARED_OBJS) \
963                 iscsit.o iscsit_tgt.o iscsit_sess.o iscsit_login.o \
964                 iscsit_text.o iscsit_isns.o iscsit_radiusauth.o \
965                 iscsit_radiuspacket.o iscsit_auth.o iscsit_authclient.o

967 PPPT_OBJS +=    alua_ic_if.o pppt.o pppt_msg.o pppt_tgt.o

969 STMF_OBJS += lun_map.o stmf.o

971 STMF_SBD_OBJS += sbd.o sbd_scsi.o sbd_pgr.o sbd_zvol.o

973 SYSMSG_OBJS +=  sysmsg.o

975 SES_OBJS +=     ses.o ses_sen.o ses_safte.o ses_ses.o

977 TNF_OBJS +=     tnf_buf.o       tnf_trace.o     tnf_writer.o    trace_init.o \
```



```
978                 trace_funcs.o   tnf_probe.o     tnf.o

980 LOGINDMUX_OBJS += logindmux.o

982 DEVINFO_OBJS += devinfo.o

984 DEVPOLL_OBJS += devpoll.o

986 DEVPOOL_OBJS += devpool.o

988 I8042_OBJS +=   i8042.o

990 KB8042_OBJS +=  \
991                 at_keyprocess.o \
992                 kb8042.o        \
993                 kb8042_keytables.o

995 MOUSE8042_OBJS += mouse8042.o

997 FDC_OBJS +=     fdc.o

999 ASY_OBJS +=     asy.o

1001 ECPP_OBJS +=    ecpp.o

1003 VUIDM3P_OBJS += vuidmice.o vuidm3p.o

1005 VUIDM4P_OBJS += vuidmice.o vuidm4p.o

1007 VUIDM5P_OBJS += vuidmice.o vuidm5p.o

1009 VUIDPS2_OBJS += vuidmice.o vuidps2.o

1011 HPCSVC_OBJS += hpcsvc.o

1013 PCIE_MISC_OBJS += pcie.o pcie_fault.o pcie_hp.o pciehpc.o pcishpc.o pcie_pwr.o p

1015 PCIHPNEXUS_OBJS += pcihp.o

1017 OPENEEPR_OBJS += openprom.o

1019 RANDOM_OBJS += random.o

1021 PSHOT_OBJS += pshot.o

1023 GEN_DRV_OBJS += gen_drv.o

1025 TCLIENT_OBJS += tclient.o

1027 TPHCI_OBJS += tphci.o

1029 TVHCI_OBJS += tvhci.o

1031 EMUL64_OBJS += emul64.o emul64_bsd.o

1033 FCP_OBJS += fcp.o

1035 FCIP_OBJS += fcip.o

1037 FCSM_OBJS += fcsm.o

1039 FCTL_OBJS += fctl.o

1041 FP_OBJS += fp.o

1043 QLC_OBJS += ql_api.o ql_debug.o ql_hba_fru.o ql_init.o ql_iocb.o ql_ioctl.o \
```

```
1044         ql_isr.o ql_mbx.o ql_nx.o ql_xioctl.o ql_fw_table.o

1046 QLC_FW_2200_OBJS += ql_fw_2200.o

1048 QLC_FW_2300_OBJS += ql_fw_2300.o

1050 QLC_FW_2400_OBJS += ql_fw_2400.o

1052 QLC_FW_2500_OBJS += ql_fw_2500.o

1054 QLC_FW_6322_OBJS += ql_fw_6322.o

1056 QLC_FW_8100_OBJS += ql_fw_8100.o

1058 QLGE_OBJS += qlge.o qlge_dbg.o qlge_flash.o qlge_fm.o qlge_gld.o qlge_mpi.o

1060 ZCONS_OBJS += zcons.o

1062 NV_SATA_OBJS += nv_sata.o

1064 SI3124_OBJS += si3124.o

1066 AHCI_OBJS += ahci.o

1068 PCIIDE_OBJS += pci-ide.o

1070 PCEPP_OBJS += pcepp.o

1072 CPC_OBJS += cpc.o

1074 CPUID_OBJS += cpuid_drv.o

1076 SYSEVENT_OBJS += sysevent.o

1078 BL_OBJS += bl.o

1080 DRM_OBJS += drm_sunmod.o drm_kstat.o drm_agpsupport.o \
1081             drm_auth.o drm_bufs.o drm_context.o drm_dma.o \
1082             drm_drawable.o drm_drv.o drm_fops.o drm_ioctl.o drm_irq.o \
1083             drm_lock.o drm_memory.o drm_msg.o drm_pci.o drm_scatter.o \
1084             drm_cache.o drm_gem.o drm_mm.o ati_pcigart.o

1086 FM_OBJS += devfm.o devfm_machdep.o

1088 RTLS_OBJS +=    rtls.o

1090 #
1091 #                       exec modules
1092 #
1093 AOUTEXEC_OBJS +=aout.o

1095 ELFEXEC_OBJS += elf.o elf_notes.o old_notes.o

1097 INTPEXEC_OBJS +=intp.o

1099 SHBINEXEC_OBJS +=shbin.o

1101 JAVAEXEC_OBJS +=java.o

1103 #
1104 #                       file system modules
1105 #
1106 AUTOFS_OBJS +=  auto_vfsops.o auto_vnops.o auto_subr.o auto_xdr.o auto_sys.o

1108 CACHEFS_OBJS += cachefs_cnode.o         cachefs_cod.o \
1109                 cachefs_dir.o           cachefs_dlog.o  cachefs_filegrp.o \
```



```
1110                 cachefs_fscache.o       cachefs_ioctl.o cachefs_log.o \
1111                 cachefs_module.o \
1112                 cachefs_noopc.o         cachefs_resource.o \
1113                 cachefs_strict.o \
1114                 cachefs_subr.o          cachefs_vfsops.o \
1115                 cachefs_vnops.o

1117 DCFS_OBJS +=    dc_vnops.o

1119 DEVFS_OBJS +=   devfs_subr.o    devfs_vfsops.o  devfs_vnops.o

1121 DEV_OBJS  +=    sdev_subr.o     sdev_vfsops.o   sdev_vnops.o    \
1122                 sdev_ptsops.o   sdev_zvolops.o  sdev_comm.o     \
1123                 sdev_profile.o  sdev_ncache.o   sdev_netops.o   \
1124                 sdev_ipnetops.o \
1125                 sdev_vtops.o

1127 CTFS_OBJS +=    ctfs_all.o ctfs_cdir.o ctfs_ctl.o ctfs_event.o \
1128                 ctfs_latest.o ctfs_root.o ctfs_sym.o ctfs_tdir.o ctfs_tmpl.o

1130 OBJFS_OBJS +=   objfs_vfs.o     objfs_root.o    objfs_common.o \
1131                 objfs_odir.o    objfs_data.o

1133 FDFS_OBJS +=    fdops.o

1135 FIFO_OBJS +=    fifosubr.o      fifovnops.o

1137 PIPE_OBJS +=    pipe.o

1139 HSFS_OBJS +=    hsfs_node.o     hsfs_subr.o     hsfs_vfsops.o   hsfs_vnops.o \
1140                 hsfs_susp.o     hsfs_rrip.o     hsfs_susp_subr.o

1142 LOFS_OBJS +=    lofs_subr.o     lofs_vfsops.o   lofs_vnops.o

1144 NAMEFS_OBJS +=  namevfs.o       namevno.o

1146 NFS_OBJS +=     nfs_client.o    nfs_common.o    nfs_dump.o \
1147                 nfs_subr.o      nfs_vfsops.o    nfs_vnops.o \
1148                 nfs_xdr.o       nfs_sys.o       nfs_strerror.o \
1149                 nfs3_vfsops.o   nfs3_vnops.o    nfs3_xdr.o \
1150                 nfs_acl_vnops.o nfs_acl_xdr.o   nfs4_vfsops.o \
1151                 nfs4_vnops.o    nfs4_xdr.o      nfs4_idmap.o \
1152                 nfs4_shadow.o   nfs4_subr.o \
1153                 nfs4_attr.o     nfs4_rnode.o    nfs4_client.o \
1154                 nfs4_acache.o   nfs4_common.o   nfs4_client_state.o \
1155                 nfs4_callback.o nfs4_recovery.o nfs4_client_secinfo.o \
1156                 nfs4_client_debug.o     nfs_stats.o \
1157                 nfs4_acl.o      nfs4_stub_vnops.o       nfs_cmd.o

1159 NFSSRV_OBJS +=  nfs_server.o    nfs_srv.o       nfs3_srv.o \
1160                 nfs_acl_srv.o   nfs_auth.o      nfs_auth_xdr.o \
1161                 nfs_export.o    nfs_log.o       nfs_log_xdr.o \
1162                 nfs4_srv.o      nfs4_state.o    nfs4_srv_attr.o \
1163                 nfs4_srv_ns.o   nfs4_db.o       nfs4_srv_deleg.o \
1164                 nfs4_deleg_ops.o nfs4_srv_readdir.o nfs4_dispatch.o

1166 SMBSRV_SHARED_OBJS += \
1167                 smb_inet.o \
1168                 smb_match.o \
1169                 smb_msgbuf.o \
1170                 smb_oem.o \
1171                 smb_string.o \
1172                 smb_utf8.o \
1173                 smb_door_legacy.o \
1174                 smb_xdr.o \
1175                 smb_token.o \
```

```
1176                    smb_token_xdr.o \
1177                    smb_sid.o \
1178                    smb_native.o \
1179                    smb_netbios_util.o

1181 SMBSRV_OBJS +=    $(SMBSRV_SHARED_OBJS)                   \
1182                    smb_acl.o                               \
1183                    smb_alloc.o                             \
1184                    smb_close.o                             \
1185                    smb_common_open.o                       \
1186                    smb_common_transact.o                   \
1187                    smb_create.o                            \
1188                    smb_delete.o                            \
1189                    smb_directory.o                         \
1190                    smb_dispatch.o                          \
1191                    smb_echo.o                              \
1192                    smb_fem.o                               \
1193                    smb_find.o                              \
1194                    smb_flush.o                             \
1195                    smb_fsinfo.o                            \
1196                    smb_fsops.o                             \
1197                    smb_init.o                              \
1198                    smb_kdoor.o                             \
1199                    smb_kshare.o                            \
1200                    smb_kutil.o                             \
1201                    smb_lock.o                              \
1202                    smb_lock_byte_range.o                   \
1203                    smb_locking_andx.o                      \
1204                    smb_logoff_andx.o                       \
1205                    smb_mangle_name.o                       \
1206                    smb_mbuf_marshaling.o                   \
1207                    smb_mbuf_util.o                         \
1208                    smb_negotiate.o                         \
1209                    smb_net.o                               \
1210                    smb_node.o                              \
1211                    smb_nt_cancel.o                         \
1212                    smb_nt_create_andx.o                    \
1213                    smb_nt_transact_create.o                \
1214                    smb_nt_transact_ioctl.o                 \
1215                    smb_nt_transact_notify_change.o         \
1216                    smb_nt_transact_quota.o                 \
1217                    smb_nt_transact_security.o              \
1218                    smb_odir.o                              \
1219                    smb_ofile.o                             \
1220                    smb_open_andx.o                         \
1221                    smb_opipe.o                             \
1222                    smb_oplock.o                            \
1223                    smb_pathname.o                          \
1224                    smb_print.o                             \
1225                    smb_process_exit.o                      \
1226                    smb_query_fileinfo.o                    \
1227                    smb_read.o                              \
1228                    smb_rename.o                            \
1229                    smb_sd.o                                \
1230                    smb_seek.o                              \
1231                    smb_server.o                            \
1232                    smb_session.o                           \
1233                    smb_session_setup_andx.o                \
1234                    smb_set_fileinfo.o                      \
1235                    smb_signing.o                           \
1236                    smb_tree.o                              \
1237                    smb_trans2_create_directory.o           \
1238                    smb_trans2_dfs.o                        \
1239                    smb_trans2_find.o                       \
1240                    smb_tree_connect.o                      \
1241                    smb_unlock_byte_range.o                 \
```



```
1242                    smb_user.o                              \
1243                    smb_vfs.o                               \
1244                    smb_vops.o                              \
1245                    smb_vss.o                               \
1246                    smb_write.o                             \
1247                    smb_write_raw.o

1249 PCFS_OBJS +=      pc_alloc.o      pc_dir.o        pc_node.o       pc_subr.o \
1250                    pc_vfsops.o     pc_vnops.o

1252 PROC_OBJS +=      prcontrol.o     prioctl.o       prsubr.o        prusrio.o \
1253                    prvfsops.o      prvnops.o

1255 MNTFS_OBJS +=     mntvfsops.o     mntvnops.o

1257 SHAREFS_OBJS +=   sharetab.o      sharefs_vfsops.o        sharefs_vnops.o

1259 SPEC_OBJS +=      specsubr.o      specvfsops.o    specvnops.o

1261 SOCK_OBJS +=      socksubr.o      sockvfsops.o    sockparams.o    \
1262                    socksyscalls.o  socktpi.o       sockstr.o \
1263                    sockcommon_vnops.o      sockcommon_subr.o \
1264                    sockcommon_sops.o       sockcommon.o    \
1265                    sock_notsupp.o  socknotify.o \
1266                    nl7c.o          nl7curi.o       nl7chttp.o      nl7clogd.o \
1267                    nl7cnca.o       sodirect.o      sockfilter.o

1269 TMPFS_OBJS +=     tmp_dir.o       tmp_subr.o      tmp_tnode.o     tmp_vfsops.o \
1270                    tmp_vnops.o

1272 UDFS_OBJS +=      udf_alloc.o     udf_bmap.o      udf_dir.o       \
1273                    udf_inode.o     udf_subr.o      udf_vfsops.o    \
1274                    udf_vnops.o

1276 UFS_OBJS +=       ufs_alloc.o     ufs_bmap.o      ufs_dir.o       ufs_xattr.o \
1277                    ufs_inode.o     ufs_subr.o      ufs_tables.o    ufs_vfsops.o \
1278                    ufs_vnops.o     quota.o         quotacalls.o    quota_ufs.o \
1279                    ufs_filio.o     ufs_lockfs.o    ufs_thread.o    ufs_trans.o \
1280                    ufs_acl.o       ufs_panic.o     ufs_directio.o  ufs_log.o \
1281                    ufs_extvnops.o  ufs_snap.o      lufs.o          lufs_thread.o \
1282                    lufs_log.o      lufs_map.o      lufs_top.o      lufs_debug.o
1283 VSCAN_OBJS +=     vscan_drv.o     vscan_svc.o vscan_door.o

1285 NSMB_OBJS +=      smb_conn.o      smb_dev.o       smb_iod.o       smb_pass.o \
1286                    smb_rq.o        smb_sign.o      smb_smb.o       smb_subrs.o \
1287                    smb_time.o      smb_tran.o      smb_trantcp.o   smb_usr.o \
1288                    subr_mchain.o

1290 SMBFS_COMMON_OBJS += smbfs_ntacl.o
1291 SMBFS_OBJS +=     smbfs_vfsops.o  smbfs_vnops.o   smbfs_node.o    \
1292                    smbfs_acl.o     smbfs_client.o  smbfs_smb.o     \
1293                    smbfs_subr.o    smbfs_subr2.o   \
1294                    smbfs_rwlock.o  smbfs_xattr.o   \
1295                    $(SMBFS_COMMON_OBJS)


1298 #
1299 #                       LVM modules
1300 #
1301 MD_OBJS += md.o md_error.o md_ioctl.o md_mddb.o md_names.o \
1302         md_med.o md_rename.o md_subr.o

1304 MD_COMMON_OBJS = md_convert.o md_crc.o md_revchk.o

1306 MD_DERIVED_OBJS = metamed_xdr.o meta_basic_xdr.o
```

```
1308 SOFTPART_OBJS += sp.o sp_ioctl.o

1310 STRIPE_OBJS += stripe.o stripe_ioctl.o

1312 HOTSPARES_OBJS += hotspares.o

1314 RAID_OBJS += raid.o raid_ioctl.o raid_replay.o raid_resync.o raid_hotspare.o

1316 MIRROR_OBJS += mirror.o mirror_ioctl.o mirror_resync.o

1318 NOTIFY_OBJS += md_notify.o

1320 TRANS_OBJS += mdtrans.o trans_ioctl.o trans_log.o

1322 ZFS_COMMON_OBJS +=             \
1323         arc.o                  \
1324         blkptr.o               \
1325         bplist.o               \
1326         bpobj.o                \
1327         bptree.o               \
1328         dbuf.o                 \
1329         ddt.o                  \
1330         ddt_zap.o              \
1331         dmu.o                  \
1332         dmu_diff.o             \
1333         dmu_send.o             \
1334         dmu_object.o           \
1335         dmu_objset.o           \
1336         dmu_traverse.o         \
1337         dmu_tx.o               \
1338         dnode.o                \
1339         dnode_sync.o           \
1340         dsl_bookmark.o         \
1341         dsl_dir.o              \
1342         dsl_dataset.o          \
1343         dsl_deadlist.o         \
1344         dsl_destroy.o          \
1345         dsl_pool.o             \
1346         dsl_synctask.o         \
1347         dsl_userhold.o         \
1348         dmu_zfetch.o           \
1349         dsl_deleg.o            \
1350         dsl_prop.o             \
1351         dsl_scan.o             \
1352         zfeature.o             \
1353         gzip.o                 \
1354         lz4.o                  \
1355         lzjb.o                 \
1356         metaslab.o             \
1357         multilist.o            \
1358         range_tree.o           \
1359         refcount.o             \
1360         rrwlock.o              \
1361         sa.o                   \
1362         sha256.o               \
1363         spa.o                  \
1364         spa_config.o           \
1365         spa_errlog.o           \
1366         spa_history.o          \
1367         spa_misc.o             \
1368         space_map.o            \
1369         space_reftree.o        \
1370         txg.o                  \
1371         uberblock.o            \
1372         unique.o               \
1373         vdev.o                 \
```



```
1374         vdev_cache.o           \
1375         vdev_file.o            \
1376         vdev_label.o           \
1377         vdev_mirror.o          \
1378         vdev_missing.o         \
1379         vdev_queue.o           \
1380         vdev_raidz.o           \
1381         vdev_root.o            \
1382         zap.o                  \
1383         zap_leaf.o             \
1384         zap_micro.o            \
1385         zfs_byteswap.o         \
1386         zfs_debug.o            \
1387         zfs_fm.o               \
1388         zfs_fuid.o             \
1389         zfs_sa.o               \
1390         zfs_znode.o            \
1391         zil.o                  \
1392         zio.o                  \
1393         zio_checksum.o         \
1394         zio_compress.o         \
1395         zio_inject.o           \
1396         zle.o                  \
1397         zrlock.o

1399 ZFS_SHARED_OBJS +=             \
1400         zfeature_common.o      \
1401         zfs_comutil.o          \
1402         zfs_deleg.o            \
1403         zfs_fletcher.o         \
1404         zfs_namecheck.o        \
1405         zfs_prop.o             \
1406         zpool_prop.o           \
1407         zprop_common.o

1409 ZFS_OBJS +=                    \
1410         $(ZFS_COMMON_OBJS)     \
1411         $(ZFS_SHARED_OBJS)     \
1412         vdev_disk.o            \
1413         zfs_acl.o              \
1414         zfs_ctldir.o           \
1415         zfs_dir.o              \
1416         zfs_ioctl.o            \
1417         zfs_log.o              \
1418         zfs_onexit.o           \
1419         zfs_replay.o           \
1420         zfs_rlock.o            \
1421         zfs_vfsops.o           \
1422         zfs_vnops.o            \
1423         zvol.o

1425 ZUT_OBJS +=                    \
1426         zut.o

1428 #
1429 #                       streams modules
1430 #
1431 BUFMOD_OBJS     +=      bufmod.o

1433 CONNLD_OBJS +=  connld.o

1435 DEDUMP_OBJS +=  dedump.o

1437 DRCOMPAT_OBJS +=        drcompat.o

1439 LDLINUX_OBJS += ldlinux.o
```

```
1441 LDTERM_OBJS +=  ldterm.o uwidth.o

1443 PCKT_OBJS +=    pckt.o

1445 PFMOD_OBJS +=   pfmod.o

1447 PTEM_OBJS +=    ptem.o

1449 REDIRMOD_OBJS += strredirm.o

1451 TIMOD_OBJS +=   timod.o

1453 TIRDWR_OBJS +=  tirdwr.o

1455 TTCOMPAT_OBJS +=ttcompat.o

1457 LOG_OBJS +=     log.o

1459 PIPEMOD_OBJS += pipemod.o

1461 RPCMOD_OBJS +=  rpcmod.o        clnt_cots.o     clnt_clts.o \
1462                 clnt_gen.o      clnt_perr.o     mt_rpcinit.o    rpc_calmsg.o \
1463                 rpc_prot.o      rpc_sztypes.o   rpc_subr.o      rpcb_prot.o \
1464                 svc.o           svc_clts.o      svc_gen.o       svc_cots.o \
1465                 rpcsys.o        xdr_sizeof.o    clnt_rdma.o     svc_rdma.o \
1466                 xdr_rdma.o      rdma_subr.o     xdrrdma_sizeof.o

1468 KLMMOD_OBJS +=  klmmod.o \
1469                 nlm_impl.o \
1470                 nlm_rpc_handle.o \
1471                 nlm_dispatch.o \
1472                 nlm_rpc_svc.o \
1473                 nlm_client.o \
1474                 nlm_service.o \
1475                 nlm_prot_clnt.o \
1476                 nlm_prot_xdr.o \
1477                 nlm_rpc_clnt.o \
1478                 nsm_addr_clnt.o \
1479                 nsm_addr_xdr.o \
1480                 sm_inter_clnt.o \
1481                 sm_inter_xdr.o

1483 KLMOPS_OBJS +=  klmops.o

1485 TLIMOD_OBJS +=  tlimod.o        t_kalloc.o      t_kbind.o       t_kclose.o \
1486                 t_kconnect.o    t_kfree.o       t_kgtstate.o    t_kopen.o \
1487                 t_krcvudat.o    t_ksndudat.o    t_kspoll.o      t_kunbind.o \
1488                 t_kutil.o

1490 RLMOD_OBJS += rlmod.o

1492 TELMOD_OBJS += telmod.o

1494 CRYPTMOD_OBJS += cryptmod.o

1496 KB_OBJS +=      kbd.o           keytables.o

1498 #
1499 #                       ID mapping module
1500 #
1501 IDMAP_OBJS +=   idmap_mod.o     idmap_kapi.o    idmap_xdr.o     idmap_cache.o

1503 #
1504 #                       scheduling class modules
1505 #
```



```
1506 SDC_OBJS +=             sysdc.o

1508 RT_OBJS +=              rt.o
1509 RT_DPTBL_OBJS +=        rt_dptbl.o

1511 TS_OBJS +=              ts.o
1512 TS_DPTBL_OBJS +=        ts_dptbl.o

1514 IA_OBJS +=              ia.o

1516 FSS_OBJS +=             fss.o

1518 FX_OBJS +=              fx.o
1519 FX_DPTBL_OBJS +=        fx_dptbl.o

1521 #
1522 #                       Inter-Process Communication (IPC) modules
1523 #
1524 IPC_OBJS +=     ipc.o

1526 IPCMSG_OBJS +=  msg.o

1528 IPCSEM_OBJS +=  sem.o

1530 IPCSHM_OBJS +=  shm.o

1532 #
1533 #                       bignum module
1534 #
1535 COMMON_BIGNUM_OBJS += bignum_mod.o bignumimpl.o

1537 BIGNUM_OBJS += $(COMMON_BIGNUM_OBJS) $(BIGNUM_PSR_OBJS)

1539 #
1540 #                       kernel cryptographic framework
1541 #
1542 KCF_OBJS +=     kcf.o kcf_callprov.o kcf_cbufcall.o kcf_cipher.o kcf_crypto.o \
1543                 kcf_cryptoadm.o kcf_ctxops.o kcf_digest.o kcf_dual.o \
1544                 kcf_keys.o kcf_mac.o kcf_mech_tabs.o kcf_miscapi.o \
1545                 kcf_object.o kcf_policy.o kcf_prov_lib.o kcf_prov_tabs.o \
1546                 kcf_sched.o kcf_session.o kcf_sign.o kcf_spi.o kcf_verify.o \
1547                 kcf_random.o modes.o ecb.o cbc.o ctr.o ccm.o gcm.o \
1548                 fips_random.o

1550 CRYPTOADM_OBJS += cryptoadm.o

1552 CRYPTO_OBJS +=  crypto.o

1554 DPROV_OBJS +=   dprov.o

1556 DCA_OBJS +=     dca.o dca_3des.o dca_debug.o dca_dsa.o dca_kstat.o dca_rng.o \
1557                 dca_rsa.o

1559 AESPROV_OBJS += aes.o aes_impl.o aes_modes.o

1561 ARCFOURPROV_OBJS += arcfour.o arcfour_crypt.o

1563 BLOWFISHPROV_OBJS += blowfish.o blowfish_impl.o

1565 ECCPROV_OBJS += ecc.o ec.o ec2_163.o ec2_mont.o ecdecode.o ecl_mult.o \
1566                 ecp_384.o ecp_jac.o ec2_193.o ecl.o ecp_192.o ecp_521.o \
1567                 ecp_jm.o ec2_233.o ecl_curve.o ecp_224.o ecp_aff.o \
1568                 ecp_mont.o ec2_aff.o ec_naf.o ecl_gf.o ecp_256.o mp_gf2m.o \
1569                 mpi.o mplogic.o mpmontg.o mpprime.o oid.o \
1570                 secitem.o ec2_test.o ecp_test.o
```

```
1572 RSAPROV_OBJS += rsa.o rsa_impl.o pkcs1.o

1574 SWRANDPROV_OBJS += swrand.o

1576 #
1577 #                       kernel SSL
1578 #
1579 KSSL_OBJS +=    kssl.o kssl_ioctl.o

1581 KSSL_SOCKFIL_MOD_OBJS += ksslfilter.o ksslapi.o ksslrec.o

1583 #
1584 #                       misc. modules
1585 #

1587 C2AUDIT_OBJS += adr.o audit.o audit_event.o audit_io.o \
1588                 audit_path.o audit_start.o audit_syscalls.o audit_token.o \
1589                 audit_mem.o

1591 PCIC_OBJS +=    pcic.o

1593 RPCSEC_OBJS +=  secmod.o        sec_clnt.o      sec_svc.o       sec_gen.o \
1594                 auth_des.o      auth_kern.o     auth_none.o     auth_loopb.o\
1595                 authdesprt.o    authdesubr.o    authu_prot.o \
1596                 key_call.o      key_prot.o      svc_authu.o     svcauthdes.o

1598 RPCSEC_GSS_OBJS +=      rpcsec_gssmod.o rpcsec_gss.o rpcsec_gss_misc.o \
1599                 rpcsec_gss_utils.o svc_rpcsec_gss.o

1601 CONSCONFIG_OBJS += consconfig.o

1603 CONSCONFIG_DACF_OBJS  += consconfig_dacf.o consplat.o

1605 TEM_OBJS += tem.o tem_safe.o 6x10.o 7x14.o 12x22.o

1607 KBTRANS_OBJS +=                        \
1608                 kbtrans.o               \
1609                 kbtrans_keytables.o     \
1610                 kbtrans_polled.o        \
1611                 kbtrans_streams.o       \
1612                 usb_keytables.o

1614 KGSSD_OBJS +=   gssd_clnt_stubs.o gssd_handle.o gssd_prot.o \
1615                 gss_display_name.o gss_release_name.o gss_import_name.o \
1616                 gss_release_buffer.o gss_release_oid_set.o gen_oids.o gssdmod.o

1618 KGSSD_DERIVED_OBJS = gssd_xdr.o

1620 KGSS_DUMMY_OBJS += dmech.o

1622 KSOCKET_OBJS += ksocket.o ksocket_mod.o

1624 CRYPTO= cksumtypes.o decrypt.o encrypt.o encrypt_length.o etypes.o \
1625         nfold.o verify_checksum.o prng.o block_size.o make_checksum.o\
1626         checksum_length.o hmac.o default_state.o mandatory_sumtype.o

1628 # crypto/des
1629 CRYPTO_DES= f_cbc.o f_cksum.o f_parity.o weak_key.o d3_cbc.o ef_crypto.o

1631 CRYPTO_DK= checksum.o derive.o dk_decrypt.o dk_encrypt.o

1633 CRYPTO_ARCFOUR= k5_arcfour.o

1635 # crypto/enc_provider
1636 CRYPTO_ENC= des.o des3.o arcfour_provider.o aes_provider.o
```



```
1638 # crypto/hash_provider
1639 CRYPTO_HASH= hash_kef_generic.o hash_kmd5.o hash_crc32.o hash_ksha1.o

1641 # crypto/keyhash_provider
1642 CRYPTO_KEYHASH= descbc.o k5_kmd5des.o k_hmac_md5.o

1644 # crypto/crc32
1645 CRYPTO_CRC32= crc32.o

1647 # crypto/old
1648 CRYPTO_OLD= old_decrypt.o old_encrypt.o

1650 # crypto/raw
1651 CRYPTO_RAW= raw_decrypt.o raw_encrypt.o

1653 K5_KRB= kfree.o copy_key.o \
1654         parse.o init_ctx.o \
1655         ser_adata.o ser_addr.o \
1656         ser_auth.o ser_cksum.o \
1657         ser_key.o ser_princ.o \
1658         serialize.o unparse.o \
1659         ser_actx.o

1661 K5_OS=  timeofday.o toffset.o \
1662         init_os_ctx.o c_ustime.o

1664 SEAL=   seal.o unseal.o

1666 MECH=   delete_sec_context.o \
1667         import_sec_context.o \
1668         gssapi_krb5.o \
1669         k5seal.o k5unseal.o k5sealv3.o \
1670         ser_sctx.o \
1671         sign.o \
1672         util_crypt.o  \
1673         util_validate.o  util_ordering.o  \
1674         util_seqnum.o util_set.o util_seed.o \
1675         wrap_size_limit.o verify.o



1679 MECH_GEN= util_token.o


1682 KGSS_KRB5_OBJS += krb5mech.o \
1683         $(MECH) $(SEAL) $(MECH_GEN) \
1684         $(CRYPTO) $(CRYPTO_DES) $(CRYPTO_DK) $(CRYPTO_ARCFOUR) \
1685         $(CRYPTO_ENC) $(CRYPTO_HASH) \
1686         $(CRYPTO_KEYHASH) $(CRYPTO_CRC32) \
1687         $(CRYPTO_OLD) \
1688         $(CRYPTO_RAW) $(K5_KRB) $(K5_OS)

1690 DES_OBJS +=     des_crypt.o des_impl.o des_ks.o des_soft.o

1692 DLBOOT_OBJS +=  bootparam_xdr.o nfs_dlinet.o scan.o

1694 KRTLD_OBJS +=   kobj_bootflags.o getoptstr.o \
1695                 kobj.o kobj_kdi.o kobj_lm.o kobj_subr.o

1697 MOD_OBJS +=     modctl.o modsubr.o modsysfile.o modconf.o modhash.o

1699 STRPLUMB_OBJS += strplumb.o

1701 CPR_OBJS +=     cpr_driver.o cpr_dump.o \
1702                 cpr_main.o cpr_misc.o cpr_mod.o cpr_stat.o \
1703                 cpr_uthread.o
```

```
1705 PROF_OBJS +=    prf.o

1707 SE_OBJS += se_driver.o

1709 SYSACCT_OBJS += acct.o

1711 ACCTCTL_OBJS += acctctl.o

1713 EXACCTSYS_OBJS += exacctsys.o

1715 KAIO_OBJS += aio.o

1717 PCMCIA_OBJS += pcmcia.o cs.o cis.o cis_callout.o cis_handlers.o cis_params.o

1719 BUSRA_OBJS += busra.o

1721 PCS_OBJS += pcs.o

1723 PSET_OBJS +=    pset.o

1725 OHCI_OBJS += ohci.o ohci_hub.o ohci_polled.o

1727 UHCI_OBJS += uhci.o uhciutil.o uhcitgt.o uhcihub.o uhcipolled.o

1729 EHCI_OBJS += ehci.o ehci_hub.o ehci_xfer.o ehci_intr.o ehci_util.o ehci_polled.o

1731 HUBD_OBJS += hubd.o

1733 USB_MID_OBJS += usb_mid.o

1735 USB_IA_OBJS += usb_ia.o

1737 SCSA2USB_OBJS += scsa2usb.o usb_ms_bulkonly.o usb_ms_cbi.o

1739 IPF_OBJS += ip_fil_solaris.o fil.o solaris.o ip_state.o ip_frag.o ip_nat.o \
1740             ip_proxy.o ip_auth.o ip_pool.o ip_htable.o ip_lookup.o \
1741             ip_log.o misc.o ip_compat.o ip_nat6.o drand48.o

1743 IPD_OBJS += ipd.o

1745 IBD_OBJS +=     ibd.o ibd_cm.o

1747 EIBNX_OBJS +=   enx_main.o enx_hdlrs.o enx_ibt.o enx_log.o enx_fip.o \
1748                 enx_misc.o enx_q.o enx_ctl.o

1750 EOIB_OBJS +=    eib_adm.o eib_chan.o eib_cmn.o eib_ctl.o eib_data.o \
1751                 eib_fip.o eib_ibt.o eib_log.o eib_mac.o eib_main.o \
1752                 eib_rsrc.o eib_svc.o eib_vnic.o

1754 DLPISTUB_OBJS += dlpistub.o

1756 SDP_OBJS +=     sdpddi.o

1758 TRILL_OBJS +=   trill.o

1760 CTF_OBJS += ctf_create.o ctf_decl.o ctf_error.o ctf_hash.o ctf_labels.o \
1761         ctf_lookup.o ctf_open.o ctf_types.o ctf_util.o ctf_subr.o ctf_mod.o

1763 SMBIOS_OBJS += smb_error.o smb_info.o smb_open.o smb_subr.o smb_dev.o

1765 RPCIB_OBJS += rpcib.o

1767 KMDB_OBJS += kdrv.o

1769 AFE_OBJS += afe.o
```



```
1771 BGE_OBJS += bge_main2.o bge_chip2.o bge_kstats.o bge_log.o bge_ndd.o \
1772                 bge_atomic.o bge_mii.o bge_send.o bge_recv2.o bge_mii_5906.o

1774 DMFE_OBJS += dmfe_log.o dmfe_main.o dmfe_mii.o

1776 EFE_OBJS += efe.o

1778 ELXL_OBJS += elxl.o

1780 HME_OBJS += hme.o

1782 IXGB_OBJS += ixgb.o ixgb_atomic.o ixgb_chip.o ixgb_gld.o ixgb_kstats.o \
1783                 ixgb_log.o ixgb_ndd.o ixgb_rx.o ixgb_tx.o ixgb_xmii.o

1785 NGE_OBJS += nge_main.o nge_atomic.o nge_chip.o nge_ndd.o nge_kstats.o \
1786                 nge_log.o nge_rx.o nge_tx.o nge_xmii.o

1788 PCN_OBJS += pcn.o

1790 RGE_OBJS += rge_main.o rge_chip.o rge_ndd.o rge_kstats.o rge_log.o rge_rxtx.o

1792 URTW_OBJS += urtw.o

1794 ARN_OBJS += arn_hw.o arn_eeprom.o arn_mac.o arn_calib.o arn_ani.o arn_phy.o arn_
1795                 arn_main.o arn_recv.o arn_xmit.o arn_rc.o

1797 ATH_OBJS += ath_aux.o ath_main.o ath_osdep.o ath_rate.o

1799 ATU_OBJS += atu.o

1801 IPW_OBJS += ipw2100_hw.o ipw2100.o

1803 IWI_OBJS += ipw2200_hw.o ipw2200.o

1805 IWH_OBJS += iwh.o

1807 IWK_OBJS += iwk2.o

1809 IWP_OBJS += iwp.o

1811 MWL_OBJS += mwl.o

1813 MWLFW_OBJS += mwlfw_mode.o

1815 WPI_OBJS += wpi.o

1817 RAL_OBJS += rt2560.o ral_rate.o

1819 RUM_OBJS += rum.o

1821 RWD_OBJS += rt2661.o

1823 RWN_OBJS += rt2860.o

1825 UATH_OBJS += uath.o

1827 UATHFW_OBJS += uathfw_mod.o

1829 URAL_OBJS += ural.o

1831 RTW_OBJS += rtw.o smc93cx6.o rtwphy.o rtwphyio.o

1833 ZYD_OBJS += zyd.o zyd_usb.o zyd_hw.o zyd_fw.o

1835 MXFE_OBJS += mxfe.o
```

```
1837 MPTSAS_OBJS += mptsas.o mptsas_hash.o mptsas_impl.o mptsas_init.o \
1838                 mptsas_raid.o mptsas_smhba.o

1840 SFE_OBJS += sfe.o sfe_util.o

1842 BFE_OBJS += bfe.o

1844 BRIDGE_OBJS += bridge.o

1846 IDM_SHARED_OBJS += base64.o

1848 IDM_OBJS +=     $(IDM_SHARED_OBJS) \
1849                 idm.o idm_impl.o idm_text.o idm_conn_sm.o idm_so.o

1851 VR_OBJS += vr.o

1853 ATGE_OBJS += atge_main.o atge_l1e.o atge_mii.o atge_l1.o atge_l1c.o

1855 YGE_OBJS = yge.o

1857 SKD_OBJS = skd.o

1859 #
1860 #       Build up defines and paths.
1861 #
1862 LINT_DEFS       += -Dunix

1864 #
1865 #       This duality can be removed when the native and target compilers
1866 #       are the same (or at least recognize the same command line syntax!)
1867 #       It is a bug in the current compilation system that the assember
1868 #       can't process the -Y I, flag.
1869 #
1870 NATIVE_INC_PATH += $(INC_PATH) $(CCYFLAG)$(UTSBASE)/common
1871 AS_INC_PATH     += $(INC_PATH) -I$(UTSBASE)/common
1872 INCLUDE_PATH    += $(INC_PATH) $(CCYFLAG)$(UTSBASE)/common

1874 PCIEB_OBJS += pcieb.o

1876 #       Chelsio N110 10G NIC driver module
1877 #
1878 CH_OBJS = ch.o glue.o pe.o sge.o

1880 CH_COM_OBJS =   ch_mac.o ch_subr.o cspi.o espi.o ixf1010.o mc3.o mc4.o mc5.o \
1881                 mv88e1xxx.o mv88x201x.o my3126.o pm3393.o tp.o ulp.o \
1882                 vsc7321.o vsc7326.o xpak.o

1884 #
1885 #       Chelsio Terminator 4 10G NIC nexus driver module
1886 #
1887 CXGBE_FW_OBJS  =        t4_fw.o t4_cfg.o
1888 CXGBE_COM_OBJS =        t4_hw.o common.o
1889 CXGBE_NEX_OBJS =        t4_nexus.o t4_sge.o t4_mac.o t4_ioctl.o shared.o \
1890                         t4_l2t.o adapter.o osdep.o

1892 #
1893 #       Chelsio Terminator 4 10G NIC driver module
1894 #
1895 CXGBE_OBJS =    cxgbe.o

1897 #
1898 #       PCI strings file
1899 #
1900 PCI_STRING_OBJS = pci_strings.o
```



```
1902 NET_DACF_OBJS += net_dacf.o

1904 #
1905 #       Xframe 10G NIC driver module
1906 #
1907 XGE_OBJS = xge.o xgell.o

1909 XGE_HAL_OBJS =  xgehal-channel.o xgehal-fifo.o xgehal-ring.o  xgehal-config.o \
1910                 xgehal-driver.o  xgehal-mm.o xgehal-stats.o  xgehal-device.o \
1911                 xge-queue.o  xgehal-mgmt.o xgehal-mgmtaux.o

1913 #
1914 #       e1000/igb common objs
1915 #
1916 #       Historically e1000g and igb had separate copies of all of the common
1917 #       code. At this time while they are now sharing the same copy of it, they
1918 #       are building it into their own modules which is due to the differences
1919 #       in the osdep and debug portions of their code.
1920 #
1921 E1000API_OBJS += e1000_80003es2lan.o e1000_82540.o e1000_82541.o e1000_82542.o \
1922                 e1000_82543.o e1000_82571.o e1000_api.o e1000_ich8lan.o \
1923                 e1000_mac.o e1000_manage.o e1000_nvm.o e1000_phy.o \
1924                 e1000_82575.o e1000_i210.o e1000_mbx.o e1000_vf.o

1926 #
1927 #       e1000g module
1928 #
1929 E1000G_OBJS +=  e1000g_debug.o e1000g_main.o e1000g_alloc.o \
1930                 e1000g_tx.o e1000g_rx.o e1000g_stat.o \
1931                 e1000g_osdep.o e1000g_workarounds.o
1932

1934 #
1935 #       Intel 82575 1G NIC driver module
1936 #
1937 IGB_OBJS =      igb_buf.o igb_debug.o igb_gld.o igb_log.o igb_main.o \
1938                 igb_rx.o igb_stat.o igb_tx.o igb_osdep.o

1940 #
1941 #       Intel Pro/100 NIC driver module
1942 #
1943 IPRB_OBJS =     iprb.o

1945 #
1946 #       Intel 10GbE PCIE NIC driver module
1947 #
1948 IXGBE_OBJS =    ixgbe_82598.o ixgbe_82599.o ixgbe_api.o         \
1949                 ixgbe_common.o ixgbe_phy.o                      \
1950                 ixgbe_buf.o ixgbe_debug.o ixgbe_gld.o           \
1951                 ixgbe_log.o ixgbe_main.o                        \
1952                 ixgbe_osdep.o ixgbe_rx.o ixgbe_stat.o           \
1953                 ixgbe_tx.o  ixgbe_x540.o ixgbe_mbx.o

1955 #
1956 #       NIU 10G/1G driver module
1957 #
1958 NXGE_OBJS =     nxge_mac.o nxge_ipp.o nxge_rxdma.o              \
1959                 nxge_txdma.o nxge_txc.o nxge_main.o             \
1960                 nxge_hw.o nxge_fzc.o nxge_virtual.o             \
1961                 nxge_send.o nxge_classify.o nxge_fflp.o         \
1962                 nxge_fflp_hash.o nxge_ndd.o nxge_kstats.o       \
1963                 nxge_zcp.o nxge_fm.o nxge_espc.o nxge_hv.o      \
1964                 nxge_hio.o nxge_hio_guest.o nxge_intr.o

1966 NXGE_NPI_OBJS = \
1967                 npi.o npi_mac.o npi_ipp.o                       \
```

```
1968                 npi_txdma.o npi_rxdma.o npi_txc.o              \
1969                 npi_zcp.o npi_espc.o npi_fflp.o                \
1970                 npi_vir.o

1972 NXGE_HCALL_OBJS =       \
1973                 nxge_hcall.o

1975 #
1976 # Virtio modules
1977 #

1979 # Virtio core
1980 VIRTIO_OBJS = virtio.o

1982 # Virtio block driver
1983 VIOBLK_OBJS = vioblk.o

1985 #
1986 #       kiconv modules
1987 #
1988 KICONV_EMEA_OBJS += kiconv_emea.o

1990 KICONV_JA_OBJS += kiconv_ja.o

1992 KICONV_KO_OBJS += kiconv_cck_common.o kiconv_ko.o

1994 KICONV_SC_OBJS += kiconv_cck_common.o kiconv_sc.o

1996 KICONV_TC_OBJS += kiconv_cck_common.o kiconv_tc.o

1998 #
1999 #       AAC module
2000 #
2001 AAC_OBJS = aac.o aac_ioctl.o

2003 #
2004 #       sdcard modules
2005 #
2006 SDA_OBJS =      sda_cmd.o sda_host.o sda_init.o sda_mem.o sda_mod.o sda_slot.o
2007 SDHOST_OBJS =   sdhost.o

2009 #
2010 #       hxge 10G driver module
2011 #
2012 HXGE_OBJS =     hxge_main.o hxge_vmac.o hxge_send.o             \
2013                 hxge_txdma.o hxge_rxdma.o hxge_virtual.o        \
2014                 hxge_fm.o hxge_fzc.o hxge_hw.o hxge_kstats.o    \
2015                 hxge_ndd.o hxge_pfc.o                           \
2016                 hpi.o hpi_vmac.o hpi_rxdma.o hpi_txdma.o        \
2017                 hpi_vir.o hpi_pfc.o

2019 #
2020 #       MEGARAID_SAS module
2021 #
2022 MEGA_SAS_OBJS = megaraid_sas.o

2024 #
2025 #       MR_SAS module
2026 #
2027 MR_SAS_OBJS = ld_pd_map.o mr_sas.o mr_sas_tbolt.o mr_sas_list.o

2029 #
2030 #       CPQARY3 module
2031 #
2032 CPQARY3_OBJS =  cpqary3.o cpqary3_noe.o cpqary3_talk2ctlr.o     \
2033                 cpqary3_isr.o cpqary3_transport.o cpqary3_mem.o \
```



```
2034                 cpqary3_scsi.o cpqary3_util.o cpqary3_ioctl.o   \
2035                 cpqary3_bd.o

2037 #
2038 #       ISCSI_INITIATOR module
2039 #
2040 ISCSI_INITIATOR_OBJS =  chap.o iscsi_io.o iscsi_thread.o        \
2041                         iscsi_ioctl.o iscsid.o iscsi.o          \
2042                         iscsi_login.o isns_client.o iscsiAuthClient.o   \
2043                         iscsi_lun.o iscsiAuthClientGlue.o       \
2044                         iscsi_net.o nvfile.o iscsi_cmd.o        \
2045                         iscsi_queue.o persistent.o iscsi_conn.o \
2046                         iscsi_sess.o radius_auth.o iscsi_crc.o  \
2047                         iscsi_stats.o radius_packet.o iscsi_doorclt.o   \
2048                         iscsi_targetparam.o utils.o kifconf.o

2050 #
2051 #       ntxn 10Gb/1Gb NIC driver module
2052 #
2053 NTXN_OBJS =     unm_nic_init.o unm_gem.o unm_nic_hw.o unm_ndd.o \
2054                         unm_nic_main.o unm_nic_isr.o unm_nic_ctx.o niu.o

2056 #
2057 #       Myricom 10Gb NIC driver module
2058 #
2059 MYRI10GE_OBJS = myri10ge.o myri10ge_lro.o

2061 #       nulldriver module
2062 #
2063 NULLDRIVER_OBJS =       nulldriver.o

2065 TPM_OBJS =      tpm.o tpm_hcall.o

2067 #
2068 #       BNXE objects
2069 #
2070 BNXE_OBJS +=    bnxe_cfg.o              \
2071                 bnxe_fcoe.o             \
2072                 bnxe_debug.o            \
2073                 bnxe_gld.o              \
2074                 bnxe_hw.o               \
2075                 bnxe_intr.o             \
2076                 bnxe_kstat.o            \
2077                 bnxe_lock.o             \
2078                 bnxe_main.o             \
2079                 bnxe_mm.o               \
2080                 bnxe_mm_l4.o            \
2081                 bnxe_mm_l5.o            \
2082                 bnxe_rr.o               \
2083                 bnxe_rx.o               \
2084                 bnxe_timer.o            \
2085                 bnxe_tx.o               \
2086                 bnxe_workq.o            \
2087                 bnxe_clc.o              \
2088                 ecore_sp_verbs.o        \
2089                 bnxe_context.o          \
2090                 57710_init_values.o     \
2091                 57711_init_values.o     \
2092                 57712_init_values.o     \
2093                 bnxe_fw_funcs.o         \
2094                 bnxe_hw_debug.o         \
2095                 lm_l4fp.o               \
2096                 lm_l4rx.o               \
2097                 lm_l4sp.o               \
2098                 lm_l4tx.o               \
2099                 lm_l5.o                 \
```

```
2100                    lm_l5sp.o               \
2101                    lm_dcbx.o               \
2102                    lm_devinfo.o            \
2103                    lm_dmae.o               \
2104                    lm_er.o                 \
2105                    lm_hw_access.o          \
2106                    lm_hw_attn.o            \
2107                    lm_hw_init_reset.o      \
2108                    lm_main.o               \
2109                    lm_mcp.o                \
2110                    lm_niv.o                \
2111                    lm_nvram.o              \
2112                    lm_phy.o                \
2113                    lm_power.o              \
2114                    lm_recv.o               \
2115                    lm_resc.o               \
2116                    lm_sb.o                 \
2117                    lm_send.o               \
2118                    lm_sp.o                 \
2119                    lm_dcbx_mp.o            \
2120                    lm_sp_req_mgr.o         \
2121                    lm_stats.o              \
2122                    lm_util.o
```

****
29374 Fri May 8 18:10:22 2015
new/usr/src/uts/common/cpr/cpr_dump.c
patch lower-case-segops
****
_____***unchanged_portion_omitted_***

```
 661 /*
 662  * Count pages within each kernel segment; call cpr_sparse_seg_check()
 663  * to find out whether a sparsely filled segment needs special
 664  * treatment (e.g. kvseg).
 665  * Todo: A "segop_cpr" like segop_dump should be introduced, the cpr
 665  * Todo: A "SEGOP_CPR" like SEGOP_DUMP should be introduced, the cpr
 666  *       module shouldn't need to know segment details like if it is
 667  *       sparsely filled or not (makes kseg_table obsolete).
 668  */
 669 pgcnt_t
 670 cpr_count_seg_pages(int mapflag, bitfunc_t bitfunc)
 671 {
 672         struct seg *segp;
 673         pgcnt_t pages;
 674         ksegtbl_entry_t *ste;

 676         pages = 0;
 677         for (segp = AS_SEGFIRST(&kas); segp; segp = AS_SEGNEXT(&kas, segp)) {
 678                 if (ste = cpr_sparse_seg_check(segp)) {
 679                         pages += (ste->st_fcn)(mapflag, bitfunc, segp);
 680                 } else {
 681                         pages += cpr_count_pages(segp->s_base,
 682                             segp->s_size, mapflag, bitfunc, DBG_SHOWRANGE);
 683                 }
 684         }

 686         return (pages);
 687 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   66813 Fri May  8 18:10:22 2015
new/usr/src/uts/common/disp/disp.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_
  75 static void     disp_dq_alloc(struct disp_queue_info *dptr, int numpris,
  76     disp_t *dp);
  77 static void     disp_dq_assign(struct disp_queue_info *dptr, int numpris);
  78 static void     disp_dq_free(struct disp_queue_info *dptr);

  80 /* platform-specific routine to call when processor is idle */
  81 static void     generic_idle_cpu();
  82 void            (*idle_cpu)() = generic_idle_cpu;

  84 /* routines invoked when a CPU enters/exits the idle loop */
  85 static void     idle_enter();
  86 static void     idle_exit();

  88 /* platform-specific routine to call when thread is enqueued */
  89 static void     generic_enq_thread(cpu_t *, int);
  90 void            (*disp_enq_thread)(cpu_t *, int) = generic_enq_thread;

  92 pri_t   kpreemptpri;            /* priority where kernel preemption applies */
  93 pri_t   upreemptpri = 0;        /* priority where normal preemption applies */
  94 pri_t   intr_pri;               /* interrupt thread priority base level */

  96 #define KPQPRI  -1              /* pri where cpu affinity is dropped for kpq */
  97 pri_t   kpqpri = KPQPRI;        /* can be set in /etc/system */
  98 disp_t  cpu0_disp;              /* boot CPU's dispatch queue */
  99 disp_lock_t     swapped_lock;   /* lock swapped threads and swap queue */
  99 int     nswapped;               /* total number of swapped threads */
 101 void    disp_swapped_enq(kthread_t *tp);
 100 static void     disp_swapped_setrun(kthread_t *tp);
 101 static void     cpu_resched(cpu_t *cp, pri_t tpri);

 103 /*
 104  * If this is set, only interrupt threads will cause kernel preemptions.
 105  * This is done by changing the value of kpreemptpri.  kpreemptpri
 106  * will either be the max sysclass pri + 1 or the min interrupt pri.
 107  */
 108 int     only_intr_kpreempt;

 110 extern void set_idle_cpu(int cpun);
 111 extern void unset_idle_cpu(int cpun);
 112 static void setkpdq(kthread_t *tp, int borf);
 113 #define SETKP_BACK      0
 114 #define SETKP_FRONT     1
 115 /*
 116  * Parameter that determines how recently a thread must have run
 117  * on the CPU to be considered loosely-bound to that CPU to reduce
 118  * cold cache effects.  The interval is in hertz.
 119  */
 120 #define RECHOOSE_INTERVAL 3
 121 int     rechoose_interval = RECHOOSE_INTERVAL;

 123 /*
 124  * Parameter that determines how long (in nanoseconds) a thread must
 125  * be sitting on a run queue before it can be stolen by another CPU
 126  * to reduce migrations.  The interval is in nanoseconds.
```



```
 127  *
 128  * The nosteal_nsec should be set by platform code cmp_set_nosteal_interval()
 129  * to an appropriate value.  nosteal_nsec is set to NOSTEAL_UNINITIALIZED
 130  * here indicating it is uninitiallized.
 131  * Setting nosteal_nsec to 0 effectively disables the nosteal 'protection'.
 132  *
 133  */
 134 #define NOSTEAL_UNINITIALIZED   (-1)
 135 hrtime_t nosteal_nsec = NOSTEAL_UNINITIALIZED;
 136 extern void cmp_set_nosteal_interval(void);

 138 id_t    defaultcid;     /* system "default" class; see dispadmin(1M) */

 140 disp_lock_t     transition_lock;        /* lock on transitioning threads */
 141 disp_lock_t     stop_lock;              /* lock on stopped threads */

 143 static void     cpu_dispqalloc(int numpris);

 145 /*
 146  * This gets returned by disp_getwork/disp_getbest if we couldn't steal
 147  * a thread because it was sitting on its run queue for a very short
 148  * period of time.
 149  */
 150 #define T_DONTSTEAL     (kthread_t *)(-1) /* returned by disp_getwork/getbest */

 152 static kthread_t        *disp_getwork(cpu_t *to);
 153 static kthread_t        *disp_getbest(disp_t *from);
 154 static kthread_t        *disp_ratify(kthread_t *tp, disp_t *kpq);

 156 void    swtch_to(kthread_t *);

 158 /*
 159  * dispatcher and scheduler initialization
 160  */

 162 /*
 163  * disp_setup - Common code to calculate and allocate dispatcher
 164  *              variables and structures based on the maximum priority.
 165  */
 166 static void
 167 disp_setup(pri_t maxglobpri, pri_t oldnglobpris)
 168 {
 169         pri_t   newnglobpris;

 171         ASSERT(MUTEX_HELD(&cpu_lock));

 173         newnglobpris = maxglobpri + 1 + LOCK_LEVEL;

 175         if (newnglobpris > oldnglobpris) {
 176                 /*
 177                  * Allocate new kp queues for each CPU partition.
 178                  */
 179                 cpupart_kpqalloc(newnglobpris);

 181                 /*
 182                  * Allocate new dispatch queues for each CPU.
 183                  */
 184                 cpu_dispqalloc(newnglobpris);

 186                 /*
 187                  * compute new interrupt thread base priority
 188                  */
 189                 intr_pri = maxglobpri;
 190                 if (only_intr_kpreempt) {
 191                         kpreemptpri = intr_pri + 1;
 192                         if (kpqpri == KPQPRI)
```

```
193                                 kpqpri = kpreemptpri;
194                 }
195                 v.v_nglobpris = newnglobpris;
196         }
197 }
_____unchanged_portion_omitted_

694 extern kthread_t *thread_unpin();

696 /*
697  * disp() - find the highest priority thread for this processor to run, and
698  * set it in TS_ONPROC state so that resume() can be called to run it.
699  */
700 static kthread_t *
701 disp()
702 {
703         cpu_t           *cpup;
704         disp_t          *dp;
705         kthread_t       *tp;
706         dispq_t         *dq;
707         int             maxrunword;
708         pri_t           pri;
709         disp_t          *kpq;

711         TRACE_0(TR_FAC_DISP, TR_DISP_START, "disp_start");

713         cpup = CPU;
714         /*
715          * Find the highest priority loaded, runnable thread.
716          */
717         dp = cpup->cpu_disp;

719 reschedule:
720         /*
721          * If there is more important work on the global queue with a better
722          * priority than the maximum on this CPU, take it now.
723          */
724         kpq = &cpup->cpu_part->cp_kp_queue;
725         while ((pri = kpq->disp_maxrunpri) >= 0 &&
726             pri >= dp->disp_maxrunpri &&
727             (cpup->cpu_flags & CPU_OFFLINE) == 0 &&
728             (tp = disp_getbest(kpq)) != NULL) {
729                 if (disp_ratify(tp, kpq) != NULL) {
730                         TRACE_1(TR_FAC_DISP, TR_DISP_END,
731                             "disp_end:tid %p", tp);
732                         return (tp);
733                 }
734         }

736         disp_lock_enter(&dp->disp_lock);
737         pri = dp->disp_maxrunpri;

739         /*
740          * If there is nothing to run, look at what's runnable on other queues.
741          * Choose the idle thread if the CPU is quiesced.
742          * Note that CPUs that have the CPU_OFFLINE flag set can still run
743          * interrupt threads, which will be the only threads on the CPU's own
744          * queue, but cannot run threads from other queues.
745          */
746         if (pri == -1) {
747                 if (!(cpup->cpu_flags & CPU_OFFLINE)) {
748                         disp_lock_exit(&dp->disp_lock);
749                         if ((tp = disp_getwork(cpup)) == NULL ||
750                             tp == T_DONTSTEAL) {
751                                 tp = cpup->cpu_idle_thread;
752                                 (void) splhigh();
```



```
753                                 THREAD_ONPROC(tp, cpup);
754                                 cpup->cpu_dispthread = tp;
755                                 cpup->cpu_dispatch_pri = -1;
756                                 cpup->cpu_runrun = cpup->cpu_kprunrun = 0;
757                                 cpup->cpu_chosen_level = -1;
758                         }
759                 } else {
760                         disp_lock_exit_high(&dp->disp_lock);
761                         tp = cpup->cpu_idle_thread;
762                         THREAD_ONPROC(tp, cpup);
763                         cpup->cpu_dispthread = tp;
764                         cpup->cpu_dispatch_pri = -1;
765                         cpup->cpu_runrun = cpup->cpu_kprunrun = 0;
766                         cpup->cpu_chosen_level = -1;
767                 }
768                 TRACE_1(TR_FAC_DISP, TR_DISP_END,
769                     "disp_end:tid %p", tp);
770                 return (tp);
771         }

773         dq = &dp->disp_q[pri];
774         tp = dq->dq_first;

776         ASSERT(tp != NULL);
779         ASSERT(tp->t_schedflag & TS_LOAD);      /* thread must be swapped in */

778         DTRACE_SCHED2(dequeue, kthread_t *, tp, disp_t *, dp);

780         /*
781          * Found it so remove it from queue.
782          */
783         dp->disp_nrunnable--;
784         dq->dq_sruncnt--;
785         if ((dq->dq_first = tp->t_link) == NULL) {
786                 ulong_t *dqactmap = dp->disp_qactmap;

788                 ASSERT(dq->dq_sruncnt == 0);
789                 dq->dq_last = NULL;

791                 /*
792                  * The queue is empty, so the corresponding bit needs to be
793                  * turned off in dqactmap.   If nrunnable != 0 just took the
794                  * last runnable thread off the
795                  * highest queue, so recompute disp_maxrunpri.
796                  */
797                 maxrunword = pri >> BT_ULSHIFT;
798                 dqactmap[maxrunword] &= ~BT_BIW(pri);

800                 if (dp->disp_nrunnable == 0) {
801                         dp->disp_max_unbound_pri = -1;
802                         dp->disp_maxrunpri = -1;
803                 } else {
804                         int ipri;

806                         ipri = bt_gethighbit(dqactmap, maxrunword);
807                         dp->disp_maxrunpri = ipri;
808                         if (ipri < dp->disp_max_unbound_pri)
809                                 dp->disp_max_unbound_pri = ipri;
810                 }
811         } else {
812                 tp->t_link = NULL;
813         }

818         /*
819          * Set TS_DONT_SWAP flag to prevent another processor from swapping
820          * out this thread before we have a chance to run it.
```

```
821          * While running, it is protected against swapping by t_lock.
822          */
823         tp->t_schedflag |= TS_DONT_SWAP;
815         cpup->cpu_dispthread = tp;              /* protected by spl only */
816         cpup->cpu_dispatch_pri = pri;
817         ASSERT(pri == DISP_PRIO(tp));
818         thread_onproc(tp, cpup);                /* set t_state to TS_ONPROC */
819         disp_lock_exit_high(&dp->disp_lock);    /* drop run queue lock */

821         ASSERT(tp != NULL);
822         TRACE_1(TR_FAC_DISP, TR_DISP_END,
823             "disp_end:tid %p", tp);

825         if (disp_ratify(tp, kpq) == NULL)
826                 goto reschedule;

828         return (tp);
829 }
```

unchanged_portion_omitted_

```
1142 /*
1143  * setbackdq() keeps runqs balanced such that the difference in length
1144  * between the chosen runq and the next one is no more than RUNQ_MAX_DIFF.
1145  * For threads with priorities below RUNQ_MATCH_PRI levels, the runq's lengths
1146  * must match.  When per-thread TS_RUNQMATCH flag is set, setbackdq() will
1147  * try to keep runqs perfectly balanced regardless of the thread priority.
1148  */
1149 #define RUNQ_MATCH_PRI  16      /* pri below which queue lengths must match */
1150 #define RUNQ_MAX_DIFF   2       /* maximum runq length difference */
1151 #define RUNQ_LEN(cp, pri)       ((cp)->cpu_disp->disp_q[pri].dq_sruncnt)

1153 /*
1154  * Macro that evaluates to true if it is likely that the thread has cache
1155  * warmth. This is based on the amount of time that has elapsed since the
1156  * thread last ran. If that amount of time is less than "rechoose_interval"
1157  * ticks, then we decide that the thread has enough cache warmth to warrant
1158  * some affinity for t->t_cpu.
1159  */
1160 #define THREAD_HAS_CACHE_WARMTH(thread) \
1161         ((thread == curthread) ||       \
1162         ((ddi_get_lbolt() - thread->t_disp_time) <= rechoose_interval))
1163 /*
1164  * Put the specified thread on the back of the dispatcher
1165  * queue corresponding to its current priority.
1166  *
1167  * Called with the thread in transition, onproc or stopped state
1168  * and locked (transition implies locked) and at high spl.
1169  * Returns with the thread in TS_RUN state and still locked.
1170  */
1171 void
1172 setbackdq(kthread_t *tp)
1173 {
1174         dispq_t *dq;
1175         disp_t          *dp;
1176         cpu_t           *cp;
1177         pri_t           tpri;
1178         int             bound;
1179         boolean_t       self;

1181         ASSERT(THREAD_LOCK_HELD(tp));
1182         ASSERT((tp->t_schedflag & TS_ALLSTART) == 0);
1183         ASSERT(!thread_on_queue(tp));   /* make sure tp isn't on a runq */

1194         /*
1195          * If thread is "swapped" or on the swap queue don't
1196          * queue it, but wake sched.
```



```
1197          */
1198         if ((tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ)) != TS_LOAD) {
1199                 disp_swapped_setrun(tp);
1200                 return;
1201         }

1185         self = (tp == curthread);

1187         if (tp->t_bound_cpu || tp->t_weakbound_cpu)
1188                 bound = 1;
1189         else
1190                 bound = 0;

1192         tpri = DISP_PRIO(tp);
1193         if (ncpus == 1)
1194                 cp = tp->t_cpu;
1195         else if (!bound) {
1196                 if (tpri >= kpqpri) {
1197                         setkpdq(tp, SETKP_BACK);
1198                         return;
1199                 }

1201                 /*
1202                  * We'll generally let this thread continue to run where
1203                  * it last ran...but will consider migration if:
1204                  * - We thread probably doesn't have much cache warmth.
1205                  * - The CPU where it last ran is the target of an offline
1206                  *   request.
1207                  * - The thread last ran outside it's home lgroup.
1208                  */
1209                 if ((!THREAD_HAS_CACHE_WARMTH(tp)) ||
1210                     (tp->t_cpu == cpu_inmotion)) {
1211                         cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri, NULL);
1212                 } else if (!LGRP_CONTAINS_CPU(tp->t_lpl->lpl_lgrp, tp->t_cpu)) {
1213                         cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri,
1214                             self ? tp->t_cpu : NULL);
1215                 } else {
1216                         cp = tp->t_cpu;
1217                 }

1219                 if (tp->t_cpupart == cp->cpu_part) {
1220                         int     qlen;

1222                         /*
1223                          * Perform any CMT load balancing
1224                          */
1225                         cp = cmt_balance(tp, cp);

1227                         /*
1228                          * Balance across the run queues
1229                          */
1230                         qlen = RUNQ_LEN(cp, tpri);
1231                         if (tpri >= RUNQ_MATCH_PRI &&
1232                             !(tp->t_schedflag & TS_RUNQMATCH))
1233                                 qlen -= RUNQ_MAX_DIFF;
1234                         if (qlen > 0) {
1235                                 cpu_t *newcp;

1237                                 if (tp->t_lpl->lpl_lgrpid == LGRP_ROOTID) {
1238                                         newcp = cp->cpu_next_part;
1239                                 } else if ((newcp = cp->cpu_next_lpl) == cp) {
1240                                         newcp = cp->cpu_next_part;
1241                                 }

1243                                 if (RUNQ_LEN(newcp, tpri) < qlen) {
1244                                         DTRACE_PROBE3(runq__balance,
```

```
1245                                            kthread_t *, tp,
1246                                            cpu_t *, cp, cpu_t *, newcp);
1247                                    cp = newcp;
1248                            }
1249                    }
1250            } else {
1251                    /*
1252                     * Migrate to a cpu in the new partition.
1253                     */
1254                    cp = disp_lowpri_cpu(tp->t_cpupart->cp_cpulist,
1255                        tp->t_lpl, tp->t_pri, NULL);
1256            }
1257            ASSERT((cp->cpu_flags & CPU_QUIESCED) == 0);
1258    } else {
1259            /*
1260             * It is possible that t_weakbound_cpu != t_bound_cpu (for
1261             * a short time until weak binding that existed when the
1262             * strong binding was established has dropped) so we must
1263             * favour weak binding over strong.
1264             */
1265            cp = tp->t_weakbound_cpu ?
1266                tp->t_weakbound_cpu : tp->t_bound_cpu;
1267    }
1268    /*
1269     * A thread that is ONPROC may be temporarily placed on the run queue
1270     * but then chosen to run again by disp.  If the thread we're placing on
1271     * the queue is in TS_ONPROC state, don't set its t_waitrq until a
1272     * replacement process is actually scheduled in swtch().  In this
1273     * situation, curthread is the only thread that could be in the ONPROC
1274     * state.
1275     */
1276    if ((!self) && (tp->t_waitrq == 0)) {
1277            hrtime_t curtime;

1279            curtime = gethrtime_unscaled();
1280            (void) cpu_update_pct(tp, curtime);
1281            tp->t_waitrq = curtime;
1282    } else {
1283            (void) cpu_update_pct(tp, gethrtime_unscaled());
1284    }

1286    dp = cp->cpu_disp;
1287    disp_lock_enter_high(&dp->disp_lock);

1289    DTRACE_SCHED3(enqueue, kthread_t *, tp, disp_t *, dp, int, 0);
1290    TRACE_3(TR_FAC_DISP, TR_BACKQ, "setbackdq:pri %d cpu %p tid %p",
1291        tpri, cp, tp);

1293 #ifndef NPROBE
1294    /* Kernel probe */
1295    if (tnf_tracing_active)
1296            tnf_thread_queue(tp, cp, tpri);
1297 #endif /* NPROBE */

1299    ASSERT(tpri >= 0 && tpri < dp->disp_npri);

1301    THREAD_RUN(tp, &dp->disp_lock);         /* set t_state to TS_RUN */
1302    tp->t_disp_queue = dp;
1303    tp->t_link = NULL;

1305    dq = &dp->disp_q[tpri];
1306    dp->disp_nrunnable++;
1307    if (!bound)
1308            dp->disp_steal = 0;
1309    membar_enter();
```



```
1311    if (dq->dq_sruncnt++ != 0) {
1312            ASSERT(dq->dq_first != NULL);
1313            dq->dq_last->t_link = tp;
1314            dq->dq_last = tp;
1315    } else {
1316            ASSERT(dq->dq_first == NULL);
1317            ASSERT(dq->dq_last == NULL);
1318            dq->dq_first = dq->dq_last = tp;
1319            BT_SET(dp->disp_qactmap, tpri);
1320            if (tpri > dp->disp_maxrunpri) {
1321                    dp->disp_maxrunpri = tpri;
1322                    membar_enter();
1323                    cpu_resched(cp, tpri);
1324            }
1325    }

1327    if (!bound && tpri > dp->disp_max_unbound_pri) {
1328            if (self && dp->disp_max_unbound_pri == -1 && cp == CPU) {
1329                    /*
1330                     * If there are no other unbound threads on the
1331                     * run queue, don't allow other CPUs to steal
1332                     * this thread while we are in the middle of a
1333                     * context switch. We may just switch to it
1334                     * again right away. CPU_DISP_DONTSTEAL is cleared
1335                     * in swtch and swtch_to.
1336                     */
1337                    cp->cpu_disp_flags |= CPU_DISP_DONTSTEAL;
1338            }
1339            dp->disp_max_unbound_pri = tpri;
1340    }
1341    (*disp_enq_thread)(cp, bound);
1342 }

1344 /*
1345  * Put the specified thread on the front of the dispatcher
1346  * queue corresponding to its current priority.
1347  *
1348  * Called with the thread in transition, onproc or stopped state
1349  * and locked (transition implies locked) and at high spl.
1350  * Returns with the thread in TS_RUN state and still locked.
1351  */
1352 void
1353 setfrontdq(kthread_t *tp)
1354 {
1355    disp_t          *dp;
1356    dispq_t         *dq;
1357    cpu_t           *cp;
1358    pri_t           tpri;
1359    int             bound;

1361    ASSERT(THREAD_LOCK_HELD(tp));
1362    ASSERT((tp->t_schedflag & TS_ALLSTART) == 0);
1363    ASSERT(!thread_on_queue(tp));   /* make sure tp isn't on a runq */

1383    /*
1384     * If thread is "swapped" or on the swap queue don't
1385     * queue it, but wake sched.
1386     */
1387    if ((tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ)) != TS_LOAD) {
1388            disp_swapped_setrun(tp);
1389            return;
1390    }

1365    if (tp->t_bound_cpu || tp->t_weakbound_cpu)
1366            bound = 1;
1367    else
```

```
1368                bound = 0;

1370        tpri = DISP_PRIO(tp);
1371        if (ncpus == 1)
1372                cp = tp->t_cpu;
1373        else if (!bound) {
1374                if (tpri >= kpqpri) {
1375                        setkpdq(tp, SETKP_FRONT);
1376                        return;
1377                }
1378                cp = tp->t_cpu;
1379                if (tp->t_cpupart == cp->cpu_part) {
1380                        /*
1381                         * We'll generally let this thread continue to run
1382                         * where it last ran, but will consider migration if:
1383                         * - The thread last ran outside it's home lgroup.
1384                         * - The CPU where it last ran is the target of an
1385                         *   offline request (a thread_nomigrate() on the in
1386                         *   motion CPU relies on this when forcing a preempt).
1387                         * - The thread isn't the highest priority thread where
1388                         *   it last ran, and it is considered not likely to
1389                         *   have significant cache warmth.
1390                         */
1391                        if ((!LGRP_CONTAINS_CPU(tp->t_lpl->lpl_lgrp, cp)) ||
1392                            (cp == cpu_inmotion)) {
1393                                cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri,
1394                                    (tp == curthread) ? cp : NULL);
1395                        } else if ((tpri < cp->cpu_disp->disp_maxrunpri) &&
1396                            (!THREAD_HAS_CACHE_WARMTH(tp))) {
1397                                cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri,
1398                                    NULL);
1399                        }
1400                } else {
1401                        /*
1402                         * Migrate to a cpu in the new partition.
1403                         */
1404                        cp = disp_lowpri_cpu(tp->t_cpupart->cp_cpulist,
1405                            tp->t_lpl, tp->t_pri, NULL);
1406                }
1407                ASSERT((cp->cpu_flags & CPU_QUIESCED) == 0);
1408        } else {
1409                /*
1410                 * It is possible that t_weakbound_cpu != t_bound_cpu (for
1411                 * a short time until weak binding that existed when the
1412                 * strong binding was established has dropped) so we must
1413                 * favour weak binding over strong.
1414                 */
1415                cp = tp->t_weakbound_cpu ?
1416                    tp->t_weakbound_cpu : tp->t_bound_cpu;
1417        }

1419        /*
1420         * A thread that is ONPROC may be temporarily placed on the run queue
1421         * but then chosen to run again by disp.  If the thread we're placing on
1422         * the queue is in TS_ONPROC state, don't set its t_waitrq until a
1423         * replacement process is actually scheduled in swtch().  In this
1424         * situation, curthread is the only thread that could be in the ONPROC
1425         * state.
1426         */
1427        if ((tp != curthread) && (tp->t_waitrq == 0)) {
1428                hrtime_t curtime;

1430                curtime = gethrtime_unscaled();
1431                (void) cpu_update_pct(tp, curtime);
1432                tp->t_waitrq = curtime;
1433        } else {
```



```
1434                (void) cpu_update_pct(tp, gethrtime_unscaled());
1435        }

1437        dp = cp->cpu_disp;
1438        disp_lock_enter_high(&dp->disp_lock);

1440        TRACE_2(TR_FAC_DISP, TR_FRONTQ, "frontq:pri %d tid %p", tpri, tp);
1441        DTRACE_SCHED3(enqueue, kthread_t *, tp, disp_t *, dp, int, 1);

1443 #ifndef NPROBE
1444        /* Kernel probe */
1445        if (tnf_tracing_active)
1446                tnf_thread_queue(tp, cp, tpri);
1447 #endif /* NPROBE */

1449        ASSERT(tpri >= 0 && tpri < dp->disp_npri);

1451        THREAD_RUN(tp, &dp->disp_lock);         /* set TS_RUN state and lock */
1452        tp->t_disp_queue = dp;

1454        dq = &dp->disp_q[tpri];
1455        dp->disp_nrunnable++;
1456        if (!bound)
1457                dp->disp_steal = 0;
1458        membar_enter();

1460        if (dq->dq_sruncnt++ != 0) {
1461                ASSERT(dq->dq_last != NULL);
1462                tp->t_link = dq->dq_first;
1463                dq->dq_first = tp;
1464        } else {
1465                ASSERT(dq->dq_last == NULL);
1466                ASSERT(dq->dq_first == NULL);
1467                tp->t_link = NULL;
1468                dq->dq_first = dq->dq_last = tp;
1469                BT_SET(dp->disp_qactmap, tpri);
1470                if (tpri > dp->disp_maxrunpri) {
1471                        dp->disp_maxrunpri = tpri;
1472                        membar_enter();
1473                        cpu_resched(cp, tpri);
1474                }
1475        }

1477        if (!bound && tpri > dp->disp_max_unbound_pri) {
1478                if (tp == curthread && dp->disp_max_unbound_pri == -1 &&
1479                    cp == CPU) {
1480                        /*
1481                         * If there are no other unbound threads on the
1482                         * run queue, don't allow other CPUs to steal
1483                         * this thread while we are in the middle of a
1484                         * context switch. We may just switch to it
1485                         * again right away. CPU_DISP_DONTSTEAL is cleared
1486                         * in swtch and swtch_to.
1487                         */
1488                        cp->cpu_disp_flags |= CPU_DISP_DONTSTEAL;
1489                }
1490                dp->disp_max_unbound_pri = tpri;
1491        }
1492        (*disp_enq_thread)(cp, bound);
1493 }
```

unchanged_portion_omitted_

```
1573 /*
1574  * Remove a thread from the dispatcher queue if it is on it.
1575  * It is not an error if it is not found but we return whether
1576  * or not it was found in case the caller wants to check.
```

```
 */
int
dispdeq(kthread_t *tp)
{
        disp_t          *dp;
        dispq_t         *dq;
        kthread_t       *rp;
        kthread_t       *trp;
        kthread_t       **ptp;
        int             tpri;

        ASSERT(THREAD_LOCK_HELD(tp));

        if (tp->t_state != TS_RUN)
                return (0);

        /*
         * The thread is "swapped" or is on the swap queue and
         * hence no longer on the run queue, so return true.
         */
        if ((tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ)) != TS_LOAD)
                return (1);

        tpri = DISP_PRIO(tp);
        dp = tp->t_disp_queue;
        ASSERT(tpri < dp->disp_npri);
        dq = &dp->disp_q[tpri];
        ptp = &dq->dq_first;
        rp = *ptp;
        trp = NULL;

        ASSERT(dq->dq_last == NULL || dq->dq_last->t_link == NULL);

        /*
         * Search for thread in queue.
         * Double links would simplify this at the expense of disp/setrun.
         */
        while (rp != tp && rp != NULL) {
                trp = rp;
                ptp = &trp->t_link;
                rp = trp->t_link;
        }

        if (rp == NULL) {
                panic("dispdeq: thread not on queue");
        }

        DTRACE_SCHED2(dequeue, kthread_t *, tp, disp_t *, dp);

        /*
         * Found it so remove it from queue.
         */
        if ((*ptp = rp->t_link) == NULL)
                dq->dq_last = trp;

        dp->disp_nrunnable--;
        if (--dq->dq_sruncnt == 0) {
                dp->disp_qactmap[tpri >> BT_ULSHIFT] &= ~BT_BIW(tpri);
                if (dp->disp_nrunnable == 0) {
                        dp->disp_max_unbound_pri = -1;
                        dp->disp_maxrunpri = -1;
                } else if (tpri == dp->disp_maxrunpri) {
                        int ipri;

                        ipri = bt_gethighbit(dp->disp_qactmap,
                            dp->disp_maxrunpri >> BT_ULSHIFT);
```



```
                        if (ipri < dp->disp_max_unbound_pri)
                                dp->disp_max_unbound_pri = ipri;
                        dp->disp_maxrunpri = ipri;
                }
        }
        tp->t_link = NULL;
        THREAD_TRANSITION(tp);          /* put in intermediate state */
        return (1);
}

/*
 * dq_sruninc and dq_srundec are public functions for
 * incrementing/decrementing the sruncnts when a thread on
 * a dispatcher queue is made schedulable/unschedulable by
 * resetting the TS_LOAD flag.
 *
 * The caller MUST have the thread lock and therefore the dispatcher
 * queue lock so that the operation which changes
 * the flag, the operation that checks the status of the thread to
 * determine if it's on a disp queue AND the call to this function
 * are one atomic operation with respect to interrupts.
 */

/*
 * Called by sched AFTER TS_LOAD flag is set on a swapped, runnable thread.
 */
void
dq_sruninc(kthread_t *t)
{
        ASSERT(t->t_state == TS_RUN);
        ASSERT(t->t_schedflag & TS_LOAD);

        THREAD_TRANSITION(t);
        setfrontdq(t);
}

/*
 * See comment on calling conventions above.
 * Called by sched BEFORE TS_LOAD flag is cleared on a runnable thread.
 */
void
dq_srundec(kthread_t *t)
{
        ASSERT(t->t_schedflag & TS_LOAD);

        (void) dispdeq(t);
        disp_swapped_enq(t);
}

/*
 * Change the dispatcher lock of thread to the "swapped_lock"
 * and return with thread lock still held.
 *
 * Called with thread_lock held, in transition state, and at high spl.
 */
void
disp_swapped_enq(kthread_t *tp)
{
        ASSERT(THREAD_LOCK_HELD(tp));
        ASSERT(tp->t_schedflag & TS_LOAD);

        switch (tp->t_state) {
        case TS_RUN:
                disp_lock_enter_high(&swapped_lock);
                THREAD_SWAP(tp, &swapped_lock); /* set TS_RUN state and lock */
```

```
1736                 break;
1737         case TS_ONPROC:
1738                 disp_lock_enter_high(&swapped_lock);
1739                 THREAD_TRANSITION(tp);
1740                 wake_sched_sec = 1;             /* tell clock to wake sched */
1741                 THREAD_SWAP(tp, &swapped_lock); /* set TS_RUN state and lock */
1742                 break;
1743         default:
1744                 panic("disp_swapped: tp: %p bad t_state", (void *)tp);
1745         }
1746 }

1748 /*
1749  * This routine is called by setbackdq/setfrontdq if the thread is
1750  * not loaded or loaded and on the swap queue.
1751  *
1752  * Thread state TS_SLEEP implies that a swapped thread
1753  * has been woken up and needs to be swapped in by the swapper.
1754  *
1755  * Thread state TS_RUN, it implies that the priority of a swapped
1756  * thread is being increased by scheduling class (e.g. ts_update).
1757  */
1758 static void
1759 disp_swapped_setrun(kthread_t *tp)
1760 {
1761         ASSERT(THREAD_LOCK_HELD(tp));
1762         ASSERT((tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ)) != TS_LOAD);

1764         switch (tp->t_state) {
1765         case TS_SLEEP:
1766                 disp_lock_enter_high(&swapped_lock);
1767                 /*
1768                  * Wakeup sched immediately (i.e., next tick) if the
1769                  * thread priority is above maxclsyspri.
1770                  */
1771                 if (DISP_PRIO(tp) > maxclsyspri)
1772                         wake_sched = 1;
1773                 else
1774                         wake_sched_sec = 1;
1775                 THREAD_RUN(tp, &swapped_lock); /* set TS_RUN state and lock */
1776                 break;
1777         case TS_RUN:                            /* called from ts_update */
1778                 break;
1779         default:
1780                 panic("disp_swapped_setrun: tp: %p bad t_state", (void *)tp);
1781         }
1782 }

1646 /*
1647  *      Make a thread give up its processor.  Find the processor on
1648  *      which this thread is executing, and have that processor
1649  *      preempt.
1650  *
1651  *      We allow System Duty Cycle (SDC) threads to be preempted even if
1652  *      they are running at kernel priorities.  To implement this, we always
1653  *      set cpu_kprunrun; this ensures preempt() will be called.  Since SDC
1654  *      calls cpu_surrender() very often, we only preempt if there is anyone
1655  *      competing with us.
1656  */
1657 void
1658 cpu_surrender(kthread_t *tp)
1659 {
1660         cpu_t   *cpup;
1661         int     max_pri;
1662         int     max_run_pri;
1663         klwp_t  *lwp;
```



```
1665         ASSERT(THREAD_LOCK_HELD(tp));

1667         if (tp->t_state != TS_ONPROC)
1668                 return;
1669         cpup = tp->t_disp_queue->disp_cpu;      /* CPU thread dispatched to */
1670         max_pri = cpup->cpu_disp->disp_maxrunpri; /* best pri of that CPU */
1671         max_run_pri = CP_MAXRUNPRI(cpup->cpu_part);
1672         if (max_pri < max_run_pri)
1673                 max_pri = max_run_pri;

1675         if (tp->t_cid == sysdccid) {
1676                 uint_t t_pri = DISP_PRIO(tp);
1677                 if (t_pri > max_pri)
1678                         return;         /* we are not competing w/ anyone */
1679                 cpup->cpu_runrun = cpup->cpu_kprunrun = 1;
1680         } else {
1681                 cpup->cpu_runrun = 1;
1682                 if (max_pri >= kpreemptpri && cpup->cpu_kprunrun == 0) {
1683                         cpup->cpu_kprunrun = 1;
1684                 }
1685         }

1687         /*
1688          * Propagate cpu_runrun, and cpu_kprunrun to global visibility.
1689          */
1690         membar_enter();

1692         DTRACE_SCHED1(surrender, kthread_t *, tp);

1694         /*
1695          * Make the target thread take an excursion through trap()
1696          * to do preempt() (unless we're already in trap or post_syscall,
1697          * calling cpu_surrender via CL_TRAPRET).
1698          */
1699         if (tp != curthread || (lwp = tp->t_lwp) == NULL ||
1700             lwp->lwp_state != LWP_USER) {
1701                 aston(tp);
1702                 if (cpup != CPU)
1703                         poke_cpu(cpup->cpu_id);
1704         }
1705         TRACE_2(TR_FAC_DISP, TR_CPU_SURRENDER,
1706             "cpu_surrender:tid %p cpu %p", tp, cpup);
1707 }
_____unchanged_portion_omitted_

2004 /*
2005  * disp_adjust_unbound_pri() - thread is becoming unbound, so we should
2006  *      check if the CPU to which is was previously bound should have
2007  *      its disp_max_unbound_pri increased.
2008  */
2009 void
2010 disp_adjust_unbound_pri(kthread_t *tp)
2011 {
2012         disp_t *dp;
2013         pri_t tpri;

2015         ASSERT(THREAD_LOCK_HELD(tp));

2017         /*
2018          * Don't do anything if the thread is not bound, or
2019          * currently not runnable.
2157          * currently not runnable or swapped out.
2020          */
2021         if (tp->t_bound_cpu == NULL ||
2022             tp->t_state != TS_RUN)
```

```
2160             tp->t_state != TS_RUN ||
2161             tp->t_schedflag & TS_ON_SWAPQ)
2023                 return;

2025         tpri = DISP_PRIO(tp);
2026         dp = tp->t_bound_cpu->cpu_disp;
2027         ASSERT(tpri >= 0 && tpri < dp->disp_npri);
2028         if (tpri > dp->disp_max_unbound_pri)
2029                 dp->disp_max_unbound_pri = tpri;
2030 }

2032 /*
2033  * disp_getbest()
2034  *   De-queue the highest priority unbound runnable thread.
2035  *   Returns with the thread unlocked and onproc but at splhigh (like disp()).
2036  *   Returns NULL if nothing found.
2037  *   Returns T_DONTSTEAL if the thread was not stealable.
2038  *   so that the caller will try again later.
2039  *
2040  *   Passed a pointer to a dispatch queue not associated with this CPU, and
2041  *   its type.
2042  */
2043 static kthread_t *
2044 disp_getbest(disp_t *dp)
2045 {
2046         kthread_t       *tp;
2047         dispq_t         *dq;
2048         pri_t           pri;
2049         cpu_t           *cp, *tcp;
2050         boolean_t       allbound;

2052         disp_lock_enter(&dp->disp_lock);

2054         /*
2055          * If there is nothing to run, or the CPU is in the middle of a
2056          * context switch of the only thread, return NULL.
2057          */
2058         tcp = dp->disp_cpu;
2059         cp = CPU;
2060         pri = dp->disp_max_unbound_pri;
2061         if (pri == -1 ||
2062             (tcp != NULL && (tcp->cpu_disp_flags & CPU_DISP_DONTSTEAL) &&
2063             tcp->cpu_disp->disp_nrunnable == 1)) {
2064                 disp_lock_exit_nopreempt(&dp->disp_lock);
2065                 return (NULL);
2066         }

2068         dq = &dp->disp_q[pri];


2071         /*
2072          * Assume that all threads are bound on this queue, and change it
2073          * later when we find out that it is not the case.
2074          */
2075         allbound = B_TRUE;
2076         for (tp = dq->dq_first; tp != NULL; tp = tp->t_link) {
2077                 hrtime_t now, nosteal, rqtime;

2079                 /*
2080                  * Skip over bound threads which could be here even
2081                  * though disp_max_unbound_pri indicated this level.
2082                  */
2083                 if (tp->t_bound_cpu || tp->t_weakbound_cpu)
2084                         continue;

2086                 /*
```



```
2087                  * We've got some unbound threads on this queue, so turn
2088                  * the allbound flag off now.
2089                  */
2090                 allbound = B_FALSE;

2092                 /*
2093                  * The thread is a candidate for stealing from its run queue. We
2094                  * don't want to steal threads that became runnable just a
2095                  * moment ago. This improves CPU affinity for threads that get
2096                  * preempted for short periods of time and go back on the run
2097                  * queue.
2098                  *
2099                  * We want to let it stay on its run queue if it was only placed
2100                  * there recently and it was running on the same CPU before that
2101                  * to preserve its cache investment. For the thread to remain on
2102                  * its run queue, ALL of the following conditions must be
2103                  * satisfied:
2104                  *
2105                  * - the disp queue should not be the kernel preemption queue
2106                  * - delayed idle stealing should not be disabled
2107                  * - nosteal_nsec should be non-zero
2108                  * - it should run with user priority
2109                  * - it should be on the run queue of the CPU where it was
2110                  *   running before being placed on the run queue
2111                  * - it should be the only thread on the run queue (to prevent
2112                  *   extra scheduling latency for other threads)
2113                  * - it should sit on the run queue for less than per-chip
2114                  *   nosteal interval or global nosteal interval
2115                  * - in case of CPUs with shared cache it should sit in a run
2116                  *   queue of a CPU from a different chip
2117                  *
2118                  * The checks are arranged so that the ones that are faster are
2119                  * placed earlier.
2120                  */
2121                 if (tcp == NULL ||
2122                     pri >= minclsyspri ||
2123                     tp->t_cpu != tcp)
2124                         break;

2126                 /*
2127                  * Steal immediately if, due to CMT processor architecture
2128                  * migraiton between cp and tcp would incur no performance
2129                  * penalty.
2130                  */
2131                 if (pg_cmt_can_migrate(cp, tcp))
2132                         break;

2134                 nosteal = nosteal_nsec;
2135                 if (nosteal == 0)
2136                         break;

2138                 /*
2139                  * Calculate time spent sitting on run queue
2140                  */
2141                 now = gethrtime_unscaled();
2142                 rqtime = now - tp->t_waitrq;
2143                 scalehrtime(&rqtime);

2145                 /*
2146                  * Steal immediately if the time spent on this run queue is more
2147                  * than allowed nosteal delay.
2148                  *
2149                  * Negative rqtime check is needed here to avoid infinite
2150                  * stealing delays caused by unlikely but not impossible
2151                  * drifts between CPU times on different CPUs.
2152                  */
```

```
2153                 if (rqtime > nosteal || rqtime < 0)
2154                         break;

2156                 DTRACE_PROBE4(nosteal, kthread_t *, tp,
2157                     cpu_t *, tcp, cpu_t *, cp, hrtime_t, rqtime);
2158                 scalehrtime(&now);
2159                 /*
2160                  * Calculate when this thread becomes stealable
2161                  */
2162                 now += (nosteal - rqtime);

2164                 /*
2165                  * Calculate time when some thread becomes stealable
2166                  */
2167                 if (now < dp->disp_steal)
2168                         dp->disp_steal = now;
2169         }

2171         /*
2172          * If there were no unbound threads on this queue, find the queue
2173          * where they are and then return later. The value of
2174          * disp_max_unbound_pri is not always accurate because it isn't
2175          * reduced until another idle CPU looks for work.
2176          */
2177         if (allbound)
2178                 disp_fix_unbound_pri(dp, pri);

2180         /*
2181          * If we reached the end of the queue and found no unbound threads
2182          * then return NULL so that other CPUs will be considered.  If there
2183          * are unbound threads but they cannot yet be stolen, then
2184          * return T_DONTSTEAL and try again later.
2185          */
2186         if (tp == NULL) {
2187                 disp_lock_exit_nopreempt(&dp->disp_lock);
2188                 return (allbound ? NULL : T_DONTSTEAL);
2189         }

2191         /*
2192          * Found a runnable, unbound thread, so remove it from queue.
2193          * dispdeq() requires that we have the thread locked, and we do,
2194          * by virtue of holding the dispatch queue lock.  dispdeq() will
2195          * put the thread in transition state, thereby dropping the dispq
2196          * lock.
2197          */

2199 #ifdef DEBUG
2200         {
2201                 int     thread_was_on_queue;

2203                 thread_was_on_queue = dispdeq(tp);      /* drops disp_lock */
2204                 ASSERT(thread_was_on_queue);
2205         }

2207 #else /* DEBUG */
2208         (void) dispdeq(tp);                     /* drops disp_lock */
2209 #endif /* DEBUG */

2211         /*
2212          * Reset the disp_queue steal time - we do not know what is the smallest
2213          * value across the queue is.
2214          */
2215         dp->disp_steal = 0;

2356         tp->t_schedflag |= TS_DONT_SWAP;
```



```
2217         /*
2218          * Setup thread to run on the current CPU.
2219          */
2220         tp->t_disp_queue = cp->cpu_disp;

2222         cp->cpu_dispthread = tp;                /* protected by spl only */
2223         cp->cpu_dispatch_pri = pri;

2225         /*
2226          * There can be a memory synchronization race between disp_getbest()
2227          * and disp_ratify() vs cpu_resched() where cpu_resched() is trying
2228          * to preempt the current thread to run the enqueued thread while
2229          * disp_getbest() and disp_ratify() are changing the current thread
2230          * to the stolen thread. This may lead to a situation where
2231          * cpu_resched() tries to preempt the wrong thread and the
2232          * stolen thread continues to run on the CPU which has been tagged
2233          * for preemption.
2234          * Later the clock thread gets enqueued but doesn't get to run on the
2235          * CPU causing the system to hang.
2236          *
2237          * To avoid this, grabbing and dropping the disp_lock (which does
2238          * a memory barrier) is needed to synchronize the execution of
2239          * cpu_resched() with disp_getbest() and disp_ratify() and
2240          * synchronize the memory read and written by cpu_resched(),
2241          * disp_getbest(), and disp_ratify() with each other.
2242          *  (see CR#6482861 for more details).
2243          */
2244         disp_lock_enter_high(&cp->cpu_disp->disp_lock);
2245         disp_lock_exit_high(&cp->cpu_disp->disp_lock);

2247         ASSERT(pri == DISP_PRIO(tp));

2249         DTRACE_PROBE3(steal, kthread_t *, tp, cpu_t *, tcp, cpu_t *, cp);

2251         thread_onproc(tp, cp);                  /* set t_state to TS_ONPROC */

2253         /*
2254          * Return with spl high so that swtch() won't need to raise it.
2255          * The disp_lock was dropped by dispdeq().
2256          */

2258         return (tp);
2259 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   79977 Fri May  8 18:10:23 2015
new/usr/src/uts/common/disp/fss.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

 316 #define FSS_TICK_COST   1000    /* tick cost for threads with nice level = 0 */

 318 /*
 319  * Decay rate percentages are based on n/128 rather than n/100 so  that
 320  * calculations can avoid having to do an integer divide by 100 (divide
 321  * by FSS_DECAY_BASE == 128 optimizes to an arithmetic shift).
 322  *
 323  * FSS_DECAY_MIN        =  83/128 ~= 65%
 324  * FSS_DECAY_MAX        = 108/128 ~= 85%
 325  * FSS_DECAY_USG        =  96/128 ~= 75%
 326  */
 327 #define FSS_DECAY_MIN   83      /* fsspri decay pct for threads w/ nice -20 */
 328 #define FSS_DECAY_MAX   108     /* fsspri decay pct for threads w/ nice +19 */
 329 #define FSS_DECAY_USG   96      /* fssusage decay pct for projects */
 330 #define FSS_DECAY_BASE  128     /* base for decay percentages above */

 332 #define FSS_NICE_MIN    0
 333 #define FSS_NICE_MAX    (2 * NZERO - 1)
 334 #define FSS_NICE_RANGE  (FSS_NICE_MAX - FSS_NICE_MIN + 1)

 336 static int      fss_nice_tick[FSS_NICE_RANGE];
 337 static int      fss_nice_decay[FSS_NICE_RANGE];

 339 static pri_t    fss_maxupri = FSS_MAXUPRI; /* maximum FSS user priority */
 340 static pri_t    fss_maxumdpri; /* maximum user mode fss priority */
 341 static pri_t    fss_maxglobpri; /* maximum global priority used by fss class */
 342 static pri_t    fss_minglobpri; /* minimum global priority */

 344 static fssproc_t fss_listhead[FSS_LISTS];
 345 static kmutex_t fss_listlock[FSS_LISTS];

 347 static fsspset_t *fsspsets;
 348 static kmutex_t fsspsets_lock;  /* protects fsspsets */

 350 static id_t     fss_cid;

 352 static time_t   fss_minrun = 2; /* t_pri becomes 59 within 2 secs */
 353 static time_t   fss_minslp = 2; /* min time on sleep queue for hardswap */
 352 static int      fss_quantum = 11;

 354 static void     fss_newpri(fssproc_t *, boolean_t);
 355 static void     fss_update(void *);
 356 static int      fss_update_list(int);
 357 static void     fss_change_priority(kthread_t *, fssproc_t *);

 359 static int      fss_admin(caddr_t, cred_t *);
 360 static int      fss_getclinfo(void *);
 361 static int      fss_parmsin(void *);
 362 static int      fss_parmsout(void *, pc_vaparms_t *);
 363 static int      fss_vaparmsin(void *, pc_vaparms_t *);
 364 static int      fss_vaparmsout(void *, pc_vaparms_t *);
 365 static int      fss_getclpri(pcpri_t *);
 366 static int      fss_alloc(void **, int);
```



```
 367 static void     fss_free(void *);

 369 static int      fss_enterclass(kthread_t *, id_t, void *, cred_t *, void *);
 370 static void     fss_exitclass(void *);
 371 static int      fss_canexit(kthread_t *, cred_t *);
 372 static int      fss_fork(kthread_t *, kthread_t *, void *);
 373 static void     fss_forkret(kthread_t *, kthread_t *);
 374 static void     fss_parmsget(kthread_t *, void *);
 375 static int      fss_parmsset(kthread_t *, void *, id_t, cred_t *);
 376 static void     fss_stop(kthread_t *, int, int);
 377 static void     fss_exit(kthread_t *);
 378 static void     fss_active(kthread_t *);
 379 static void     fss_inactive(kthread_t *);
 382 static pri_t    fss_swapin(kthread_t *, int);
 383 static pri_t    fss_swapout(kthread_t *, int);
 380 static void     fss_trapret(kthread_t *);
 381 static void     fss_preempt(kthread_t *);
 382 static void     fss_setrun(kthread_t *);
 383 static void     fss_sleep(kthread_t *);
 384 static void     fss_tick(kthread_t *);
 385 static void     fss_wakeup(kthread_t *);
 386 static int      fss_donice(kthread_t *, cred_t *, int, int *);
 387 static int      fss_doprio(kthread_t *, cred_t *, int, int *);
 388 static pri_t    fss_globpri(kthread_t *);
 389 static void     fss_yield(kthread_t *);
 390 static void     fss_nullsys();

 392 static struct classfuncs fss_classfuncs = {
 393         /* class functions */
 394         fss_admin,
 395         fss_getclinfo,
 396         fss_parmsin,
 397         fss_parmsout,
 398         fss_vaparmsin,
 399         fss_vaparmsout,
 400         fss_getclpri,
 401         fss_alloc,
 402         fss_free,

 404         /* thread functions */
 405         fss_enterclass,
 406         fss_exitclass,
 407         fss_canexit,
 408         fss_fork,
 409         fss_forkret,
 410         fss_parmsget,
 411         fss_parmsset,
 412         fss_stop,
 413         fss_exit,
 414         fss_active,
 415         fss_inactive,
 420         fss_swapin,
 421         fss_swapout,
 416         fss_trapret,
 417         fss_preempt,
 418         fss_setrun,
 419         fss_sleep,
 420         fss_tick,
 421         fss_wakeup,
 422         fss_donice,
 423         fss_globpri,
 424         fss_nullsys,    /* set_process_group */
 425         fss_yield,
 426         fss_doprio,
 427 };
______unchanged_portion_omitted_
```

```
2136 /*
2143  * fss_swapin() returns -1 if the thread is loaded or is not eligible to be
2144  * swapped in. Otherwise, it returns the thread's effective priority based
2145  * on swapout time and size of process (0 <= epri <= 0 SHRT_MAX).
2146  */
2147 /*ARGSUSED*/
2148 static pri_t
2149 fss_swapin(kthread_t *t, int flags)
2150 {
2151         fssproc_t *fssproc = FSSPROC(t);
2152         long epri = -1;
2153         proc_t *pp = ttoproc(t);

2155         ASSERT(THREAD_LOCK_HELD(t));

2157         if (t->t_state == TS_RUN && (t->t_schedflag & TS_LOAD) == 0) {
2158                 time_t swapout_time;

2160                 swapout_time = (ddi_get_lbolt() - t->t_stime) / hz;
2161                 if (INHERITED(t) || (fssproc->fss_flags & FSSKPRI)) {
2162                         epri = (long)DISP_PRIO(t) + swapout_time;
2163                 } else {
2164                         /*
2165                          * Threads which have been out for a long time,
2166                          * have high user mode priority and are associated
2167                          * with a small address space are more deserving.
2168                          */
2169                         epri = fssproc->fss_umdpri;
2170                         ASSERT(epri >= 0 && epri <= fss_maxumdpri);
2171                         epri += swapout_time - pp->p_swrss / nz(maxpgio)/2;
2172                 }
2173                 /*
2174                  * Scale epri so that SHRT_MAX / 2 represents zero priority.
2175                  */
2176                 epri += SHRT_MAX / 2;
2177                 if (epri < 0)
2178                         epri = 0;
2179                 else if (epri > SHRT_MAX)
2180                         epri = SHRT_MAX;
2181         }
2182         return ((pri_t)epri);
2183 }

2185 /*
2186  * fss_swapout() returns -1 if the thread isn't loaded or is not eligible to
2187  * be swapped out. Otherwise, it returns the thread's effective priority
2188  * based on if the swapper is in softswap or hardswap mode.
2189  */
2190 static pri_t
2191 fss_swapout(kthread_t *t, int flags)
2192 {
2193         fssproc_t *fssproc = FSSPROC(t);
2194         long epri = -1;
2195         proc_t *pp = ttoproc(t);
2196         time_t swapin_time;

2198         ASSERT(THREAD_LOCK_HELD(t));

2200         if (INHERITED(t) ||
2201             (fssproc->fss_flags & FSSKPRI) ||
2202             (t->t_proc_flag & TP_LWPEXIT) ||
2203             (t->t_state & (TS_ZOMB|TS_FREE|TS_STOPPED|TS_ONPROC|TS_WAIT)) ||
2204             !(t->t_schedflag & TS_LOAD) ||
2205             !(SWAP_OK(t)))
2206                 return (-1);
```



```
2208         ASSERT(t->t_state & (TS_SLEEP | TS_RUN));

2210         swapin_time = (ddi_get_lbolt() - t->t_stime) / hz;

2212         if (flags == SOFTSWAP) {
2213                 if (t->t_state == TS_SLEEP && swapin_time > maxslp) {
2214                         epri = 0;
2215                 } else {
2216                         return ((pri_t)epri);
2217                 }
2218         } else {
2219                 pri_t pri;

2221                 if ((t->t_state == TS_SLEEP && swapin_time > fss_minslp) ||
2222                     (t->t_state == TS_RUN && swapin_time > fss_minrun)) {
2223                         pri = fss_maxumdpri;
2224                         epri = swapin_time -
2225                             (rm_asrss(pp->p_as) / nz(maxpgio)/2) - (long)pri;
2226                 } else {
2227                         return ((pri_t)epri);
2228                 }
2229         }

2231         /*
2232          * Scale epri so that SHRT_MAX / 2 represents zero priority.
2233          */
2234         epri += SHRT_MAX / 2;
2235         if (epri < 0)
2236                 epri = 0;
2237         else if (epri > SHRT_MAX)
2238                 epri = SHRT_MAX;

2240         return ((pri_t)epri);
2241 }

2243 /*
2137  * If thread is currently at a kernel mode priority (has slept) and is
2138  * returning to the userland we assign it the appropriate user mode priority
2139  * and time quantum here.  If we're lowering the thread's priority below that
2140  * of other runnable threads then we will set runrun via cpu_surrender() to
2141  * cause preemption.
2142  */
2143 static void
2144 fss_trapret(kthread_t *t)
2145 {
2146         fssproc_t *fssproc = FSSPROC(t);
2147         cpu_t *cp = CPU;

2149         ASSERT(THREAD_LOCK_HELD(t));
2150         ASSERT(t == curthread);
2151         ASSERT(cp->cpu_dispthread == t);
2152         ASSERT(t->t_state == TS_ONPROC);

2154         t->t_kpri_req = 0;
2155         if (fssproc->fss_flags & FSSKPRI) {
2156                 /*
2157                  * If thread has blocked in the kernel
2158                  */
2159                 THREAD_CHANGE_PRI(t, fssproc->fss_umdpri);
2160                 cp->cpu_dispatch_pri = DISP_PRIO(t);
2161                 ASSERT(t->t_pri >= 0 && t->t_pri <= fss_maxglobpri);
2162                 fssproc->fss_flags &= ~FSSKPRI;

2164                 if (DISP_MUST_SURRENDER(t))
2165                         cpu_surrender(t);
```

```
2166         }

2275         /*
2276          * Swapout lwp if the swapper is waiting for this thread to reach
2277          * a safe point.
2278          */
2279         if (t->t_schedflag & TS_SWAPENQ) {
2280                 thread_unlock(t);
2281                 swapout_lwp(ttolwp(t));
2282                 thread_lock(t);
2283         }
2167 }

2169 /*
2170  * Arrange for thread to be placed in appropriate location on dispatcher queue.
2171  * This is called with the current thread in TS_ONPROC and locked.
2172  */
2173 static void
2174 fss_preempt(kthread_t *t)
2175 {
2176         fssproc_t *fssproc = FSSPROC(t);
2177         klwp_t *lwp;
2178         uint_t flags;

2180         ASSERT(t == curthread);
2181         ASSERT(THREAD_LOCK_HELD(curthread));
2182         ASSERT(t->t_state == TS_ONPROC);

2184         /*
2185          * If preempted in the kernel, make sure the thread has a kernel
2186          * priority if needed.
2187          */
2188         lwp = curthread->t_lwp;
2189         if (!(fssproc->fss_flags & FSSKPRI) && lwp != NULL && t->t_kpri_req) {
2190                 fssproc->fss_flags |= FSSKPRI;
2191                 THREAD_CHANGE_PRI(t, minclsyspri);
2192                 ASSERT(t->t_pri >= 0 && t->t_pri <= fss_maxglobpri);
2193                 t->t_trapret = 1;       /* so that fss_trapret will run */
2194                 aston(t);
2195         }

2197         /*
2198          * This thread may be placed on wait queue by CPU Caps. In this case we
2199          * do not need to do anything until it is removed from the wait queue.
2200          * Do not enforce CPU caps on threads running at a kernel priority
2201          */
2202         if (CPUCAPS_ON()) {
2203                 (void) cpucaps_charge(t, &fssproc->fss_caps,
2204                     CPUCAPS_CHARGE_ENFORCE);

2206                 if (!(fssproc->fss_flags & FSSKPRI) && CPUCAPS_ENFORCE(t))
2207                         return;
2208         }

2210         /*
2328          * If preempted in user-land mark the thread as swappable because it
2329          * cannot be holding any kernel locks.
2330          */
2331         ASSERT(t->t_schedflag & TS_DONT_SWAP);
2332         if (lwp != NULL && lwp->lwp_state == LWP_USER)
2333                 t->t_schedflag &= ~TS_DONT_SWAP;

2335         /*
2211          * Check to see if we're doing "preemption control" here.  If
2212          * we are, and if the user has requested that this thread not
2213          * be preempted, and if preemptions haven't been put off for
```



```
2214          * too long, let the preemption happen here but try to make
2215          * sure the thread is rescheduled as soon as possible.  We do
2216          * this by putting it on the front of the highest priority run
2217          * queue in the FSS class.  If the preemption has been put off
2218          * for too long, clear the "nopreempt" bit and let the thread
2219          * be preempted.
2220          */
2221         if (t->t_schedctl && schedctl_get_nopreempt(t)) {
2222                 if (fssproc->fss_timeleft > -SC_MAX_TICKS) {
2223                         DTRACE_SCHED1(schedctl__nopreempt, kthread_t *, t);
2224                         if (!(fssproc->fss_flags & FSSKPRI)) {
2225                                 /*
2226                                  * If not already remembered, remember current
2227                                  * priority for restoration in fss_yield().
2228                                  */
2229                                 if (!(fssproc->fss_flags & FSSRESTORE)) {
2230                                         fssproc->fss_scpri = t->t_pri;
2231                                         fssproc->fss_flags |= FSSRESTORE;
2232                                 }
2233                                 THREAD_CHANGE_PRI(t, fss_maxumdpri);
2359                                 t->t_schedflag |= TS_DONT_SWAP;
2234                         }
2235                         schedctl_set_yield(t, 1);
2236                         setfrontdq(t);
2237                         return;
2238                 } else {
2239                         if (fssproc->fss_flags & FSSRESTORE) {
2240                                 THREAD_CHANGE_PRI(t, fssproc->fss_scpri);
2241                                 fssproc->fss_flags &= ~FSSRESTORE;
2242                         }
2243                         schedctl_set_nopreempt(t, 0);
2244                         DTRACE_SCHED1(schedctl__preempt, kthread_t *, t);
2245                         /*
2246                          * Fall through and be preempted below.
2247                          */
2248                 }
2249         }

2251         flags = fssproc->fss_flags & (FSSBACKQ | FSSKPRI);

2253         if (flags == FSSBACKQ) {
2254                 fssproc->fss_timeleft = fss_quantum;
2255                 fssproc->fss_flags &= ~FSSBACKQ;
2256                 setbackdq(t);
2257         } else if (flags == (FSSBACKQ | FSSKPRI)) {
2258                 fssproc->fss_flags &= ~FSSBACKQ;
2259                 setbackdq(t);
2260         } else {
2261                 setfrontdq(t);
2262         }
2263 }
______unchanged_portion_omitted_

2294 /*
2295  * Prepare thread for sleep. We reset the thread priority so it will run at the
2296  * kernel priority level when it wakes up.
2297  */
2298 static void
2299 fss_sleep(kthread_t *t)
2300 {
2301         fssproc_t *fssproc = FSSPROC(t);

2303         ASSERT(t == curthread);
2304         ASSERT(THREAD_LOCK_HELD(t));

2306         ASSERT(t->t_state == TS_ONPROC);
```

```
2308         /*
2309          * Account for time spent on CPU before going to sleep.
2310          */
2311         (void) CPUCAPS_CHARGE(t, &fssproc->fss_caps, CPUCAPS_CHARGE_ENFORCE);

2313         fss_inactive(t);

2315         /*
2316          * Assign a system priority to the thread and arrange for it to be
2317          * retained when the thread is next placed on the run queue (i.e.,
2318          * when it wakes up) instead of being given a new pri.  Also arrange
2319          * for trapret processing as the thread leaves the system call so it
2320          * will drop back to normal priority range.
2321          */
2322         if (t->t_kpri_req) {
2323                 THREAD_CHANGE_PRI(t, minclsyspri);
2324                 fssproc->fss_flags |= FSSKPRI;
2325                 t->t_trapret = 1;       /* so that fss_trapret will run */
2326                 aston(t);
2327         } else if (fssproc->fss_flags & FSSKPRI) {
2328                 /*
2329                  * The thread has done a THREAD_KPRI_REQUEST(), slept, then
2330                  * done THREAD_KPRI_RELEASE() (so no t_kpri_req is 0 again),
2331                  * then slept again all without finishing the current system
2332                  * call so trapret won't have cleared FSSKPRI
2333                  */
2334                 fssproc->fss_flags &= ~FSSKPRI;
2335                 THREAD_CHANGE_PRI(t, fssproc->fss_umdpri);
2336                 if (DISP_MUST_SURRENDER(curthread))
2337                         cpu_surrender(t);
2338         }
2465         t->t_stime = ddi_get_lbolt();   /* time stamp for the swapper */
2339 }

2341 /*
2342  * A tick interrupt has ocurrend on a running thread. Check to see if our
2343  * time slice has expired.
2470  * time slice has expired.  We must also clear the TS_DONT_SWAP flag in
2471  * t_schedflag if the thread is eligible to be swapped out.
2344  */
2345 static void
2346 fss_tick(kthread_t *t)
2347 {
2348         fssproc_t *fssproc;
2349         fssproj_t *fssproj;
2478         klwp_t *lwp;
2350         boolean_t call_cpu_surrender = B_FALSE;
2351         boolean_t cpucaps_enforce = B_FALSE;

2353         ASSERT(MUTEX_HELD(&(ttoproc(t))->p_lock));

2355         /*
2356          * It's safe to access fsspset and fssproj structures because we're
2357          * holding our p_lock here.
2358          */
2359         thread_lock(t);
2360         fssproc = FSSPROC(t);
2361         fssproj = FSSPROC2FSSPROJ(fssproc);
2362         if (fssproj != NULL) {
2363                 fsspset_t *fsspset = FSSPROJ2FSSPSET(fssproj);
2364                 disp_lock_enter_high(&fsspset->fssps_displock);
2365                 fssproj->fssp_ticks += fss_nice_tick[fssproc->fss_nice];
2366                 fssproj->fssp_tick_cnt++;
2367                 fssproc->fss_ticks++;
2368                 disp_lock_exit_high(&fsspset->fssps_displock);
```



```
2369         }

2371         /*
2372          * Keep track of thread's project CPU usage.  Note that projects
2373          * get charged even when threads are running in the kernel.
2374          * Do not surrender CPU if running in the SYS class.
2375          */
2376         if (CPUCAPS_ON()) {
2377                 cpucaps_enforce = cpucaps_charge(t,
2378                     &fssproc->fss_caps, CPUCAPS_CHARGE_ENFORCE) &&
2379                     !(fssproc->fss_flags & FSSKPRI);
2380         }

2382         /*
2383          * A thread's execution time for threads running in the SYS class
2384          * is not tracked.
2385          */
2386         if ((fssproc->fss_flags & FSSKPRI) == 0) {
2387                 /*
2388                  * If thread is not in kernel mode, decrement its fss_timeleft
2389                  */
2390                 if (--fssproc->fss_timeleft <= 0) {
2391                         pri_t new_pri;

2393                         /*
2394                          * If we're doing preemption control and trying to
2395                          * avoid preempting this thread, just note that the
2396                          * thread should yield soon and let it keep running
2397                          * (unless it's been a while).
2398                          */
2399                         if (t->t_schedctl && schedctl_get_nopreempt(t)) {
2400                                 if (fssproc->fss_timeleft > -SC_MAX_TICKS) {
2401                                         DTRACE_SCHED1(schedctl__nopreempt,
2402                                             kthread_t *, t);
2403                                         schedctl_set_yield(t, 1);
2404                                         thread_unlock_nopreempt(t);
2405                                         return;
2406                                 }
2407                         }
2408                         fssproc->fss_flags &= ~FSSRESTORE;

2410                         fss_newpri(fssproc, B_TRUE);
2411                         new_pri = fssproc->fss_umdpri;
2412                         ASSERT(new_pri >= 0 && new_pri <= fss_maxglobpri);

2414                         /*
2415                          * When the priority of a thread is changed, it may
2416                          * be necessary to adjust its position on a sleep queue
2417                          * or dispatch queue. The function thread_change_pri
2418                          * accomplishes this.
2419                          */
2420                         if (thread_change_pri(t, new_pri, 0)) {
2550                                 if ((t->t_schedflag & TS_LOAD) &&
2551                                     (lwp = t->t_lwp) &&
2552                                     lwp->lwp_state == LWP_USER)
2553                                         t->t_schedflag &= ~TS_DONT_SWAP;
2421                                 fssproc->fss_timeleft = fss_quantum;
2422                         } else {
2423                                 call_cpu_surrender = B_TRUE;
2424                         }
2425                 } else if (t->t_state == TS_ONPROC &&
2426                     t->t_pri < t->t_disp_queue->disp_maxrunpri) {
2427                         /*
2428                          * If there is a higher-priority thread which is
2429                          * waiting for a processor, then thread surrenders
2430                          * the processor.
```

```
2431                         */
2432                        call_cpu_surrender = B_TRUE;
2433                }
2434        }

2436        if (cpucaps_enforce && 2 * fssproc->fss_timeleft > fss_quantum) {
2437                /*
2438                 * The thread used more than half of its quantum, so assume that
2439                 * it used the whole quantum.
2440                 *
2441                 * Update thread's priority just before putting it on the wait
2442                 * queue so that it gets charged for the CPU time from its
2443                 * quantum even before that quantum expires.
2444                 */
2445                fss_newpri(fssproc, B_FALSE);
2446                if (t->t_pri != fssproc->fss_umdpri)
2447                        fss_change_priority(t, fssproc);

2449                /*
2450                 * We need to call cpu_surrender for this thread due to cpucaps
2451                 * enforcement, but fss_change_priority may have already done
2452                 * so. In this case FSSBACKQ is set and there is no need to call
2453                 * cpu-surrender again.
2454                 */
2455                if (!(fssproc->fss_flags & FSSBACKQ))
2456                        call_cpu_surrender = B_TRUE;
2457        }

2459        if (call_cpu_surrender) {
2460                fssproc->fss_flags |= FSSBACKQ;
2461                cpu_surrender(t);
2462        }

2464        thread_unlock_nopreempt(t);     /* clock thread can't be preempted */
2465 }

2467 /*
2468  * Processes waking up go to the back of their queue.  We don't need to assign
2469  * a time quantum here because thread is still at a kernel mode priority and
2470  * the time slicing is not done for threads running in the kernel after
2471  * sleeping.  The proper time quantum will be assigned by fss_trapret before the
2472  * thread returns to user mode.
2473  */
2474 static void
2475 fss_wakeup(kthread_t *t)
2476 {
2477        fssproc_t *fssproc;

2479        ASSERT(THREAD_LOCK_HELD(t));
2480        ASSERT(t->t_state == TS_SLEEP);

2482        fss_active(t);

2617        t->t_stime = ddi_get_lbolt();           /* time stamp for the swapper */
2484        fssproc = FSSPROC(t);
2485        fssproc->fss_flags &= ~FSSBACKQ;

2487        if (fssproc->fss_flags & FSSKPRI) {
2488                /*
2489                 * If we already have a kernel priority assigned, then we
2490                 * just use it.
2491                 */
2492                setbackdq(t);
2493        } else if (t->t_kpri_req) {
2494                /*
2495                 * Give thread a priority boost if we were asked.
```



```
2496                 */
2497                fssproc->fss_flags |= FSSKPRI;
2498                THREAD_CHANGE_PRI(t, minclsyspri);
2499                setbackdq(t);
2500                t->t_trapret = 1;       /* so that fss_trapret will run */
2501                aston(t);
2502        } else {
2503                /*
2504                 * Otherwise, we recalculate the priority.
2505                 */
2506                if (t->t_disp_time == ddi_get_lbolt()) {
2507                        setfrontdq(t);
2508                } else {
2509                        fssproc->fss_timeleft = fss_quantum;
2510                        THREAD_CHANGE_PRI(t, fssproc->fss_umdpri);
2511                        setbackdq(t);
2512                }
2513        }
2514 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   42975 Fri May  8 18:10:23 2015
new/usr/src/uts/common/disp/fx.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

 144 #define FX_ISVALID(pri, quantum) \
 145         (((pri >= 0) || (pri == FX_CB_NOCHANGE)) &&                     \
 146             ((quantum >= 0) || (quantum == FX_NOCHANGE) ||              \
 147                 (quantum == FX_TQDEF) || (quantum == FX_TQINF)))


 150 static id_t     fx_cid;         /* fixed priority class ID */
 151 static fxdpent_t *fx_dptbl;     /* fixed priority disp parameter table */

 153 static pri_t    fx_maxupri = FXMAXUPRI;
 154 static pri_t    fx_maxumdpri;   /* max user mode fixed priority */

 156 static pri_t    fx_maxglobpri;  /* maximum global priority used by fx class */
 157 static kmutex_t fx_dptblock;    /* protects fixed priority dispatch table */


 160 static kmutex_t fx_cb_list_lock[FX_CB_LISTS];   /* protects list of fxprocs */
 161                                                 /* that have callbacks */
 162 static fxproc_t fx_cb_plisthead[FX_CB_LISTS];   /* dummy fxproc at head of */
 163                                                 /* list of fxprocs with */
 164                                                 /* callbacks */

 166 static int      fx_admin(caddr_t, cred_t *);
 167 static int      fx_getclinfo(void *);
 168 static int      fx_parmsin(void *);
 169 static int      fx_parmsout(void *, pc_vaparms_t *);
 170 static int      fx_vaparmsin(void *, pc_vaparms_t *);
 171 static int      fx_vaparmsout(void *, pc_vaparms_t *);
 172 static int      fx_getclpri(pcpri_t *);
 173 static int      fx_alloc(void **, int);
 174 static void     fx_free(void *);
 175 static int      fx_enterclass(kthread_t *, id_t, void *, cred_t *, void *);
 176 static void     fx_exitclass(void *);
 177 static int      fx_canexit(kthread_t *, cred_t *);
 178 static int      fx_fork(kthread_t *, kthread_t *, void *);
 179 static void     fx_forkret(kthread_t *, kthread_t *);
 180 static void     fx_parmsget(kthread_t *, void *);
 181 static int      fx_parmsset(kthread_t *, void *, id_t, cred_t *);
 182 static void     fx_stop(kthread_t *, int, int);
 183 static void     fx_exit(kthread_t *);
 184 static pri_t    fx_swapin(kthread_t *, int);
 185 static pri_t    fx_swapout(kthread_t *, int);
 184 static void     fx_trapret(kthread_t *);
 185 static void     fx_preempt(kthread_t *);
 186 static void     fx_setrun(kthread_t *);
 187 static void     fx_sleep(kthread_t *);
 188 static void     fx_tick(kthread_t *);
 189 static void     fx_wakeup(kthread_t *);
 190 static int      fx_donice(kthread_t *, cred_t *, int, int *);
 191 static int      fx_doprio(kthread_t *, cred_t *, int, int *);
 192 static pri_t    fx_globpri(kthread_t *);
 193 static void     fx_yield(kthread_t *);
 194 static void     fx_nullsys();
```



```
 196 extern fxdpent_t *fx_getdptbl(void);

 198 static void     fx_change_priority(kthread_t *, fxproc_t *);
 199 static fxproc_t *fx_list_lookup(kt_did_t);
 200 static void fx_list_release(fxproc_t *);


 203 static struct classfuncs fx_classfuncs = {
 204         /* class functions */
 205         fx_admin,
 206         fx_getclinfo,
 207         fx_parmsin,
 208         fx_parmsout,
 209         fx_vaparmsin,
 210         fx_vaparmsout,
 211         fx_getclpri,
 212         fx_alloc,
 213         fx_free,

 215         /* thread functions */
 216         fx_enterclass,
 217         fx_exitclass,
 218         fx_canexit,
 219         fx_fork,
 220         fx_forkret,
 221         fx_parmsget,
 222         fx_parmsset,
 223         fx_stop,
 224         fx_exit,
 225         fx_nullsys,     /* active */
 226         fx_nullsys,     /* inactive */
 229         fx_swapin,
 230         fx_swapout,
 227         fx_trapret,
 228         fx_preempt,
 229         fx_setrun,
 230         fx_sleep,
 231         fx_tick,
 232         fx_wakeup,
 233         fx_donice,
 234         fx_globpri,
 235         fx_nullsys,     /* set_process_group */
 236         fx_yield,
 237         fx_doprio,
 238 };
______unchanged_portion_omitted_


1203 /*
1204  * Prepare thread for sleep. We reset the thread priority so it will
1205  * run at the kernel priority level when it wakes up.
1206  */
1207 static void
1208 fx_sleep(kthread_t *t)
1209 {
1210         fxproc_t        *fxpp = (fxproc_t *)(t->t_cldata);

1212         ASSERT(t == curthread);
1213         ASSERT(THREAD_LOCK_HELD(t));

1215         /*
1216          * Account for time spent on CPU before going to sleep.
1217          */
1218         (void) CPUCAPS_CHARGE(t, &fxpp->fx_caps, CPUCAPS_CHARGE_ENFORCE);
```

```
1220         if (FX_HAS_CB(fxpp)) {
1221                 FX_CB_SLEEP(FX_CALLB(fxpp), fxpp->fx_cookie);
1222         }
1227         t->t_stime = ddi_get_lbolt();          /* time stamp for the swapper */
1228 }


1231 /*
1232  * Return Values:
1233  *
1234  *      -1 if the thread is loaded or is not eligible to be swapped in.
1235  *
1236  * FX and RT threads are designed so that they don't swapout; however,
1237  * it is possible that while the thread is swapped out and in another class, it
1238  * can be changed to FX or RT.  Since these threads should be swapped in
1239  * as soon as they're runnable, rt_swapin returns SHRT_MAX, and fx_swapin
1240  * returns SHRT_MAX - 1, so that it gives deference to any swapped out
1241  * RT threads.
1242  */
1243 /* ARGSUSED */
1244 static pri_t
1245 fx_swapin(kthread_t *t, int flags)
1246 {
1247         pri_t   tpri = -1;

1249         ASSERT(THREAD_LOCK_HELD(t));

1251         if (t->t_state == TS_RUN && (t->t_schedflag & TS_LOAD) == 0) {
1252                 tpri = (pri_t)SHRT_MAX - 1;
1253         }

1255         return (tpri);
1256 }

1258 /*
1259  * Return Values
1260  *      -1 if the thread isn't loaded or is not eligible to be swapped out.
1261  */
1262 /* ARGSUSED */
1263 static pri_t
1264 fx_swapout(kthread_t *t, int flags)
1265 {
1266         ASSERT(THREAD_LOCK_HELD(t));

1268         return (-1);

1223 }
```

_____unchanged_portion_omitted_

```
1342 /*
1343  * Processes waking up go to the back of their queue.
1344  */
1345 static void
1346 fx_wakeup(kthread_t *t)
1347 {
1348         fxproc_t        *fxpp = (fxproc_t *)(t->t_cldata);

1350         ASSERT(THREAD_LOCK_HELD(t));

1399         t->t_stime = ddi_get_lbolt();          /* time stamp for the swapper */
1352         if (FX_HAS_CB(fxpp)) {
1353                 clock_t new_quantum =  (clock_t)fxpp->fx_pquantum;
1354                 pri_t   newpri = fxpp->fx_pri;
1355                 FX_CB_WAKEUP(FX_CALLB(fxpp), fxpp->fx_cookie,
1356                     &new_quantum, &newpri);
```



```
1357                 FX_ADJUST_QUANTUM(new_quantum);
1358                 if ((int)new_quantum != fxpp->fx_pquantum) {
1359                         fxpp->fx_pquantum = (int)new_quantum;
1360                         fxpp->fx_timeleft = fxpp->fx_pquantum;
1361                 }

1363                 FX_ADJUST_PRI(newpri);
1364                 if (newpri != fxpp->fx_pri) {
1365                         fxpp->fx_pri = newpri;
1366                         THREAD_CHANGE_PRI(t, fx_dptbl[fxpp->fx_pri].fx_globpri);
1367                 }
1368         }

1370         fxpp->fx_flags &= ~FXBACKQ;

1372         if (t->t_disp_time != ddi_get_lbolt())
1373                 setbackdq(t);
1374         else
1375                 setfrontdq(t);
1376 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   25930 Fri May  8 18:10:23 2015
new/usr/src/uts/common/disp/rt.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_


  94 /*
  95  * Class specific code for the real-time class
  96  */

  98 /*
  99  * Extern declarations for variables defined in the rt master file
 100  */
 101 #define RTMAXPRI 59

 103 pri_t rt_maxpri = RTMAXPRI;     /* maximum real-time priority */
 104 rtdpent_t *rt_dptbl;      /* real-time dispatcher parameter table */

 106 /*
 107  * control flags (kparms->rt_cflags).
 108  */
 109 #define RT_DOPRI        0x01    /* change priority */
 110 #define RT_DOTQ         0x02    /* change RT time quantum */
 111 #define RT_DOSIG        0x04    /* change RT time quantum signal */

 113 static int      rt_admin(caddr_t, cred_t *);
 114 static int      rt_enterclass(kthread_t *, id_t, void *, cred_t *, void *);
 115 static int      rt_fork(kthread_t *, kthread_t *, void *);
 116 static int      rt_getclinfo(void *);
 117 static int      rt_getclpri(pcpri_t *);
 118 static int      rt_parmsin(void *);
 119 static int      rt_parmsout(void *, pc_vaparms_t *);
 120 static int      rt_vaparmsin(void *, pc_vaparms_t *);
 121 static int      rt_vaparmsout(void *, pc_vaparms_t *);
 122 static int      rt_parmsset(kthread_t *, void *, id_t, cred_t *);
 123 static int      rt_donice(kthread_t *, cred_t *, int, int *);
 124 static int      rt_doprio(kthread_t *, cred_t *, int, int *);
 125 static void     rt_exitclass(void *);
 126 static int      rt_canexit(kthread_t *, cred_t *);
 127 static void     rt_forkret(kthread_t *, kthread_t *);
 128 static void     rt_nullsys();
 129 static void     rt_parmsget(kthread_t *, void *);
 130 static void     rt_preempt(kthread_t *);
 131 static void     rt_setrun(kthread_t *);
 132 static void     rt_tick(kthread_t *);
 133 static void     rt_wakeup(kthread_t *);
 134 static pri_t    rt_swapin(kthread_t *, int);
 135 static pri_t    rt_swapout(kthread_t *, int);
 134 static pri_t    rt_globpri(kthread_t *);
 135 static void     rt_yield(kthread_t *);
 136 static int      rt_alloc(void **, int);
 137 static void     rt_free(void *);

 139 static void     rt_change_priority(kthread_t *, rtproc_t *);

 141 static id_t     rt_cid;         /* real-time class ID */
 142 static rtproc_t rt_plisthead;   /* dummy rtproc at head of rtproc list */
 143 static kmutex_t rt_dptblock;    /* protects realtime dispatch table */
```



```
 144 static kmutex_t rt_list_lock;   /* protects RT thread list */

 146 extern rtdpent_t *rt_getdptbl(void);

 148 static struct classfuncs rt_classfuncs = {
 149         /* class ops */
 150         rt_admin,
 151         rt_getclinfo,
 152         rt_parmsin,
 153         rt_parmsout,
 154         rt_vaparmsin,
 155         rt_vaparmsout,
 156         rt_getclpri,
 157         rt_alloc,
 158         rt_free,
 159         /* thread ops */
 160         rt_enterclass,
 161         rt_exitclass,
 162         rt_canexit,
 163         rt_fork,
 164         rt_forkret,
 165         rt_parmsget,
 166         rt_parmsset,
 167         rt_nullsys,     /* stop */
 168         rt_nullsys,     /* exit */
 169         rt_nullsys,     /* active */
 170         rt_nullsys,     /* inactive */
 173         rt_swapin,
 174         rt_swapout,
 171         rt_nullsys,     /* trapret */
 172         rt_preempt,
 173         rt_setrun,
 174         rt_nullsys,     /* sleep */
 175         rt_tick,
 176         rt_wakeup,
 177         rt_donice,
 178         rt_globpri,
 179         rt_nullsys,     /* set_process_group */
 180         rt_yield,
 181         rt_doprio,
 182 };
______unchanged_portion_omitted_


 892 /*
 893  * Arrange for thread to be placed in appropriate location
 894  * on dispatcher queue.  Runs at splhi() since the clock
 895  * interrupt can cause RTBACKQ to be set.
 896  */
 897 static void
 898 rt_preempt(kthread_t *t)
 899 {
 900         rtproc_t *rtpp = (rtproc_t *)(t->t_cldata);
 905         klwp_t *lwp;

 902         ASSERT(THREAD_LOCK_HELD(t));

 909         /*
 910          * If the state is user I allow swapping because I know I won't
 911          * be holding any locks.
 912          */
 913         if ((lwp = curthread->t_lwp) != NULL && lwp->lwp_state == LWP_USER)
 914                 t->t_schedflag &= ~TS_DONT_SWAP;
 904         if ((rtpp->rt_flags & RTBACKQ) != 0) {
 905                 rtpp->rt_timeleft = rtpp->rt_pquantum;
 906                 rtpp->rt_flags &= ~RTBACKQ;
```

```
 907                 setbackdq(t);
 908         } else
 909                 setfrontdq(t);

 911 }
______unchanged_portion_omitted_

 923 static void
 924 rt_setrun(kthread_t *t)
 925 {
 926         rtproc_t *rtpp = (rtproc_t *)(t->t_cldata);

 928         ASSERT(THREAD_LOCK_HELD(t));

 930         rtpp->rt_timeleft = rtpp->rt_pquantum;
 931         rtpp->rt_flags &= ~RTBACKQ;
 932         setbackdq(t);
 944 }

 946 /*
 947  * Returns the priority of the thread, -1 if the thread is loaded or ineligible
 948  * for swapin.
 949  *
 950  * FX and RT threads are designed so that they don't swapout; however, it
 951  * is possible that while the thread is swapped out and in another class, it
 952  * can be changed to FX or RT.  Since these threads should be swapped in as
 953  * soon as they're runnable, rt_swapin returns SHRT_MAX, and fx_swapin
 954  * returns SHRT_MAX - 1, so that it gives deference to any swapped out RT
 955  * threads.
 956  */
 957 /* ARGSUSED */
 958 static pri_t
 959 rt_swapin(kthread_t *t, int flags)
 960 {
 961         pri_t   tpri = -1;

 963         ASSERT(THREAD_LOCK_HELD(t));

 965         if (t->t_state == TS_RUN && (t->t_schedflag & TS_LOAD) == 0) {
 966                 tpri = (pri_t)SHRT_MAX;
 967         }

 969         return (tpri);
 970 }

 972 /*
 973  * Return an effective priority for swapout.
 974  */
 975 /* ARGSUSED */
 976 static pri_t
 977 rt_swapout(kthread_t *t, int flags)
 978 {
 979         ASSERT(THREAD_LOCK_HELD(t));

 981         return (-1);
 933 }
______unchanged_portion_omitted_
```

```
**********************************************************
    4804 Fri May  8 18:10:23 2015
new/usr/src/uts/common/disp/sysclass.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
```

```
/*
 * 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.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

#pragma ident   "%Z%%M% %I%     %E% SMI"        /* from SVr4.0 1.12 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/signal.h>
#include <sys/pcb.h>
#include <sys/user.h>
#include <sys/systm.h>
#include <sys/sysinfo.h>
#include <sys/var.h>
#include <sys/errno.h>
#include <sys/cmn_err.h>
#include <sys/proc.h>
#include <sys/debug.h>
#include <sys/inline.h>
#include <sys/disp.h>
#include <sys/class.h>
#include <sys/kmem.h>
#include <sys/cpuvar.h>
#include <sys/priocntl.h>

/*
 * Class specific code for the sys class. There are no
 * class specific data structures associated with
 * the sys class and the scheduling policy is trivially
```



```
 * simple. There is no time slicing.
 */

pri_t           sys_init(id_t, int, classfuncs_t **);
static int      sys_getclpri(pcpri_t *);
static int      sys_fork(kthread_t *, kthread_t *, void *);
static int      sys_enterclass(kthread_t *, id_t, void *, cred_t *, void *);
static int      sys_canexit(kthread_t *, cred_t *);
static int      sys_nosys();
static int      sys_donice(kthread_t *, cred_t *, int, int *);
static int      sys_doprio(kthread_t *, cred_t *, int, int *);
static void     sys_forkret(kthread_t *, kthread_t *);
static void     sys_nullsys();
static pri_t    sys_swappri(kthread_t *, int);
static int      sys_alloc(void **, int);

struct classfuncs sys_classfuncs = {
        /* messages to class manager */
        {
                sys_nosys,      /* admin */
                sys_nosys,      /* getclinfo */
                sys_nosys,      /* parmsin */
                sys_nosys,      /* parmsout */
                sys_nosys,      /* vaparmsin */
                sys_nosys,      /* vaparmsout */
                sys_getclpri,   /* getclpri */
                sys_alloc,
                sys_nullsys,    /* free */
        },
        /* operations on threads */
        {
                sys_enterclass, /* enterclass */
                sys_nullsys,    /* exitclass */
                sys_canexit,
                sys_fork,
                sys_forkret,    /* forkret */
                sys_nullsys,    /* parmsget */
                sys_nosys,      /* parmsset */
                sys_nullsys,    /* stop */
                sys_nullsys,    /* exit */
                sys_nullsys,    /* active */
                sys_nullsys,    /* inactive */
                sys_swappri,    /* swapin */
                sys_swappri,    /* swapout */
                sys_nullsys,    /* trapret */
                setfrontdq,     /* preempt */
                setbackdq,      /* setrun */
                sys_nullsys,    /* sleep */
                sys_nullsys,    /* tick */
                setbackdq,      /* wakeup */
                sys_donice,
                (pri_t (*)())sys_nosys, /* globpri */
                sys_nullsys,    /* set_process_group */
                sys_nullsys,    /* yield */
                sys_doprio,
        }

};
______unchanged_portion_omitted_


/* ARGSUSED */
static void
sys_forkret(t, ct)
        kthread_t *t;
        kthread_t *ct;
```

```
171 {
172         register proc_t *pp = ttoproc(t);
173         register proc_t *cp = ttoproc(ct);

175         ASSERT(t == curthread);
176         ASSERT(MUTEX_HELD(&pidlock));

178         /*
179          * Grab the child's p_lock before dropping pidlock to ensure
180          * the process does not disappear before we set it running.
181          */
182         mutex_enter(&cp->p_lock);
183         mutex_exit(&pidlock);
184         continuelwps(cp);
185         mutex_exit(&cp->p_lock);

187         mutex_enter(&pp->p_lock);
188         continuelwps(pp);
189         mutex_exit(&pp->p_lock);
195 }

197 /* ARGSUSED */
198 static pri_t
199 sys_swappri(t, flags)
200         kthread_t       *t;
201         int             flags;
202 {
203         return (-1);
190 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   37694 Fri May  8 18:10:24 2015
new/usr/src/uts/common/disp/sysdc.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

1113 /*ARGSUSED*/
1114 static pri_t
1115 sysdc_no_swap(kthread_t *t, int flags)
1116 {
1117         /* SDC threads cannot be swapped. */
1118         return (-1);
1119 }

1113 /*
1114  * Get maximum and minimum priorities enjoyed by SDC threads.
1115  */
1116 static int
1117 sysdc_getclpri(pcpri_t *pcprip)
1118 {
1119         pcprip->pc_clpmax = sysdc_maxpri;
1120         pcprip->pc_clpmin = sysdc_minpri;
1121         return (0);
1122 }
______unchanged_portion_omitted_

1167 static int sysdc_enosys();      /* Boy, ANSI-C's K&R compatibility is weird. */
1168 static int sysdc_einval();
1169 static void sysdc_nullsys();

1171 static struct classfuncs sysdc_classfuncs = {
1172         /* messages to class manager */
1173         {
1174                 sysdc_enosys,   /* admin */
1175                 sysdc_getclinfo,
1176                 sysdc_enosys,   /* parmsin */
1177                 sysdc_enosys,   /* parmsout */
1178                 sysdc_enosys,   /* vaparmsin */
1179                 sysdc_enosys,   /* vaparmsout */
1180                 sysdc_getclpri,
1181                 sysdc_alloc,
1182                 sysdc_free,
1183         },
1184         /* operations on threads */
1185         {
1186                 sysdc_enterclass,
1187                 sysdc_exitclass,
1188                 sysdc_canexit,
1189                 sysdc_fork,
1190                 sysdc_forkret,
1191                 sysdc_nullsys,  /* parmsget */
1192                 sysdc_enosys,   /* parmsset */
1193                 sysdc_nullsys,  /* stop */
1194                 sysdc_exit,
1195                 sysdc_nullsys,  /* active */
1196                 sysdc_nullsys,  /* inactive */
1205                 sysdc_no_swap,  /* swapin */
1206                 sysdc_no_swap,  /* swapout */
1197                 sysdc_nullsys,  /* trapret */
```



```
1198                 sysdc_preempt,
1199                 sysdc_setrun,
1200                 sysdc_sleep,
1201                 sysdc_tick,
1202                 sysdc_wakeup,
1203                 sysdc_einval,   /* donice */
1204                 sysdc_globpri,
1205                 sysdc_nullsys,  /* set_process_group */
1206                 sysdc_nullsys,  /* yield */
1207                 sysdc_einval,   /* doprio */
1208         }
1209 };
______unchanged_portion_omitted_
```

```
**********************************************************
   53361 Fri May  8 18:10:24 2015
new/usr/src/uts/common/disp/thread.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

 314 /*
 315  * Create a thread.
 316  *
 317  * thread_create() blocks for memory if necessary.  It never fails.
 318  *
 319  * If stk is NULL, the thread is created at the base of the stack
 320  * and cannot be swapped.
 321  */
 322 kthread_t *
 323 thread_create(
 324         caddr_t stk,
 325         size_t  stksize,
 326         void    (*proc)(),
 327         void    *arg,
 328         size_t  len,
 329         proc_t   *pp,
 330         int     state,
 331         pri_t   pri)
 332 {
 333         kthread_t *t;
 334         extern struct classfuncs sys_classfuncs;
 335         turnstile_t *ts;

 337         /*
 338          * Every thread keeps a turnstile around in case it needs to block.
 339          * The only reason the turnstile is not simply part of the thread
 340          * structure is that we may have to break the association whenever
 341          * more than one thread blocks on a given synchronization object.
 342          * From a memory-management standpoint, turnstiles are like the
 343          * "attached mblks" that hang off dblks in the streams allocator.
 344          */
 345         ts = kmem_cache_alloc(turnstile_cache, KM_SLEEP);

 347         if (stk == NULL) {
 348                 /*
 349                  * alloc both thread and stack in segkp chunk
 350                  */

 352                 if (stksize < default_stksize)
 353                         stksize = default_stksize;

 355                 if (stksize == default_stksize) {
 356                         stk = (caddr_t)segkp_cache_get(segkp_thread);
 357                 } else {
 358                         stksize = roundup(stksize, PAGESIZE);
 359                         stk = (caddr_t)segkp_get(segkp, stksize,
 360                             (KPD_HASREDZONE | KPD_NO_ANON | KPD_LOCKED));
 361                 }

 363                 ASSERT(stk != NULL);

 365                 /*
 366                  * The machine-dependent mutex code may require that
```



```
 367                  * thread pointers (since they may be used for mutex owner
 368                  * fields) have certain alignment requirements.
 369                  * PTR24_ALIGN is the size of the alignment quanta.
 370                  * XXX - assumes stack grows toward low addresses.
 371                  */
 372                 if (stksize <= sizeof (kthread_t) + PTR24_ALIGN)
 373                         cmn_err(CE_PANIC, "thread_create: proposed stack size"
 374                             " too small to hold thread.");
 375 #ifdef STACK_GROWTH_DOWN
 376                 stksize -= SA(sizeof (kthread_t) + PTR24_ALIGN - 1);
 377                 stksize &= -PTR24_ALIGN;        /* make thread aligned */
 378                 t = (kthread_t *)(stk + stksize);
 379                 bzero(t, sizeof (kthread_t));
 380                 if (audit_active)
 381                         audit_thread_create(t);
 382                 t->t_stk = stk + stksize;
 383                 t->t_stkbase = stk;
 384 #else   /* stack grows to larger addresses */
 385                 stksize -= SA(sizeof (kthread_t));
 386                 t = (kthread_t *)(stk);
 387                 bzero(t, sizeof (kthread_t));
 388                 t->t_stk = stk + sizeof (kthread_t);
 389                 t->t_stkbase = stk + stksize + sizeof (kthread_t);
 390 #endif  /* STACK_GROWTH_DOWN */
 391                 t->t_flag |= T_TALLOCSTK;
 392                 t->t_swap = stk;
 393         } else {
 394                 t = kmem_cache_alloc(thread_cache, KM_SLEEP);
 395                 bzero(t, sizeof (kthread_t));
 396                 ASSERT(((uintptr_t)t & (PTR24_ALIGN - 1)) == 0);
 397                 if (audit_active)
 398                         audit_thread_create(t);
 399                 /*
 400                  * Initialize t_stk to the kernel stack pointer to use
 401                  * upon entry to the kernel
 402                  */
 403 #ifdef STACK_GROWTH_DOWN
 404                 t->t_stk = stk + stksize;
 405                 t->t_stkbase = stk;
 406 #else
 407                 t->t_stk = stk;                 /* 3b2-like */
 408                 t->t_stkbase = stk + stksize;
 409 #endif /* STACK_GROWTH_DOWN */
 410         }

 412         if (kmem_stackinfo != 0) {
 413                 stkinfo_begin(t);
 414         }

 416         t->t_ts = ts;

 418         /*
 419          * p_cred could be NULL if it thread_create is called before cred_init
 420          * is called in main.
 421          */
 422         mutex_enter(&pp->p_crlock);
 423         if (pp->p_cred)
 424                 crhold(t->t_cred = pp->p_cred);
 425         mutex_exit(&pp->p_crlock);
 426         t->t_start = gethrestime_sec();
 427         t->t_startpc = proc;
 428         t->t_procp = pp;
 429         t->t_clfuncs = &sys_classfuncs.thread;
 430         t->t_cid = syscid;
 431         t->t_pri = pri;
 432         t->t_schedflag = 0;
```

```
432         t->t_stime = ddi_get_lbolt();
433         t->t_schedflag = TS_LOAD | TS_DONT_SWAP;
433         t->t_bind_cpu = PBIND_NONE;
434         t->t_bindflag = (uchar_t)default_binding_mode;
435         t->t_bind_pset = PS_NONE;
436         t->t_plockp = &pp->p_lock;
437         t->t_copyops = NULL;
438         t->t_taskq = NULL;
439         t->t_anttime = 0;
440         t->t_hatdepth = 0;

442         t->t_dtrace_vtime = 1;  /* assure vtimestamp is always non-zero */

444         CPU_STATS_ADDQ(CPU, sys, nthreads, 1);
445 #ifndef NPROBE
446         /* Kernel probe */
447         tnf_thread_create(t);
448 #endif /* NPROBE */
449         LOCK_INIT_CLEAR(&t->t_lock);

451         /*
452          * Callers who give us a NULL proc must do their own
453          * stack initialization.  e.g. lwp_create()
454          */
455         if (proc != NULL) {
456                 t->t_stk = thread_stk_init(t->t_stk);
457                 thread_load(t, proc, arg, len);
458         }

460         /*
461          * Put a hold on project0. If this thread is actually in a
462          * different project, then t_proj will be changed later in
463          * lwp_create().  All kernel-only threads must be in project 0.
464          */
465         t->t_proj = project_hold(proj0p);

467         lgrp_affinity_init(&t->t_lgrp_affinity);

469         mutex_enter(&pidlock);
470         nthread++;
471         t->t_did = next_t_id++;
472         t->t_prev = curthread->t_prev;
473         t->t_next = curthread;

475         /*
476          * Add the thread to the list of all threads, and initialize
477          * its t_cpu pointer.  We need to block preemption since
478          * cpu_offline walks the thread list looking for threads
479          * with t_cpu pointing to the CPU being offlined.  We want
480          * to make sure that the list is consistent and that if t_cpu
481          * is set, the thread is on the list.
482          */
483         kpreempt_disable();
484         curthread->t_prev->t_next = t;
485         curthread->t_prev = t;

487         /*
488          * Threads should never have a NULL t_cpu pointer so assign it
489          * here.  If the thread is being created with state TS_RUN a
490          * better CPU may be chosen when it is placed on the run queue.
491          *
492          * We need to keep kernel preemption disabled when setting all
493          * three fields to keep them in sync.  Also, always create in
494          * the default partition since that's where kernel threads go
495          * (if this isn't a kernel thread, t_cpupart will be changed
496          * in lwp_create before setting the thread runnable).
```



```
497          */
498         t->t_cpupart = &cp_default;

500         /*
501          * For now, affiliate this thread with the root lgroup.
502          * Since the kernel does not (presently) allocate its memory
503          * in a locality aware fashion, the root is an appropriate home.
504          * If this thread is later associated with an lwp, it will have
505          * it's lgroup re-assigned at that time.
506          */
507         lgrp_move_thread(t, &cp_default.cp_lgrploads[LGRP_ROOTID], 1);

509         /*
510          * Inherit the current cpu.  If this cpu isn't part of the chosen
511          * lgroup, a new cpu will be chosen by cpu_choose when the thread
512          * is ready to run.
513          */
514         if (CPU->cpu_part == &cp_default)
515                 t->t_cpu = CPU;
516         else
517                 t->t_cpu = disp_lowpri_cpu(cp_default.cp_cpulist, t->t_lpl,
518                     t->t_pri, NULL);

520         t->t_disp_queue = t->t_cpu->cpu_disp;
521         kpreempt_enable();

523         /*
524          * Initialize thread state and the dispatcher lock pointer.
525          * Need to hold onto pidlock to block allthreads walkers until
526          * the state is set.
527          */
528         switch (state) {
529         case TS_RUN:
530                 curthread->t_oldspl = splhigh();        /* get dispatcher spl */
531                 THREAD_SET_STATE(t, TS_STOPPED, &transition_lock);
532                 CL_SETRUN(t);
533                 thread_unlock(t);
534                 break;

536         case TS_ONPROC:
537                 THREAD_ONPROC(t, t->t_cpu);
538                 break;

540         case TS_FREE:
541                 /*
542                  * Free state will be used for intr threads.
543                  * The interrupt routine must set the thread dispatcher
544                  * lock pointer (t_lockp) if starting on a CPU
545                  * other than the current one.
546                  */
547                 THREAD_FREEINTR(t, CPU);
548                 break;

550         case TS_STOPPED:
551                 THREAD_SET_STATE(t, TS_STOPPED, &stop_lock);
552                 break;

554         default:                        /* TS_SLEEP, TS_ZOMB or TS_TRANS */
555                 cmn_err(CE_PANIC, "thread_create: invalid state %d", state);
556         }
557         mutex_exit(&pidlock);
558         return (t);
559 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   57791 Fri May  8 18:10:24 2015
new/usr/src/uts/common/disp/ts.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_


 184 static int      ts_admin(caddr_t, cred_t *);
 185 static int      ts_enterclass(kthread_t *, id_t, void *, cred_t *, void *);
 186 static int      ts_fork(kthread_t *, kthread_t *, void *);
 187 static int      ts_getclinfo(void *);
 188 static int      ts_getclpri(pcpri_t *);
 189 static int      ts_parmsin(void *);
 190 static int      ts_parmsout(void *, pc_vaparms_t *);
 191 static int      ts_vaparmsin(void *, pc_vaparms_t *);
 192 static int      ts_vaparmsout(void *, pc_vaparms_t *);
 193 static int      ts_parmsset(kthread_t *, void *, id_t, cred_t *);
 194 static void     ts_exit(kthread_t *);
 195 static int      ts_donice(kthread_t *, cred_t *, int, int *);
 196 static int      ts_doprio(kthread_t *, cred_t *, int, int *);
 197 static void     ts_exitclass(void *);
 198 static int      ts_canexit(kthread_t *, cred_t *);
 199 static void     ts_forkret(kthread_t *, kthread_t *);
 200 static void     ts_nullsys();
 201 static void     ts_parmsget(kthread_t *, void *);
 202 static void     ts_preempt(kthread_t *);
 203 static void     ts_setrun(kthread_t *);
 204 static void     ts_sleep(kthread_t *);
 205 static pri_t    ts_swapin(kthread_t *, int);
 206 static pri_t    ts_swapout(kthread_t *, int);
 205 static void     ts_tick(kthread_t *);
 206 static void     ts_trapret(kthread_t *);
 207 static void     ts_update(void *);
 208 static int      ts_update_list(int);
 209 static void     ts_wakeup(kthread_t *);
 210 static pri_t    ts_globpri(kthread_t *);
 211 static void     ts_yield(kthread_t *);
 212 extern tsdpent_t *ts_getdptbl(void);
 213 extern pri_t    *ts_getkmdpris(void);
 214 extern pri_t    td_getmaxumdpri(void);
 215 static int      ts_alloc(void **, int);
 216 static void     ts_free(void *);

 218 pri_t           ia_init(id_t, int, classfuncs_t **);
 219 static int      ia_getclinfo(void *);
 220 static int      ia_getclpri(pcpri_t *);
 221 static int      ia_parmsin(void *);
 222 static int      ia_vaparmsin(void *, pc_vaparms_t *);
 223 static int      ia_vaparmsout(void *, pc_vaparms_t *);
 224 static int      ia_parmsset(kthread_t *, void *, id_t, cred_t *);
 225 static void     ia_parmsget(kthread_t *, void *);
 226 static void     ia_set_process_group(pid_t, pid_t, pid_t);

 228 static void     ts_change_priority(kthread_t *, tsproc_t *);

 230 extern pri_t    ts_maxkmdpri;   /* maximum kernel mode ts priority */
 231 static pri_t    ts_maxglobpri;  /* maximum global priority used by ts class */
 232 static kmutex_t ts_dptblock;    /* protects time sharing dispatch table */
 233 static kmutex_t ts_list_lock[TS_LISTS]; /* protects tsproc lists */
```



```
 234 static tsproc_t ts_plisthead[TS_LISTS]; /* dummy tsproc at head of lists */

 236 static gid_t    IA_gid = 0;

 238 static struct classfuncs ts_classfuncs = {
 239         /* class functions */
 240         ts_admin,
 241         ts_getclinfo,
 242         ts_parmsin,
 243         ts_parmsout,
 244         ts_vaparmsin,
 245         ts_vaparmsout,
 246         ts_getclpri,
 247         ts_alloc,
 248         ts_free,

 250         /* thread functions */
 251         ts_enterclass,
 252         ts_exitclass,
 253         ts_canexit,
 254         ts_fork,
 255         ts_forkret,
 256         ts_parmsget,
 257         ts_parmsset,
 258         ts_nullsys,     /* stop */
 259         ts_exit,
 260         ts_nullsys,     /* active */
 261         ts_nullsys,     /* inactive */
 264         ts_swapin,
 265         ts_swapout,
 262         ts_trapret,
 263         ts_preempt,
 264         ts_setrun,
 265         ts_sleep,
 266         ts_tick,
 267         ts_wakeup,
 268         ts_donice,
 269         ts_globpri,
 270         ts_nullsys,     /* set_process_group */
 271         ts_yield,
 272         ts_doprio,
 273 };

 275 /*
 276  * ia_classfuncs is used for interactive class threads; IA threads are stored
 277  * on the same class list as TS threads, and most of the class functions are
 278  * identical, but a few have different enough functionality to require their
 279  * own functions.
 280  */
 281 static struct classfuncs ia_classfuncs = {
 282         /* class functions */
 283         ts_admin,
 284         ia_getclinfo,
 285         ia_parmsin,
 286         ts_parmsout,
 287         ia_vaparmsin,
 288         ia_vaparmsout,
 289         ia_getclpri,
 290         ts_alloc,
 291         ts_free,

 293         /* thread functions */
 294         ts_enterclass,
 295         ts_exitclass,
 296         ts_canexit,
 297         ts_fork,
```

```
 298         ts_forkret,
 299         ia_parmsget,
 300         ia_parmsset,
 301         ts_nullsys,     /* stop */
 302         ts_exit,
 303         ts_nullsys,     /* active */
 304         ts_nullsys,     /* inactive */
 309         ts_swapin,
 310         ts_swapout,
 305         ts_trapret,
 306         ts_preempt,
 307         ts_setrun,
 308         ts_sleep,
 309         ts_tick,
 310         ts_wakeup,
 311         ts_donice,
 312         ts_globpri,
 313         ia_set_process_group,
 314         ts_yield,
 315         ts_doprio,
 316 };
_____unchanged_portion_omitted_

1360 /*
1361  * Arrange for thread to be placed in appropriate location
1362  * on dispatcher queue.
1363  *
1364  * This is called with the current thread in TS_ONPROC and locked.
1365  */
1366 static void
1367 ts_preempt(kthread_t *t)
1368 {
1369         tsproc_t        *tspp = (tsproc_t *)(t->t_cldata);
1370         klwp_t          *lwp = curthread->t_lwp;
1371         pri_t           oldpri = t->t_pri;

1373         ASSERT(t == curthread);
1374         ASSERT(THREAD_LOCK_HELD(curthread));

1376         /*
1377          * If preempted in the kernel, make sure the thread has
1378          * a kernel priority if needed.
1379          */
1380         if (!(tspp->ts_flags & TSKPRI) && lwp != NULL && t->t_kpri_req) {
1381                 tspp->ts_flags |= TSKPRI;
1382                 THREAD_CHANGE_PRI(t, ts_kmdpris[0]);
1383                 ASSERT(t->t_pri >= 0 && t->t_pri <= ts_maxglobpri);
1384                 t->t_trapret = 1;               /* so ts_trapret will run */
1385                 aston(t);
1386         }

1388         /*
1389          * This thread may be placed on wait queue by CPU Caps. In this case we
1390          * do not need to do anything until it is removed from the wait queue.
1391          * Do not enforce CPU caps on threads running at a kernel priority
1392          */
1393         if (CPUCAPS_ON()) {
1394                 (void) cpucaps_charge(t, &tspp->ts_caps,
1395                     CPUCAPS_CHARGE_ENFORCE);
1396                 if (!(tspp->ts_flags & TSKPRI) && CPUCAPS_ENFORCE(t))
1397                         return;
1398         }

1400         /*
1407          * If thread got preempted in the user-land then we know
1408          * it isn't holding any locks.  Mark it as swappable.
```



```
1409          */
1410         ASSERT(t->t_schedflag & TS_DONT_SWAP);
1411         if (lwp != NULL && lwp->lwp_state == LWP_USER)
1412                 t->t_schedflag &= ~TS_DONT_SWAP;

1414         /*
1401          * Check to see if we're doing "preemption control" here.  If
1402          * we are, and if the user has requested that this thread not
1403          * be preempted, and if preemptions haven't been put off for
1404          * too long, let the preemption happen here but try to make
1405          * sure the thread is rescheduled as soon as possible.  We do
1406          * this by putting it on the front of the highest priority run
1407          * queue in the TS class.  If the preemption has been put off
1408          * for too long, clear the "nopreempt" bit and let the thread
1409          * be preempted.
1410          */
1411         if (t->t_schedctl && schedctl_get_nopreempt(t)) {
1412                 if (tspp->ts_timeleft > -SC_MAX_TICKS) {
1413                         DTRACE_SCHED1(schedctl__nopreempt, kthread_t *, t);
1414                         if (!(tspp->ts_flags & TSKPRI)) {
1415                                 /*
1416                                  * If not already remembered, remember current
1417                                  * priority for restoration in ts_yield().
1418                                  */
1419                                 if (!(tspp->ts_flags & TSRESTORE)) {
1420                                         tspp->ts_scpri = t->t_pri;
1421                                         tspp->ts_flags |= TSRESTORE;
1422                                 }
1423                                 THREAD_CHANGE_PRI(t, ts_maxumdpri);
1438                                 t->t_schedflag |= TS_DONT_SWAP;
1424                         }
1425                         schedctl_set_yield(t, 1);
1426                         setfrontdq(t);
1427                         goto done;
1428                 } else {
1429                         if (tspp->ts_flags & TSRESTORE) {
1430                                 THREAD_CHANGE_PRI(t, tspp->ts_scpri);
1431                                 tspp->ts_flags &= ~TSRESTORE;
1432                         }
1433                         schedctl_set_nopreempt(t, 0);
1434                         DTRACE_SCHED1(schedctl__preempt, kthread_t *, t);
1435                         TNF_PROBE_2(schedctl_preempt, "schedctl TS ts_preempt",
1436                             /* CSTYLED */, tnf_pid, pid, ttoproc(t)->p_pid,
1437                             tnf_lwpid, lwpid, t->t_tid);
1438                         /*
1439                          * Fall through and be preempted below.
1440                          */
1441                 }
1442         }

1444         if ((tspp->ts_flags & (TSBACKQ|TSKPRI)) == TSBACKQ) {
1445                 tspp->ts_timeleft = ts_dptbl[tspp->ts_cpupri].ts_quantum;
1446                 tspp->ts_dispwait = 0;
1447                 tspp->ts_flags &= ~TSBACKQ;
1448                 setbackdq(t);
1449         } else if ((tspp->ts_flags & (TSBACKQ|TSKPRI)) == (TSBACKQ|TSKPRI)) {
1450                 tspp->ts_flags &= ~TSBACKQ;
1451                 setbackdq(t);
1452         } else {
1453                 setfrontdq(t);
1454         }

1456 done:
1457         TRACE_2(TR_FAC_DISP, TR_PREEMPT,
1458             "preempt:tid %p old pri %d", t, oldpri);
1459 }
_____unchanged_portion_omitted_
```

```
1496 /*
1497  * Prepare thread for sleep. We reset the thread priority so it will
1498  * run at the kernel priority level when it wakes up.
1499  */
1500 static void
1501 ts_sleep(kthread_t *t)
1502 {
1503         tsproc_t        *tspp = (tsproc_t *)(t->t_cldata);
1504         int             flags;
1505         pri_t           old_pri = t->t_pri;

1507         ASSERT(t == curthread);
1508         ASSERT(THREAD_LOCK_HELD(t));

1510         /*
1511          * Account for time spent on CPU before going to sleep.
1512          */
1513         (void) CPUCAPS_CHARGE(t, &tspp->ts_caps, CPUCAPS_CHARGE_ENFORCE);

1515         flags = tspp->ts_flags;
1516         if (t->t_kpri_req) {
1517                 tspp->ts_flags = flags | TSKPRI;
1518                 THREAD_CHANGE_PRI(t, ts_kmdpris[0]);
1519                 ASSERT(t->t_pri >= 0 && t->t_pri <= ts_maxglobpri);
1520                 t->t_trapret = 1;               /* so ts_trapret will run */
1521                 aston(t);
1522         } else if (tspp->ts_dispwait > ts_dptbl[tspp->ts_umdpri].ts_maxwait) {
1523                 /*
1524                  * If thread has blocked in the kernel (as opposed to
1525                  * being merely preempted), recompute the user mode priority.
1526                  */
1527                 tspp->ts_cpupri = ts_dptbl[tspp->ts_cpupri].ts_slpret;
1528                 TS_NEWUMDPRI(tspp);
1529                 tspp->ts_timeleft = ts_dptbl[tspp->ts_cpupri].ts_quantum;
1530                 tspp->ts_dispwait = 0;

1532                 THREAD_CHANGE_PRI(curthread,
1533                     ts_dptbl[tspp->ts_umdpri].ts_globpri);
1534                 ASSERT(curthread->t_pri >= 0 &&
1535                     curthread->t_pri <= ts_maxglobpri);
1536                 tspp->ts_flags = flags & ~TSKPRI;

1538                 if (DISP_MUST_SURRENDER(curthread))
1539                         cpu_surrender(curthread);
1540         } else if (flags & TSKPRI) {
1541                 THREAD_CHANGE_PRI(curthread,
1542                     ts_dptbl[tspp->ts_umdpri].ts_globpri);
1543                 ASSERT(curthread->t_pri >= 0 &&
1544                     curthread->t_pri <= ts_maxglobpri);
1545                 tspp->ts_flags = flags & ~TSKPRI;

1547                 if (DISP_MUST_SURRENDER(curthread))
1548                         cpu_surrender(curthread);
1549         }
1565         t->t_stime = ddi_get_lbolt();           /* time stamp for the swapper */
1550         TRACE_2(TR_FAC_DISP, TR_SLEEP,
1551             "sleep:tid %p old pri %d", t, old_pri);
1552 }

1571 /*
1572  * Return Values:
1573  *
1574  *      -1 if the thread is loaded or is not eligible to be swapped in.
```



```
1575  *
1576  *      effective priority of the specified thread based on swapout time
1577  *              and size of process (epri >= 0 , epri <= SHRT_MAX).
1578  */
1579 /* ARGSUSED */
1580 static pri_t
1581 ts_swapin(kthread_t *t, int flags)
1582 {
1583         tsproc_t        *tspp = (tsproc_t *)(t->t_cldata);
1584         long            epri = -1;
1585         proc_t          *pp = ttoproc(t);

1587         ASSERT(THREAD_LOCK_HELD(t));

1589         /*
1590          * We know that pri_t is a short.
1591          * Be sure not to overrun its range.
1592          */
1593         if (t->t_state == TS_RUN && (t->t_schedflag & TS_LOAD) == 0) {
1594                 time_t swapout_time;

1596                 swapout_time = (ddi_get_lbolt() - t->t_stime) / hz;
1597                 if (INHERITED(t) || (tspp->ts_flags & (TSKPRI | TSIASET)))
1598                         epri = (long)DISP_PRIO(t) + swapout_time;
1599                 else {
1600                         /*
1601                          * Threads which have been out for a long time,
1602                          * have high user mode priority and are associated
1603                          * with a small address space are more deserving
1604                          */
1605                         epri = ts_dptbl[tspp->ts_umdpri].ts_globpri;
1606                         ASSERT(epri >= 0 && epri <= ts_maxumdpri);
1607                         epri += swapout_time - pp->p_swrss / nz(maxpgio)/2;
1608                 }
1609                 /*
1610                  * Scale epri so SHRT_MAX/2 represents zero priority.
1611                  */
1612                 epri += SHRT_MAX/2;
1613                 if (epri < 0)
1614                         epri = 0;
1615                 else if (epri > SHRT_MAX)
1616                         epri = SHRT_MAX;
1617         }
1618         return ((pri_t)epri);
1619 }

1621 /*
1622  * Return Values
1623  *      -1 if the thread isn't loaded or is not eligible to be swapped out.
1624  *
1625  *      effective priority of the specified thread based on if the swapper
1626  *              is in softswap or hardswap mode.
1627  *
1628  *              Softswap:  Return a low effective priority for threads
1629  *                         sleeping for more than maxslp secs.
1630  *
1631  *              Hardswap:  Return an effective priority such that threads
1632  *                         which have been in memory for a while and are
1633  *                         associated with a small address space are swapped
1634  *                         in before others.
1635  *
1636  *              (epri >= 0 , epri <= SHRT_MAX).
1637  */
1638 time_t  ts_minrun = 2;          /* XXX - t_pri becomes 59 within 2 secs */
1639 time_t  ts_minslp = 2;          /* min time on sleep queue for hardswap */
```

```
1641 static pri_t
1642 ts_swapout(kthread_t *t, int flags)
1643 {
1644         tsproc_t        *tspp = (tsproc_t *)(t->t_cldata);
1645         long            epri = -1;
1646         proc_t          *pp = ttoproc(t);
1647         time_t          swapin_time;

1649         ASSERT(THREAD_LOCK_HELD(t));

1651         if (INHERITED(t) || (tspp->ts_flags & (TSKPRI | TSIASET)) ||
1652             (t->t_proc_flag & TP_LWPEXIT) ||
1653             (t->t_state & (TS_ZOMB | TS_FREE | TS_STOPPED |
1654             TS_ONPROC | TS_WAIT)) ||
1655             !(t->t_schedflag & TS_LOAD) || !SWAP_OK(t))
1656                 return (-1);

1658         ASSERT(t->t_state & (TS_SLEEP | TS_RUN));

1660         /*
1661          * We know that pri_t is a short.
1662          * Be sure not to overrun its range.
1663          */
1664         swapin_time = (ddi_get_lbolt() - t->t_stime) / hz;
1665         if (flags == SOFTSWAP) {
1666                 if (t->t_state == TS_SLEEP && swapin_time > maxslp) {
1667                         epri = 0;
1668                 } else {
1669                         return ((pri_t)epri);
1670                 }
1671         } else {
1672                 pri_t pri;

1674                 if ((t->t_state == TS_SLEEP && swapin_time > ts_minslp) ||
1675                     (t->t_state == TS_RUN && swapin_time > ts_minrun)) {
1676                         pri = ts_dptbl[tspp->ts_umdpri].ts_globpri;
1677                         ASSERT(pri >= 0 && pri <= ts_maxumdpri);
1678                         epri = swapin_time -
1679                             (rm_asrss(pp->p_as) / nz(maxpgio)/2) - (long)pri;
1680                 } else {
1681                         return ((pri_t)epri);
1682                 }
1683         }

1685         /*
1686          * Scale epri so SHRT_MAX/2 represents zero priority.
1687          */
1688         epri += SHRT_MAX/2;
1689         if (epri < 0)
1690                 epri = 0;
1691         else if (epri > SHRT_MAX)
1692                 epri = SHRT_MAX;

1694         return ((pri_t)epri);
1695 }
```

```
1554 /*
1555  * Check for time slice expiration.  If time slice has expired
1556  * move thread to priority specified in tsdptbl for time slice expiration
1557  * and set runrun to cause preemption.
1558  */
1559 static void
1560 ts_tick(kthread_t *t)
1561 {
1562         tsproc_t *tspp = (tsproc_t *)(t->t_cldata);
1706         klwp_t *lwp;
```



```
1563         boolean_t call_cpu_surrender = B_FALSE;
1564         pri_t   oldpri = t->t_pri;

1566         ASSERT(MUTEX_HELD(&(ttoproc(t))->p_lock));

1568         thread_lock(t);

1570         /*
1571          * Keep track of thread's project CPU usage.  Note that projects
1572          * get charged even when threads are running in the kernel.
1573          */
1574         if (CPUCAPS_ON()) {
1575                 call_cpu_surrender = cpucaps_charge(t, &tspp->ts_caps,
1576                     CPUCAPS_CHARGE_ENFORCE) && !(tspp->ts_flags & TSKPRI);
1577         }

1579         if ((tspp->ts_flags & TSKPRI) == 0) {
1580                 if (--tspp->ts_timeleft <= 0) {
1581                         pri_t   new_pri;

1583                         /*
1584                          * If we're doing preemption control and trying to
1585                          * avoid preempting this thread, just note that
1586                          * the thread should yield soon and let it keep
1587                          * running (unless it's been a while).
1588                          */
1589                         if (t->t_schedctl && schedctl_get_nopreempt(t)) {
1590                                 if (tspp->ts_timeleft > -SC_MAX_TICKS) {
1591                                         DTRACE_SCHED1(schedctl__nopreempt,
1592                                             kthread_t *, t);
1593                                         schedctl_set_yield(t, 1);
1594                                         thread_unlock_nopreempt(t);
1595                                         return;
1596                                 }

1598                                 TNF_PROBE_2(schedctl_failsafe,
1599                                     "schedctl TS ts_tick", /* CSTYLED */,
1600                                     tnf_pid, pid, ttoproc(t)->p_pid,
1601                                     tnf_lwpid, lwpid, t->t_tid);
1602                         }
1603                         tspp->ts_flags &= ~TSRESTORE;
1604                         tspp->ts_cpupri = ts_dptbl[tspp->ts_cpupri].ts_tqexp;
1605                         TS_NEWUMDPRI(tspp);
1606                         tspp->ts_dispwait = 0;
1607                         new_pri = ts_dptbl[tspp->ts_umdpri].ts_globpri;
1608                         ASSERT(new_pri >= 0 && new_pri <= ts_maxglobpri);
1609                         /*
1610                          * When the priority of a thread is changed,
1611                          * it may be necessary to adjust its position
1612                          * on a sleep queue or dispatch queue.
1613                          * The function thread_change_pri accomplishes
1614                          * this.
1615                          */
1616                         if (thread_change_pri(t, new_pri, 0)) {
1761                                 if ((t->t_schedflag & TS_LOAD) &&
1762                                     (lwp = t->t_lwp) &&
1763                                     lwp->lwp_state == LWP_USER)
1764                                         t->t_schedflag &= ~TS_DONT_SWAP;
1617                                 tspp->ts_timeleft =
1618                                     ts_dptbl[tspp->ts_cpupri].ts_quantum;
1619                         } else {
1620                                 call_cpu_surrender = B_TRUE;
1621                         }
1622                         TRACE_2(TR_FAC_DISP, TR_TICK,
1623                             "tick:tid %p old pri %d", t, oldpri);
1624                 } else if (t->t_state == TS_ONPROC &&
```

```
1625                    t->t_pri < t->t_disp_queue->disp_maxrunpri) {
1626                        call_cpu_surrender = B_TRUE;
1627                }
1628        }

1630        if (call_cpu_surrender) {
1631                tspp->ts_flags |= TSBACKQ;
1632                cpu_surrender(t);
1633        }

1635        thread_unlock_nopreempt(t);     /* clock thread can't be preempted */
1636 }


1639 /*
1640  * If thread is currently at a kernel mode priority (has slept)
1641  * we assign it the appropriate user mode priority and time quantum
1642  * here.  If we are lowering the thread's priority below that of
1643  * other runnable threads we will normally set runrun via cpu_surrender() to
1644  * cause preemption.
1645  */
1646 static void
1647 ts_trapret(kthread_t *t)
1648 {
1649        tsproc_t        *tspp = (tsproc_t *)t->t_cldata;
1650        cpu_t           *cp = CPU;
1651        pri_t           old_pri = curthread->t_pri;

1653        ASSERT(THREAD_LOCK_HELD(t));
1654        ASSERT(t == curthread);
1655        ASSERT(cp->cpu_dispthread == t);
1656        ASSERT(t->t_state == TS_ONPROC);

1658        t->t_kpri_req = 0;
1659        if (tspp->ts_dispwait > ts_dptbl[tspp->ts_umdpri].ts_maxwait) {
1660                tspp->ts_cpupri = ts_dptbl[tspp->ts_cpupri].ts_slpret;
1661                TS_NEWUMDPRI(tspp);
1662                tspp->ts_timeleft = ts_dptbl[tspp->ts_cpupri].ts_quantum;
1663                tspp->ts_dispwait = 0;

1665                /*
1666                 * If thread has blocked in the kernel (as opposed to
1667                 * being merely preempted), recompute the user mode priority.
1668                 */
1669                THREAD_CHANGE_PRI(t, ts_dptbl[tspp->ts_umdpri].ts_globpri);
1670                cp->cpu_dispatch_pri = DISP_PRIO(t);
1671                ASSERT(t->t_pri >= 0 && t->t_pri <= ts_maxglobpri);
1672                tspp->ts_flags &= ~TSKPRI;

1674                if (DISP_MUST_SURRENDER(t))
1675                        cpu_surrender(t);
1676        } else if (tspp->ts_flags & TSKPRI) {
1677                /*
1678                 * If thread has blocked in the kernel (as opposed to
1679                 * being merely preempted), recompute the user mode priority.
1680                 */
1681                THREAD_CHANGE_PRI(t, ts_dptbl[tspp->ts_umdpri].ts_globpri);
1682                cp->cpu_dispatch_pri = DISP_PRIO(t);
1683                ASSERT(t->t_pri >= 0 && t->t_pri <= ts_maxglobpri);
1684                tspp->ts_flags &= ~TSKPRI;

1686                if (DISP_MUST_SURRENDER(t))
1687                        cpu_surrender(t);
1688        }

1838        /*
```



```
1839         * Swapout lwp if the swapper is waiting for this thread to
1840         * reach a safe point.
1841         */
1842        if ((t->t_schedflag & TS_SWAPENQ) && !(tspp->ts_flags & TSIASET)) {
1843                thread_unlock(t);
1844                swapout_lwp(ttolwp(t));
1845                thread_lock(t);
1846        }

1690        TRACE_2(TR_FAC_DISP, TR_TRAPRET,
1691            "trapret:tid %p old pri %d", t, old_pri);
1692 }
______unchanged_portion_omitted_

1812 /*
1813  * Processes waking up go to the back of their queue.  We don't
1814  * need to assign a time quantum here because thread is still
1815  * at a kernel mode priority and the time slicing is not done
1816  * for threads running in the kernel after sleeping.  The proper
1817  * time quantum will be assigned by ts_trapret before the thread
1818  * returns to user mode.
1819  */
1820 static void
1821 ts_wakeup(kthread_t *t)
1822 {
1823        tsproc_t        *tspp = (tsproc_t *)(t->t_cldata);

1825        ASSERT(THREAD_LOCK_HELD(t));

1985        t->t_stime = ddi_get_lbolt();           /* time stamp for the swapper */

1827        if (tspp->ts_flags & TSKPRI) {
1828                tspp->ts_flags &= ~TSBACKQ;
1829                if (tspp->ts_flags & TSIASET)
1830                        setfrontdq(t);
1831                else
1832                        setbackdq(t);
1833        } else if (t->t_kpri_req) {
1834                /*
1835                 * Give thread a priority boost if we were asked.
1836                 */
1837                tspp->ts_flags |= TSKPRI;
1838                THREAD_CHANGE_PRI(t, ts_kmdpris[0]);
1839                setbackdq(t);
1840                t->t_trapret = 1;       /* so that ts_trapret will run */
1841                aston(t);
1842        } else {
1843                if (tspp->ts_dispwait > ts_dptbl[tspp->ts_umdpri].ts_maxwait) {
1844                        tspp->ts_cpupri = ts_dptbl[tspp->ts_cpupri].ts_slpret;
1845                        TS_NEWUMDPRI(tspp);
1846                        tspp->ts_timeleft =
1847                            ts_dptbl[tspp->ts_cpupri].ts_quantum;
1848                        tspp->ts_dispwait = 0;
1849                        THREAD_CHANGE_PRI(t,
1850                            ts_dptbl[tspp->ts_umdpri].ts_globpri);
1851                        ASSERT(t->t_pri >= 0 && t->t_pri <= ts_maxglobpri);
1852                }

1854                tspp->ts_flags &= ~TSBACKQ;

1856                if (tspp->ts_flags & TSIA) {
1857                        if (tspp->ts_flags & TSIASET)
1858                                setfrontdq(t);
1859                        else
1860                                setbackdq(t);
1861                } else {
```

```
1862                         if (t->t_disp_time != ddi_get_lbolt())
1863                                 setbackdq(t);
1864                         else
1865                                 setfrontdq(t);
1866                 }
1867         }
1868 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   55638 Fri May  8 18:10:24 2015
new/usr/src/uts/common/exec/elf/elf.c
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

1424 #ifdef _ELF32_COMPAT
1425 extern size_t elf_datasz_max;
1426 #else
1427 size_t elf_datasz_max = 1 * 1024 * 1024;
1428 #endif

1430 /*
1431  * This function processes mappings that correspond to load objects to
1432  * examine their respective sections for elfcore(). It's called once with
1433  * v set to NULL to count the number of sections that we're going to need
1434  * and then again with v set to some allocated buffer that we fill in with
1435  * all the section data.
1436  */
1437 static int
1438 process_scns(core_content_t content, proc_t *p, cred_t *credp, vnode_t *vp,
1439     Shdr *v, int nv, rlim64_t rlimit, Off *doffsetp, int *nshdrsp)
1440 {
1441         vnode_t *lastvp = NULL;
1442         struct seg *seg;
1443         int i, j;
1444         void *data = NULL;
1445         size_t datasz = 0;
1446         shstrtab_t shstrtab;
1447         struct as *as = p->p_as;
1448         int error = 0;

1450         if (v != NULL)
1451                 shstrtab_init(&shstrtab);

1453         i = 1;
1454         for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {
1455                 uint_t prot;
1456                 vnode_t *mvp;
1457                 void *tmp = NULL;
1458                 caddr_t saddr = seg->s_base;
1459                 caddr_t naddr;
1460                 caddr_t eaddr;
1461                 size_t segsize;

1463                 Ehdr ehdr;
1464                 int nshdrs, shstrndx, nphdrs;
1465                 caddr_t shbase;
1466                 ssize_t shsize;
1467                 char *shstrbase;
1468                 ssize_t shstrsize;

1470                 Shdr *shdr;
1471                 const char *name;
1472                 size_t sz;
1473                 uintptr_t off;

1475                 int ctf_ndx = 0;
1476                 int symtab_ndx = 0;

1478                 /*
1479                  * Since we're just looking for text segments of load
1480                  * objects, we only care about the protection bits; we don't
```



```
1481                  * care about the actual size of the segment so we use the
1482                  * reserved size. If the segment's size is zero, there's
1483                  * something fishy going on so we ignore this segment.
1484                  */
1485                 if (seg->s_ops != &segvn_ops ||
1486                     segop_getvp(seg, seg->s_base, &mvp) != 0 ||
1486                     SEGOP_GETVP(seg, seg->s_base, &mvp) != 0 ||
1487                     mvp == lastvp || mvp == NULL || mvp->v_type != VREG ||
1488                     (segsize = pr_getsegsize(seg, 1)) == 0)
1489                         continue;

1491                 eaddr = saddr + segsize;
1492                 prot = pr_getprot(seg, 1, &tmp, &saddr, &naddr, eaddr);
1493                 pr_getprot_done(&tmp);

1495                 /*
1496                  * Skip this segment unless the protection bits look like
1497                  * what we'd expect for a text segment.
1498                  */
1499                 if ((prot & (PROT_WRITE | PROT_EXEC)) != PROT_EXEC)
1500                         continue;

1502                 if (getelfhead(mvp, credp, &ehdr, &nshdrs, &shstrndx,
1503                     &nphdrs) != 0 ||
1504                     getelfshdr(mvp, credp, &ehdr, nshdrs, shstrndx,
1505                     &shbase, &shsize, &shstrbase, &shstrsize) != 0)
1506                         continue;

1508                 off = ehdr.e_shentsize;
1509                 for (j = 1; j < nshdrs; j++, off += ehdr.e_shentsize) {
1510                         Shdr *symtab = NULL, *strtab;

1512                         shdr = (Shdr *)(shbase + off);

1514                         if (shdr->sh_name >= shstrsize)
1515                                 continue;

1517                         name = shstrbase + shdr->sh_name;

1519                         if (strcmp(name, shstrtab_data[STR_CTF]) == 0) {
1520                                 if ((content & CC_CONTENT_CTF) == 0 ||
1521                                     ctf_ndx != 0)
1522                                         continue;

1524                                 if (shdr->sh_link > 0 &&
1525                                     shdr->sh_link < nshdrs) {
1526                                         symtab = (Shdr *)(shbase +
1527                                             shdr->sh_link * ehdr.e_shentsize);
1528                                 }

1530                                 if (v != NULL && i < nv - 1) {
1531                                         if (shdr->sh_size > datasz &&
1532                                             shdr->sh_size <= elf_datasz_max) {
1533                                                 if (data != NULL)
1534                                                         kmem_free(data, datasz);

1536                                                 datasz = shdr->sh_size;
1537                                                 data = kmem_alloc(datasz,
1538                                                     KM_SLEEP);
1539                                         }

1541                                         v[i].sh_name = shstrtab_ndx(&shstrtab,
1542                                             STR_CTF);
1543                                         v[i].sh_addr = (Addr)(uintptr_t)saddr;
1544                                         v[i].sh_type = SHT_PROGBITS;
1545                                         v[i].sh_addralign = 4;
```

```
1546                                    *doffsetp = roundup(*doffsetp,
1547                                        v[i].sh_addralign);
1548                                    v[i].sh_offset = *doffsetp;
1549                                    v[i].sh_size = shdr->sh_size;
1550                                    if (symtab == NULL)  {
1551                                            v[i].sh_link = 0;
1552                                    } else if (symtab->sh_type ==
1553                                        SHT_SYMTAB &&
1554                                        symtab_ndx != 0) {
1555                                            v[i].sh_link =
1556                                                symtab_ndx;
1557                                    } else {
1558                                            v[i].sh_link = i + 1;
1559                                    }

1561                                    copy_scn(shdr, mvp, &v[i], vp,
1562                                        doffsetp, data, datasz, credp,
1563                                        rlimit);
1564                            }

1566                            ctf_ndx = i++;

1568                            /*
1569                             * We've already dumped the symtab.
1570                             */
1571                            if (symtab != NULL &&
1572                                symtab->sh_type == SHT_SYMTAB &&
1573                                symtab_ndx != 0)
1574                                    continue;

1576                    } else if (strcmp(name,
1577                        shstrtab_data[STR_SYMTAB]) == 0) {
1578                            if ((content & CC_CONTENT_SYMTAB) == 0 ||
1579                                symtab != 0)
1580                                    continue;

1582                            symtab = shdr;
1583                    }

1585                    if (symtab != NULL) {
1586                            if ((symtab->sh_type != SHT_DYNSYM &&
1587                                symtab->sh_type != SHT_SYMTAB) ||
1588                                symtab->sh_link == 0 ||
1589                                symtab->sh_link >= nshdrs)
1590                                    continue;

1592                            strtab = (Shdr *)(shbase +
1593                                symtab->sh_link * ehdr.e_shentsize);

1595                            if (strtab->sh_type != SHT_STRTAB)
1596                                    continue;

1598                            if (v != NULL && i < nv - 2) {
1599                                    sz = MAX(symtab->sh_size,
1600                                        strtab->sh_size);
1601                                    if (sz > datasz &&
1602                                        sz <= elf_datasz_max) {
1603                                            if (data != NULL)
1604                                                    kmem_free(data, datasz);

1606                                            datasz = sz;
1607                                            data = kmem_alloc(datasz,
1608                                                KM_SLEEP);
1609                                    }

1611                                    if (symtab->sh_type == SHT_DYNSYM) {
```



```
1612                                            v[i].sh_name = shstrtab_ndx(
1613                                                &shstrtab, STR_DYNSYM);
1614                                            v[i + 1].sh_name = shstrtab_ndx(
1615                                                &shstrtab, STR_DYNSTR);
1616                                    } else {
1617                                            v[i].sh_name = shstrtab_ndx(
1618                                                &shstrtab, STR_SYMTAB);
1619                                            v[i + 1].sh_name = shstrtab_ndx(
1620                                                &shstrtab, STR_STRTAB);
1621                                    }

1623                                    v[i].sh_type = symtab->sh_type;
1624                                    v[i].sh_addr = symtab->sh_addr;
1625                                    if (ehdr.e_type == ET_DYN ||
1626                                        v[i].sh_addr == 0)
1627                                            v[i].sh_addr +=
1628                                                (Addr)(uintptr_t)saddr;
1629                                    v[i].sh_addralign =
1630                                        symtab->sh_addralign;
1631                                    *doffsetp = roundup(*doffsetp,
1632                                        v[i].sh_addralign);
1633                                    v[i].sh_offset = *doffsetp;
1634                                    v[i].sh_size = symtab->sh_size;
1635                                    v[i].sh_link = i + 1;
1636                                    v[i].sh_entsize = symtab->sh_entsize;
1637                                    v[i].sh_info = symtab->sh_info;

1639                                    copy_scn(symtab, mvp, &v[i], vp,
1640                                        doffsetp, data, datasz, credp,
1641                                        rlimit);

1643                                    v[i + 1].sh_type = SHT_STRTAB;
1644                                    v[i + 1].sh_flags = SHF_STRINGS;
1645                                    v[i + 1].sh_addr = symtab->sh_addr;
1646                                    if (ehdr.e_type == ET_DYN ||
1647                                        v[i + 1].sh_addr == 0)
1648                                            v[i + 1].sh_addr +=
1649                                                (Addr)(uintptr_t)saddr;
1650                                    v[i + 1].sh_addralign =
1651                                        strtab->sh_addralign;
1652                                    *doffsetp = roundup(*doffsetp,
1653                                        v[i + 1].sh_addralign);
1654                                    v[i + 1].sh_offset = *doffsetp;
1655                                    v[i + 1].sh_size = strtab->sh_size;

1657                                    copy_scn(strtab, mvp, &v[i + 1], vp,
1658                                        doffsetp, data, datasz, credp,
1659                                        rlimit);
1660                            }

1662                            if (symtab->sh_type == SHT_SYMTAB)
1663                                    symtab_ndx = i;
1664                            i += 2;
1665                    }
1666            }

1668            kmem_free(shstrbase, shstrsize);
1669            kmem_free(shbase, shsize);

1671            lastvp = mvp;
1672    }

1674    if (v == NULL) {
1675            if (i == 1)
1676                    *nshdrsp = 0;
1677            else
```

```
1678                        *nshdrsp = i + 1;
1679                goto done;
1680        }

1682        if (i != nv - 1) {
1683                cmn_err(CE_WARN, "elfcore: core dump failed for "
1684                    "process %d; address space is changing", p->p_pid);
1685                error = EIO;
1686                goto done;
1687        }

1689        v[i].sh_name = shstrtab_ndx(&shstrtab, STR_SHSTRTAB);
1690        v[i].sh_size = shstrtab_size(&shstrtab);
1691        v[i].sh_addralign = 1;
1692        *doffsetp = roundup(*doffsetp, v[i].sh_addralign);
1693        v[i].sh_offset = *doffsetp;
1694        v[i].sh_flags = SHF_STRINGS;
1695        v[i].sh_type = SHT_STRTAB;

1697        if (v[i].sh_size > datasz) {
1698                if (data != NULL)
1699                        kmem_free(data, datasz);

1701                datasz = v[i].sh_size;
1702                data = kmem_alloc(datasz,
1703                    KM_SLEEP);
1704        }

1706        shstrtab_dump(&shstrtab, data);

1708        if ((error = core_write(vp, UIO_SYSSPACE, *doffsetp,
1709            data, v[i].sh_size, rlimit, credp)) != 0)
1710                goto done;

1712        *doffsetp += v[i].sh_size;

1714 done:
1715        if (data != NULL)
1716                kmem_free(data, datasz);

1718        return (error);
1719 }

1721 int
1722 elfcore(vnode_t *vp, proc_t *p, cred_t *credp, rlim64_t rlimit, int sig,
1723     core_content_t content)
1724 {
1725        offset_t poffset, soffset;
1726        Off doffset;
1727        int error, i, nphdrs, nshdrs;
1728        int overflow = 0;
1729        struct seg *seg;
1730        struct as *as = p->p_as;
1731        union {
1732                Ehdr ehdr;
1733                Phdr phdr[1];
1734                Shdr shdr[1];
1735        } *bigwad;
1736        size_t bigsize;
1737        size_t phdrsz, shdrsz;
1738        Ehdr *ehdr;
1739        Phdr *v;
1740        caddr_t brkbase;
1741        size_t brksize;
1742        caddr_t stkbase;
1743        size_t stksize;
```



```
1744        int ntries = 0;
1745        klwp_t *lwp = ttolwp(curthread);

1747 top:
1748        /*
1749         * Make sure we have everything we need (registers, etc.).
1750         * All other lwps have already stopped and are in an orderly state.
1751         */
1752        ASSERT(p == ttoproc(curthread));
1753        prstop(0, 0);

1755        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1756        nphdrs = prnsegs(as, 0) + 2;            /* two CORE note sections */

1758        /*
1759         * Count the number of section headers we're going to need.
1760         */
1761        nshdrs = 0;
1762        if (content & (CC_CONTENT_CTF | CC_CONTENT_SYMTAB)) {
1763                (void) process_scns(content, p, credp, NULL, NULL, NULL, 0,
1764                    NULL, &nshdrs);
1765        }
1766        AS_LOCK_EXIT(as, &as->a_lock);

1768        ASSERT(nshdrs == 0 || nshdrs > 1);

1770        /*
1771         * The core file contents may required zero section headers, but if
1772         * we overflow the 16 bits allotted to the program header count in
1773         * the ELF header, we'll need that program header at index zero.
1774         */
1775        if (nshdrs == 0 && nphdrs >= PN_XNUM)
1776                nshdrs = 1;

1778        phdrsz = nphdrs * sizeof (Phdr);
1779        shdrsz = nshdrs * sizeof (Shdr);

1781        bigsize = MAX(sizeof (*bigwad), MAX(phdrsz, shdrsz));
1782        bigwad = kmem_alloc(bigsize, KM_SLEEP);

1784        ehdr = &bigwad->ehdr;
1785        bzero(ehdr, sizeof (*ehdr));

1787        ehdr->e_ident[EI_MAG0] = ELFMAG0;
1788        ehdr->e_ident[EI_MAG1] = ELFMAG1;
1789        ehdr->e_ident[EI_MAG2] = ELFMAG2;
1790        ehdr->e_ident[EI_MAG3] = ELFMAG3;
1791        ehdr->e_ident[EI_CLASS] = ELFCLASS;
1792        ehdr->e_type = ET_CORE;

1794 #if !defined(_LP64) || defined(_ELF32_COMPAT)

1796 #if defined(__sparc)
1797        ehdr->e_ident[EI_DATA] = ELFDATA2MSB;
1798        ehdr->e_machine = EM_SPARC;
1799 #elif defined(__i386) || defined(__i386_COMPAT)
1800        ehdr->e_ident[EI_DATA] = ELFDATA2LSB;
1801        ehdr->e_machine = EM_386;
1802 #else
1803 #error "no recognized machine type is defined"
1804 #endif

1806 #else   /* !defined(_LP64) || defined(_ELF32_COMPAT) */

1808 #if defined(__sparc)
1809        ehdr->e_ident[EI_DATA] = ELFDATA2MSB;
```

```
1810         ehdr->e_machine = EM_SPARCV9;
1811 #elif defined(__amd64)
1812         ehdr->e_ident[EI_DATA] = ELFDATA2LSB;
1813         ehdr->e_machine = EM_AMD64;
1814 #else
1815 #error "no recognized 64-bit machine type is defined"
1816 #endif

1818 #endif  /* !defined(_LP64) || defined(_ELF32_COMPAT) */

1820         /*
1821          * If the count of program headers or section headers or the index
1822          * of the section string table can't fit in the mere 16 bits
1823          * shortsightedly allotted to them in the ELF header, we use the
1824          * extended formats and put the real values in the section header
1825          * as index 0.
1826          */
1827         ehdr->e_version = EV_CURRENT;
1828         ehdr->e_ehsize = sizeof (Ehdr);

1830         if (nphdrs >= PN_XNUM)
1831                 ehdr->e_phnum = PN_XNUM;
1832         else
1833                 ehdr->e_phnum = (unsigned short)nphdrs;

1835         ehdr->e_phoff = sizeof (Ehdr);
1836         ehdr->e_phentsize = sizeof (Phdr);

1838         if (nshdrs > 0) {
1839                 if (nshdrs >= SHN_LORESERVE)
1840                         ehdr->e_shnum = 0;
1841                 else
1842                         ehdr->e_shnum = (unsigned short)nshdrs;

1844                 if (nshdrs - 1 >= SHN_LORESERVE)
1845                         ehdr->e_shstrndx = SHN_XINDEX;
1846                 else
1847                         ehdr->e_shstrndx = (unsigned short)(nshdrs - 1);

1849                 ehdr->e_shoff = ehdr->e_phoff + ehdr->e_phentsize * nphdrs;
1850                 ehdr->e_shentsize = sizeof (Shdr);
1851         }

1853         if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, ehdr,
1854             sizeof (Ehdr), rlimit, credp))
1855                 goto done;

1857         poffset = sizeof (Ehdr);
1858         soffset = sizeof (Ehdr) + phdrsz;
1859         doffset = sizeof (Ehdr) + phdrsz + shdrsz;

1861         v = &bigwad->phdr[0];
1862         bzero(v, phdrsz);

1864         setup_old_note_header(&v[0], p);
1865         v[0].p_offset = doffset = roundup(doffset, sizeof (Word));
1866         doffset += v[0].p_filesz;

1868         setup_note_header(&v[1], p);
1869         v[1].p_offset = doffset = roundup(doffset, sizeof (Word));
1870         doffset += v[1].p_filesz;

1872         mutex_enter(&p->p_lock);

1874         brkbase = p->p_brkbase;
1875         brksize = p->p_brksize;
```



```
1877         stkbase = p->p_usrstack - p->p_stksize;
1878         stksize = p->p_stksize;

1880         mutex_exit(&p->p_lock);

1882         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1883         i = 2;
1884         for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {
1885                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
1886                 caddr_t saddr, naddr;
1887                 void *tmp = NULL;
1888                 extern const struct seg_ops segspt_shmops;
1888                 extern struct seg_ops segspt_shmops;

1890                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
1891                         uint_t prot;
1892                         size_t size;
1893                         int type;
1894                         vnode_t *mvp;

1896                         prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
1897                         prot &= PROT_READ | PROT_WRITE | PROT_EXEC;
1898                         if ((size = (size_t)(naddr - saddr)) == 0)
1899                                 continue;
1900                         if (i == nphdrs) {
1901                                 overflow++;
1902                                 continue;
1903                         }
1904                         v[i].p_type = PT_LOAD;
1905                         v[i].p_vaddr = (Addr)(uintptr_t)saddr;
1906                         v[i].p_memsz = size;
1907                         if (prot & PROT_READ)
1908                                 v[i].p_flags |= PF_R;
1909                         if (prot & PROT_WRITE)
1910                                 v[i].p_flags |= PF_W;
1911                         if (prot & PROT_EXEC)
1912                                 v[i].p_flags |= PF_X;

1914                         /*
1915                          * Figure out which mappings to include in the core.
1916                          */
1917                         type = segop_gettype(seg, saddr);
1917                         type = SEGOP_GETTYPE(seg, saddr);

1919                         if (saddr == stkbase && size == stksize) {
1920                                 if (!(content & CC_CONTENT_STACK))
1921                                         goto exclude;

1923                         } else if (saddr == brkbase && size == brksize) {
1924                                 if (!(content & CC_CONTENT_HEAP))
1925                                         goto exclude;

1927                         } else if (seg->s_ops == &segspt_shmops) {
1928                                 if (type & MAP_NORESERVE) {
1929                                         if (!(content & CC_CONTENT_DISM))
1930                                                 goto exclude;
1931                                 } else {
1932                                         if (!(content & CC_CONTENT_ISM))
1933                                                 goto exclude;
1934                                 }

1936                         } else if (seg->s_ops != &segvn_ops) {
1937                                 goto exclude;

1939                         } else if (type & MAP_SHARED) {
```

```
1940                                if (shmgetid(p, saddr) != SHMID_NONE) {
1941                                        if (!(content & CC_CONTENT_SHM))
1942                                                goto exclude;

1944                                } else if (segop_getvp(seg, seg->s_base,
1944                                } else if (SEGOP_GETVP(seg, seg->s_base,
1945                                    &mvp) != 0 || mvp == NULL ||
1946                                    mvp->v_type != VREG) {
1947                                        if (!(content & CC_CONTENT_SHANON))
1948                                                goto exclude;

1950                                } else {
1951                                        if (!(content & CC_CONTENT_SHFILE))
1952                                                goto exclude;
1953                                }

1955                            } else if (segop_getvp(seg, seg->s_base, &mvp) != 0 ||
1955                            } else if (SEGOP_GETVP(seg, seg->s_base, &mvp) != 0 ||
1956                                mvp == NULL || mvp->v_type != VREG) {
1957                                    if (!(content & CC_CONTENT_ANON))
1958                                            goto exclude;

1960                            } else if (prot == (PROT_READ | PROT_EXEC)) {
1961                                    if (!(content & CC_CONTENT_TEXT))
1962                                            goto exclude;

1964                            } else if (prot == PROT_READ) {
1965                                    if (!(content & CC_CONTENT_RODATA))
1966                                            goto exclude;

1968                            } else {
1969                                    if (!(content & CC_CONTENT_DATA))
1970                                            goto exclude;
1971                            }

1973                            doffset = roundup(doffset, sizeof (Word));
1974                            v[i].p_offset = doffset;
1975                            v[i].p_filesz = size;
1976                            doffset += size;
1977 exclude:
1978                            i++;
1979                    }
1980                    ASSERT(tmp == NULL);
1981            }
1982            AS_LOCK_EXIT(as, &as->a_lock);

1984            if (overflow || i != nphdrs) {
1985                    if (ntries++ == 0) {
1986                            kmem_free(bigwad, bigsize);
1987                            overflow = 0;
1988                            goto top;
1989                    }
1990                    cmn_err(CE_WARN, "elfcore: core dump failed for "
1991                        "process %d; address space is changing", p->p_pid);
1992                    error = EIO;
1993                    goto done;
1994            }

1996            if ((error = core_write(vp, UIO_SYSSPACE, poffset,
1997                v, phdrsz, rlimit, credp)) != 0)
1998                    goto done;

2000            if ((error = write_old_elfnotes(p, sig, vp, v[0].p_offset, rlimit,
2001                credp)) != 0)
2002                    goto done;
```



```
2004            if ((error = write_elfnotes(p, sig, vp, v[1].p_offset, rlimit,
2005                credp, content)) != 0)
2006                    goto done;

2008            for (i = 2; i < nphdrs; i++) {
2009                    prkillinfo_t killinfo;
2010                    sigqueue_t *sq;
2011                    int sig, j;

2013                    if (v[i].p_filesz == 0)
2014                            continue;

2016                    /*
2017                     * If dumping out this segment fails, rather than failing
2018                     * the core dump entirely, we reset the size of the mapping
2019                     * to zero to indicate that the data is absent from the core
2020                     * file and or in the PF_SUNW_FAILURE flag to differentiate
2021                     * this from mappings that were excluded due to the core file
2022                     * content settings.
2023                     */
2024                    if ((error = core_seg(p, vp, v[i].p_offset,
2025                        (caddr_t)(uintptr_t)v[i].p_vaddr, v[i].p_filesz,
2026                        rlimit, credp)) == 0) {
2027                            continue;
2028                    }

2030                    if ((sig = lwp->lwp_cursig) == 0) {
2031                            /*
2032                             * We failed due to something other than a signal.
2033                             * Since the space reserved for the segment is now
2034                             * unused, we stash the errno in the first four
2035                             * bytes. This undocumented interface will let us
2036                             * understand the nature of the failure.
2037                             */
2038                            (void) core_write(vp, UIO_SYSSPACE, v[i].p_offset,
2039                                &error, sizeof (error), rlimit, credp);

2041                            v[i].p_filesz = 0;
2042                            v[i].p_flags |= PF_SUNW_FAILURE;
2043                            if ((error = core_write(vp, UIO_SYSSPACE,
2044                                poffset + sizeof (v[i]) * i, &v[i], sizeof (v[i]),
2045                                rlimit, credp)) != 0)
2046                                    goto done;

2048                            continue;
2049                    }

2051                    /*
2052                     * We took a signal.  We want to abort the dump entirely, but
2053                     * we also want to indicate what failed and why.  We therefore
2054                     * use the space reserved for the first failing segment to
2055                     * write our error (which, for purposes of compatability with
2056                     * older core dump readers, we set to EINTR) followed by any
2057                     * siginfo associated with the signal.
2058                     */
2059                    bzero(&killinfo, sizeof (killinfo));
2060                    killinfo.prk_error = EINTR;

2062                    sq = sig == SIGKILL ? curproc->p_killsqp : lwp->lwp_curinfo;

2064                    if (sq != NULL) {
2065                            bcopy(&sq->sq_info, &killinfo.prk_info,
2066                                sizeof (sq->sq_info));
2067                    } else {
2068                            killinfo.prk_info.si_signo = lwp->lwp_cursig;
2069                            killinfo.prk_info.si_code = SI_NOINFO;
```

```
2070                }

2072 #if (defined(_SYSCALL32_IMPL) || defined(_LP64))
2073                /*
2074                 * If this is a 32-bit process, we need to translate from the
2075                 * native siginfo to the 32-bit variant.  (Core readers must
2076                 * always have the same data model as their target or must
2077                 * be aware of -- and compensate for -- data model differences.)
2078                 */
2079                if (curproc->p_model == DATAMODEL_ILP32) {
2080                        siginfo32_t si32;

2082                        siginfo_kto32((k_siginfo_t *)&killinfo.prk_info, &si32);
2083                        bcopy(&si32, &killinfo.prk_info, sizeof (si32));
2084                }
2085 #endif

2087                (void) core_write(vp, UIO_SYSSPACE, v[i].p_offset,
2088                    &killinfo, sizeof (killinfo), rlimit, credp);

2090                /*
2091                 * For the segment on which we took the signal, indicate that
2092                 * its data now refers to a siginfo.
2093                 */
2094                v[i].p_filesz = 0;
2095                v[i].p_flags |= PF_SUNW_FAILURE | PF_SUNW_KILLED |
2096                    PF_SUNW_SIGINFO;

2098                /*
2099                 * And for every other segment, indicate that its absence
2100                 * is due to a signal.
2101                 */
2102                for (j = i + 1; j < nphdrs; j++) {
2103                        v[j].p_filesz = 0;
2104                        v[j].p_flags |= PF_SUNW_FAILURE | PF_SUNW_KILLED;
2105                }

2107                /*
2108                 * Finally, write out our modified program headers.
2109                 */
2110                if ((error = core_write(vp, UIO_SYSSPACE,
2111                    poffset + sizeof (v[i]) * i, &v[i],
2112                    sizeof (v[i]) * (nphdrs - i), rlimit, credp)) != 0)
2113                        goto done;

2115                break;
2116        }

2118        if (nshdrs > 0) {
2119                bzero(&bigwad->shdr[0], shdrsz);

2121                if (nshdrs >= SHN_LORESERVE)
2122                        bigwad->shdr[0].sh_size = nshdrs;

2124                if (nshdrs - 1 >= SHN_LORESERVE)
2125                        bigwad->shdr[0].sh_link = nshdrs - 1;

2127                if (nphdrs >= PN_XNUM)
2128                        bigwad->shdr[0].sh_info = nphdrs;

2130                if (nshdrs > 1) {
2131                        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
2132                        if ((error = process_scns(content, p, credp, vp,
2133                            &bigwad->shdr[0], nshdrs, rlimit, &doffset,
2134                            NULL)) != 0) {
2135                                AS_LOCK_EXIT(as, &as->a_lock);
```



```
2136                                goto done;
2137                        }
2138                        AS_LOCK_EXIT(as, &as->a_lock);
2139                }

2141                if ((error = core_write(vp, UIO_SYSSPACE, soffset,
2142                    &bigwad->shdr[0], shdrsz, rlimit, credp)) != 0)
2143                        goto done;
2144        }

2146 done:
2147        kmem_free(bigwad, bigsize);
2148        return (error);
2149 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  171811 Fri May  8 18:10:25 2015
new/usr/src/uts/common/fs/nfs/nfs3_vnops.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

5543 /*
5544  * Setup and add an address space callback to do the work of the delmap call.
5545  * The callback will (and must be) deleted in the actual callback function.
5546  *
5547  * This is done in order to take care of the problem that we have with holding
5548  * the address space's a_lock for a long period of time (e.g. if the NFS server
5549  * is down).  Callbacks will be executed in the address space code while the
5550  * a_lock is not held.  Holding the address space's a_lock causes things such
5551  * as ps and fork to hang because they are trying to acquire this lock as well.
5552  */
5553 /* ARGSUSED */
5554 static int
5555 nfs3_delmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
5556         size_t len, uint_t prot, uint_t maxprot, uint_t flags,
5557         cred_t *cr, caller_context_t *ct)
5558 {
5559         int                     caller_found;
5560         int                     error;
5561         rnode_t                 *rp;
5562         nfs_delmap_args_t       *dmapp;
5563         nfs_delmapcall_t        *delmap_call;

5565         if (vp->v_flag & VNOMAP)
5566                 return (ENOSYS);
5567         /*
5568          * A process may not change zones if it has NFS pages mmap'ed
5569          * in, so we can't legitimately get here from the wrong zone.
5570          */
5571         ASSERT(nfs_zone() == VTOMI(vp)->mi_zone);

5573         rp = VTOR(vp);

5575         /*
5576          * The way that the address space of this process deletes its mapping
5577          * of this file is via the following call chains:
5578          * - as_free()->segop_unmap()/segvn_unmap()->VOP_DELMAP()/nfs3_delmap()
5579          * - as_unmap()->segop_unmap()/segvn_unmap()->VOP_DELMAP()/nfs3_delmap()
5578          * - as_free()->SEGOP_UNMAP()/segvn_unmap()->VOP_DELMAP()/nfs3_delmap()
5579          * - as_unmap()->SEGOP_UNMAP()/segvn_unmap()->VOP_DELMAP()/nfs3_delmap()
5580          *
5581          * With the use of address space callbacks we are allowed to drop the
5582          * address space lock, a_lock, while executing the NFS operations that
5583          * need to go over the wire.  Returning EAGAIN to the caller of this
5584          * function is what drives the execution of the callback that we add
5585          * below.  The callback will be executed by the address space code
5586          * after dropping the a_lock.  When the callback is finished, since
5587          * we dropped the a_lock, it must be re-acquired and segvn_unmap()
5588          * is called again on the same segment to finish the rest of the work
5589          * that needs to happen during unmapping.
5590          *
5591          * This action of calling back into the segment driver causes
5592          * nfs3_delmap() to get called again, but since the callback was
5593          * already executed at this point, it already did the work and there
5594          * is nothing left for us to do.
5595          *
5596          * To Summarize:
5597          * - The first time nfs3_delmap is called by the current thread is when
5598          * we add the caller associated with this delmap to the delmap caller
5599          * list, add the callback, and return EAGAIN.
```



```
5600          * - The second time in this call chain when nfs3_delmap is called we
5601          * will find this caller in the delmap caller list and realize there
5602          * is no more work to do thus removing this caller from the list and
5603          * returning the error that was set in the callback execution.
5604          */
5605         caller_found = nfs_find_and_delete_delmapcall(rp, &error);
5606         if (caller_found) {
5607                 /*
5608                  * 'error' is from the actual delmap operations.  To avoid
5609                  * hangs, we need to handle the return of EAGAIN differently
5610                  * since this is what drives the callback execution.
5611                  * In this case, we don't want to return EAGAIN and do the
5612                  * callback execution because there are none to execute.
5613                  */
5614                 if (error == EAGAIN)
5615                         return (0);
5616                 else
5617                         return (error);
5618         }

5620         /* current caller was not in the list */
5621         delmap_call = nfs_init_delmapcall();

5623         mutex_enter(&rp->r_statelock);
5624         list_insert_tail(&rp->r_indelmap, delmap_call);
5625         mutex_exit(&rp->r_statelock);

5627         dmapp = kmem_alloc(sizeof (nfs_delmap_args_t), KM_SLEEP);

5629         dmapp->vp = vp;
5630         dmapp->off = off;
5631         dmapp->addr = addr;
5632         dmapp->len = len;
5633         dmapp->prot = prot;
5634         dmapp->maxprot = maxprot;
5635         dmapp->flags = flags;
5636         dmapp->cr = cr;
5637         dmapp->caller = delmap_call;

5639         error = as_add_callback(as, nfs3_delmap_callback, dmapp,
5640             AS_UNMAP_EVENT, addr, len, KM_SLEEP);

5642         return (error ? error : EAGAIN);
5643 }
______unchanged_portion_omitted_
```

****
429784 Fri May 8 18:10:25 2015
new/usr/src/uts/common/fs/nfs/nfs4_vnops.c
patch lower-case-segops
****
_____unchanged_portion_omitted_

```
11024 /*
11025  * Setup and add an address space callback to do the work of the delmap call.
11026  * The callback will (and must be) deleted in the actual callback function.
11027  *
11028  * This is done in order to take care of the problem that we have with holding
11029  * the address space's a_lock for a long period of time (e.g. if the NFS server
11030  * is down).  Callbacks will be executed in the address space code while the
11031  * a_lock is not held.  Holding the address space's a_lock causes things such
11032  * as ps and fork to hang because they are trying to acquire this lock as well.
11033  */
11034 /* ARGSUSED */
11035 static int
11036 nfs4_delmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
11037     size_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *cr,
11038     caller_context_t *ct)
11039 {
11040         int                     caller_found;
11041         int                     error;
11042         rnode4_t                *rp;
11043         nfs4_delmap_args_t      *dmapp;
11044         nfs4_delmapcall_t       *delmap_call;

11046         if (vp->v_flag & VNOMAP)
11047                 return (ENOSYS);

11049         /*
11050          * A process may not change zones if it has NFS pages mmap'ed
11051          * in, so we can't legitimately get here from the wrong zone.
11052          */
11053         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

11055         rp = VTOR4(vp);

11057         /*
11058          * The way that the address space of this process deletes its mapping
11059          * of this file is via the following call chains:
11060          * - as_free()->segop_unmap()/segvn_unmap()->VOP_DELMAP()/nfs4_delmap()
11061          * - as_unmap()->segop_unmap()/segvn_unmap()->VOP_DELMAP()/nfs4_delmap()
11060          * - as_free()->SEGOP_UNMAP()/segvn_unmap()->VOP_DELMAP()/nfs4_delmap()
11061          * - as_unmap()->SEGOP_UNMAP()/segvn_unmap()->VOP_DELMAP()/nfs4_delmap()
11062          *
11063          * With the use of address space callbacks we are allowed to drop the
11064          * address space lock, a_lock, while executing the NFS operations that
11065          * need to go over the wire.  Returning EAGAIN to the caller of this
11066          * function is what drives the execution of the callback that we add
11067          * below.  The callback will be executed by the address space code
11068          * after dropping the a_lock.  When the callback is finished, since
11069          * we dropped the a_lock, it must be re-acquired and segvn_unmap()
11070          * is called again on the same segment to finish the rest of the work
11071          * that needs to happen during unmapping.
11072          *
11073          * This action of calling back into the segment driver causes
11074          * nfs4_delmap() to get called again, but since the callback was
11075          * already executed at this point, it already did the work and there
11076          * is nothing left for us to do.
11077          *
11078          * To Summarize:
11079          * - The first time nfs4_delmap is called by the current thread is when
11080          * we add the caller associated with this delmap to the delmap caller
```



```
11081          * list, add the callback, and return EAGAIN.
11082          * - The second time in this call chain when nfs4_delmap is called we
11083          * will find this caller in the delmap caller list and realize there
11084          * is no more work to do thus removing this caller from the list and
11085          * returning the error that was set in the callback execution.
11086          */
11087         caller_found = nfs4_find_and_delete_delmapcall(rp, &error);
11088         if (caller_found) {
11089                 /*
11090                  * 'error' is from the actual delmap operations.  To avoid
11091                  * hangs, we need to handle the return of EAGAIN differently
11092                  * since this is what drives the callback execution.
11093                  * In this case, we don't want to return EAGAIN and do the
11094                  * callback execution because there are none to execute.
11095                  */
11096                 if (error == EAGAIN)
11097                         return (0);
11098                 else
11099                         return (error);
11100         }

11102         /* current caller was not in the list */
11103         delmap_call = nfs4_init_delmapcall();

11105         mutex_enter(&rp->r_statelock);
11106         list_insert_tail(&rp->r_indelmap, delmap_call);
11107         mutex_exit(&rp->r_statelock);

11109         dmapp = kmem_alloc(sizeof (nfs4_delmap_args_t), KM_SLEEP);

11111         dmapp->vp = vp;
11112         dmapp->off = off;
11113         dmapp->addr = addr;
11114         dmapp->len = len;
11115         dmapp->prot = prot;
11116         dmapp->maxprot = maxprot;
11117         dmapp->flags = flags;
11118         dmapp->cr = cr;
11119         dmapp->caller = delmap_call;

11121         error = as_add_callback(as, nfs4_delmap_callback, dmapp,
11122             AS_UNMAP_EVENT, addr, len, KM_SLEEP);

11124         return (error ? error : EAGAIN);
11125 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   67734 Fri May  8 18:10:26 2015
new/usr/src/uts/common/fs/nfs/nfs_srv.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

1151 static struct rfs_async_write_list *rfs_async_write_head = NULL;
1152 static kmutex_t rfs_async_write_lock;
1153 static int rfs_write_async = 1; /* enables write clustering if == 1 */

1155 #define MAXCLIOVECS     42
1156 #define RFSWRITE_INITVAL (enum nfsstat) -1

1158 #ifdef DEBUG
1159 static int rfs_write_hits = 0;
1160 static int rfs_write_misses = 0;
1161 #endif

1163 /*
1164  * Write data to file.
1165  * Returns attributes of a file after writing some data to it.
1166  */
1167 void
1168 rfs_write(struct nfswriteargs *wa, struct nfsattrstat *ns,
1169     struct exportinfo *exi, struct svc_req *req, cred_t *cr, bool_t ro)
1170 {
1171         int error;
1172         vnode_t *vp;
1173         rlim64_t rlimit;
1174         struct vattr va;
1175         struct uio uio;
1176         struct rfs_async_write_list *lp;
1177         struct rfs_async_write_list *nlp;
1178         struct rfs_async_write *rp;
1179         struct rfs_async_write *nrp;
1180         struct rfs_async_write *trp;
1181         struct rfs_async_write *lrp;
1182         int data_written;
1183         int iovcnt;
1184         mblk_t *m;
1185         struct iovec *iovp;
1186         struct iovec *niovp;
1187         struct iovec iov[MAXCLIOVECS];
1188         int count;
1189         int rcount;
1190         uint_t off;
1191         uint_t len;
1192         struct rfs_async_write nrpsp;
1193         struct rfs_async_write_list nlpsp;
1194         ushort_t t_flag;
1195         cred_t *savecred;
1196         int in_crit = 0;
1197         caller_context_t ct;

1199         if (!rfs_write_async) {
1200                 rfs_write_sync(wa, ns, exi, req, cr, ro);
1201                 return;
1202         }
```



```
1204         /*
1205          * Initialize status to RFSWRITE_INITVAL instead of 0, since value of 0
1206          * is considered an OK.
1207          */
1208         ns->ns_status = RFSWRITE_INITVAL;

1210         nrp = &nrpsp;
1211         nrp->wa = wa;
1212         nrp->ns = ns;
1213         nrp->req = req;
1214         nrp->cr = cr;
1215         nrp->ro = ro;
1216         nrp->thread = curthread;

1218         ASSERT(curthread->t_schedflag & TS_DONT_SWAP);

1218         /*
1219          * Look to see if there is already a cluster started
1220          * for this file.
1221          */
1222         mutex_enter(&rfs_async_write_lock);
1223         for (lp = rfs_async_write_head; lp != NULL; lp = lp->next) {
1224                 if (bcmp(&wa->wa_fhandle, lp->fhp,
1225                     sizeof (fhandle_t)) == 0)
1226                         break;
1227         }

1229         /*
1230          * If lp is non-NULL, then there is already a cluster
1231          * started.  We need to place ourselves in the cluster
1232          * list in the right place as determined by starting
1233          * offset.  Conflicts with non-blocking mandatory locked
1234          * regions will be checked when the cluster is processed.
1235          */
1236         if (lp != NULL) {
1237                 rp = lp->list;
1238                 trp = NULL;
1239                 while (rp != NULL && rp->wa->wa_offset < wa->wa_offset) {
1240                         trp = rp;
1241                         rp = rp->list;
1242                 }
1243                 nrp->list = rp;
1244                 if (trp == NULL)
1245                         lp->list = nrp;
1246                 else
1247                         trp->list = nrp;
1248                 while (nrp->ns->ns_status == RFSWRITE_INITVAL)
1249                         cv_wait(&lp->cv, &rfs_async_write_lock);
1250                 mutex_exit(&rfs_async_write_lock);

1252                 return;
1253         }

1255         /*
1256          * No cluster started yet, start one and add ourselves
1257          * to the list of clusters.
1258          */
1259         nrp->list = NULL;

1261         nlp = &nlpsp;
1262         nlp->fhp = &wa->wa_fhandle;
1263         cv_init(&nlp->cv, NULL, CV_DEFAULT, NULL);
1264         nlp->list = nrp;
1265         nlp->next = NULL;

1267         if (rfs_async_write_head == NULL) {
```

```
1268                    rfs_async_write_head = nlp;
1269            } else {
1270                    lp = rfs_async_write_head;
1271                    while (lp->next != NULL)
1272                            lp = lp->next;
1273                    lp->next = nlp;
1274            }
1275            mutex_exit(&rfs_async_write_lock);

1277            /*
1278             * Convert the file handle common to all of the requests
1279             * in this cluster to a vnode.
1280             */
1281            vp = nfs_fhtovp(&wa->wa_fhandle, exi);
1282            if (vp == NULL) {
1283                    mutex_enter(&rfs_async_write_lock);
1284                    if (rfs_async_write_head == nlp)
1285                            rfs_async_write_head = nlp->next;
1286                    else {
1287                            lp = rfs_async_write_head;
1288                            while (lp->next != nlp)
1289                                    lp = lp->next;
1290                            lp->next = nlp->next;
1291                    }
1292                    t_flag = curthread->t_flag & T_WOULDBLOCK;
1293                    for (rp = nlp->list; rp != NULL; rp = rp->list) {
1294                            rp->ns->ns_status = NFSERR_STALE;
1295                            rp->thread->t_flag |= t_flag;
1296                    }
1297                    cv_broadcast(&nlp->cv);
1298                    mutex_exit(&rfs_async_write_lock);

1300                    return;
1301            }

1303            /*
1304             * Can only write regular files.  Attempts to write any
1305             * other file types fail with EISDIR.
1306             */
1307            if (vp->v_type != VREG) {
1308                    VN_RELE(vp);
1309                    mutex_enter(&rfs_async_write_lock);
1310                    if (rfs_async_write_head == nlp)
1311                            rfs_async_write_head = nlp->next;
1312                    else {
1313                            lp = rfs_async_write_head;
1314                            while (lp->next != nlp)
1315                                    lp = lp->next;
1316                            lp->next = nlp->next;
1317                    }
1318                    t_flag = curthread->t_flag & T_WOULDBLOCK;
1319                    for (rp = nlp->list; rp != NULL; rp = rp->list) {
1320                            rp->ns->ns_status = NFSERR_ISDIR;
1321                            rp->thread->t_flag |= t_flag;
1322                    }
1323                    cv_broadcast(&nlp->cv);
1324                    mutex_exit(&rfs_async_write_lock);

1326                    return;
1327            }

1329            /*
1330             * Enter the critical region before calling VOP_RWLOCK, to avoid a
1331             * deadlock with ufs.
1332             */
1333            if (nbl_need_check(vp)) {
```



```
1334                    nbl_start_crit(vp, RW_READER);
1335                    in_crit = 1;
1336            }

1338            ct.cc_sysid = 0;
1339            ct.cc_pid = 0;
1340            ct.cc_caller_id = nfs2_srv_caller_id;
1341            ct.cc_flags = CC_DONTBLOCK;

1343            /*
1344             * Lock the file for writing.  This operation provides
1345             * the delay which allows clusters to grow.
1346             */
1347            error = VOP_RWLOCK(vp, V_WRITELOCK_TRUE, &ct);

1349            /* check if a monitor detected a delegation conflict */
1350            if (error == EAGAIN && (ct.cc_flags & CC_WOULDBLOCK)) {
1351                    if (in_crit)
1352                            nbl_end_crit(vp);
1353                    VN_RELE(vp);
1354                    /* mark as wouldblock so response is dropped */
1355                    curthread->t_flag |= T_WOULDBLOCK;
1356                    mutex_enter(&rfs_async_write_lock);
1357                    if (rfs_async_write_head == nlp)
1358                            rfs_async_write_head = nlp->next;
1359                    else {
1360                            lp = rfs_async_write_head;
1361                            while (lp->next != nlp)
1362                                    lp = lp->next;
1363                            lp->next = nlp->next;
1364                    }
1365                    for (rp = nlp->list; rp != NULL; rp = rp->list) {
1366                            if (rp->ns->ns_status == RFSWRITE_INITVAL) {
1367                                    rp->ns->ns_status = puterrno(error);
1368                                    rp->thread->t_flag |= T_WOULDBLOCK;
1369                            }
1370                    }
1371                    cv_broadcast(&nlp->cv);
1372                    mutex_exit(&rfs_async_write_lock);

1374                    return;
1375            }

1377            /*
1378             * Disconnect this cluster from the list of clusters.
1379             * The cluster that is being dealt with must be fixed
1380             * in size after this point, so there is no reason
1381             * to leave it on the list so that new requests can
1382             * find it.
1383             *
1384             * The algorithm is that the first write request will
1385             * create a cluster, convert the file handle to a
1386             * vnode pointer, and then lock the file for writing.
1387             * This request is not likely to be clustered with
1388             * any others.  However, the next request will create
1389             * a new cluster and be blocked in VOP_RWLOCK while
1390             * the first request is being processed.  This delay
1391             * will allow more requests to be clustered in this
1392             * second cluster.
1393             */
1394            mutex_enter(&rfs_async_write_lock);
1395            if (rfs_async_write_head == nlp)
1396                    rfs_async_write_head = nlp->next;
1397            else {
1398                    lp = rfs_async_write_head;
1399                    while (lp->next != nlp)
```

```
1400                        lp = lp->next;
1401                lp->next = nlp->next;
1402        }
1403        mutex_exit(&rfs_async_write_lock);

1405        /*
1406         * Step through the list of requests in this cluster.
1407         * We need to check permissions to make sure that all
1408         * of the requests have sufficient permission to write
1409         * the file.  A cluster can be composed of requests
1410         * from different clients and different users on each
1411         * client.
1412         *
1413         * As a side effect, we also calculate the size of the
1414         * byte range that this cluster encompasses.
1415         */
1416        rp = nlp->list;
1417        off = rp->wa->wa_offset;
1418        len = (uint_t)0;
1419        do {
1420                if (rdonly(rp->ro, vp)) {
1421                        rp->ns->ns_status = NFSERR_ROFS;
1422                        t_flag = curthread->t_flag & T_WOULDBLOCK;
1423                        rp->thread->t_flag |= t_flag;
1424                        continue;
1425                }

1427                va.va_mask = AT_UID|AT_MODE;

1429                error = VOP_GETATTR(vp, &va, 0, rp->cr, &ct);

1431                if (!error) {
1432                        if (crgetuid(rp->cr) != va.va_uid) {
1433                                /*
1434                                 * This is a kludge to allow writes of files
1435                                 * created with read only permission.  The
1436                                 * owner of the file is always allowed to
1437                                 * write it.
1438                                 */
1439                                error = VOP_ACCESS(vp, VWRITE, 0, rp->cr, &ct);
1440                        }
1441                        if (!error && MANDLOCK(vp, va.va_mode))
1442                                error = EACCES;
1443                }

1445                /*
1446                 * Check for a conflict with a nbmand-locked region.
1447                 */
1448                if (in_crit && nbl_conflict(vp, NBL_WRITE, rp->wa->wa_offset,
1449                    rp->wa->wa_count, 0, NULL)) {
1450                        error = EACCES;
1451                }

1453                if (error) {
1454                        rp->ns->ns_status = puterrno(error);
1455                        t_flag = curthread->t_flag & T_WOULDBLOCK;
1456                        rp->thread->t_flag |= t_flag;
1457                        continue;
1458                }
1459                if (len < rp->wa->wa_offset + rp->wa->wa_count - off)
1460                        len = rp->wa->wa_offset + rp->wa->wa_count - off;
1461        } while ((rp = rp->list) != NULL);

1463        /*
1464         * Step through the cluster attempting to gather as many
1465         * requests which are contiguous as possible.  These
```



```
1466         * contiguous requests are handled via one call to VOP_WRITE
1467         * instead of different calls to VOP_WRITE.  We also keep
1468         * track of the fact that any data was written.
1469         */
1470        rp = nlp->list;
1471        data_written = 0;
1472        do {
1473                /*
1474                 * Skip any requests which are already marked as having an
1475                 * error.
1476                 */
1477                if (rp->ns->ns_status != RFSWRITE_INITVAL) {
1478                        rp = rp->list;
1479                        continue;
1480                }

1482                /*
1483                 * Count the number of iovec's which are required
1484                 * to handle this set of requests.  One iovec is
1485                 * needed for each data buffer, whether addressed
1486                 * by wa_data or by the b_rptr pointers in the
1487                 * mblk chains.
1488                 */
1489                iovcnt = 0;
1490                lrp = rp;
1491                for (;;) {
1492                        if (lrp->wa->wa_data || lrp->wa->wa_rlist)
1493                                iovcnt++;
1494                        else {
1495                                m = lrp->wa->wa_mblk;
1496                                while (m != NULL) {
1497                                        iovcnt++;
1498                                        m = m->b_cont;
1499                                }
1500                        }
1501                        if (lrp->list == NULL ||
1502                            lrp->list->ns->ns_status != RFSWRITE_INITVAL ||
1503                            lrp->wa->wa_offset + lrp->wa->wa_count !=
1504                            lrp->list->wa->wa_offset) {
1505                                lrp = lrp->list;
1506                                break;
1507                        }
1508                        lrp = lrp->list;
1509                }

1511                if (iovcnt <= MAXCLIOVECS) {
1512 #ifdef DEBUG
1513                        rfs_write_hits++;
1514 #endif
1515                        niovp = iov;
1516                } else {
1517 #ifdef DEBUG
1518                        rfs_write_misses++;
1519 #endif
1520                        niovp = kmem_alloc(sizeof (*niovp) * iovcnt, KM_SLEEP);
1521                }
1522                /*
1523                 * Put together the scatter/gather iovecs.
1524                 */
1525                iovp = niovp;
1526                trp = rp;
1527                count = 0;
1528                do {
1529                        if (trp->wa->wa_data || trp->wa->wa_rlist) {
1530                                if (trp->wa->wa_rlist) {
1531                                        iovp->iov_base =
```

```
1532                                        (char *)((trp->wa->wa_rlist)->
1533                                        u.c_daddr3);
1534                                iovp->iov_len = trp->wa->wa_count;
1535                        } else  {
1536                                iovp->iov_base = trp->wa->wa_data;
1537                                iovp->iov_len = trp->wa->wa_count;
1538                        }
1539                        iovp++;
1540                } else {
1541                        m = trp->wa->wa_mblk;
1542                        rcount = trp->wa->wa_count;
1543                        while (m != NULL) {
1544                                iovp->iov_base = (caddr_t)m->b_rptr;
1545                                iovp->iov_len = (m->b_wptr - m->b_rptr);
1546                                rcount -= iovp->iov_len;
1547                                if (rcount < 0)
1548                                        iovp->iov_len += rcount;
1549                                iovp++;
1550                                if (rcount <= 0)
1551                                        break;
1552                                m = m->b_cont;
1553                        }
1554                }
1555                count += trp->wa->wa_count;
1556                trp = trp->list;
1557        } while (trp != lrp);

1559        uio.uio_iov = niovp;
1560        uio.uio_iovcnt = iovcnt;
1561        uio.uio_segflg = UIO_SYSSPACE;
1562        uio.uio_extflg = UIO_COPY_DEFAULT;
1563        uio.uio_loffset = (offset_t)rp->wa->wa_offset;
1564        uio.uio_resid = count;
1565        /*
1566         * The limit is checked on the client. We
1567         * should allow any size writes here.
1568         */
1569        uio.uio_llimit = curproc->p_fsz_ctl;
1570        rlimit = uio.uio_llimit - rp->wa->wa_offset;
1571        if (rlimit < (rlim64_t)uio.uio_resid)
1572                uio.uio_resid = (uint_t)rlimit;

1574        /*
1575         * For now we assume no append mode.
1576         */

1578        /*
1579         * We're changing creds because VM may fault
1580         * and we need the cred of the current
1581         * thread to be used if quota * checking is
1582         * enabled.
1583         */
1584        savecred = curthread->t_cred;
1585        curthread->t_cred = cr;
1586        error = VOP_WRITE(vp, &uio, 0, rp->cr, &ct);
1587        curthread->t_cred = savecred;

1589        /* check if a monitor detected a delegation conflict */
1590        if (error == EAGAIN && (ct.cc_flags & CC_WOULDBLOCK))
1591                /* mark as wouldblock so response is dropped */
1592                curthread->t_flag |= T_WOULDBLOCK;

1594        if (niovp != iov)
1595                kmem_free(niovp, sizeof (*niovp) * iovcnt);

1597        if (!error) {
```



```
1598                        data_written = 1;
1599                        /*
1600                         * Get attributes again so we send the latest mod
1601                         * time to the client side for his cache.
1602                         */
1603                        va.va_mask = AT_ALL;    /* now we want everything */

1605                        error = VOP_GETATTR(vp, &va, 0, rp->cr, &ct);

1607                        if (!error)
1608                                acl_perm(vp, exi, &va, rp->cr);
1609                }

1611                /*
1612                 * Fill in the status responses for each request
1613                 * which was just handled.  Also, copy the latest
1614                 * attributes in to the attribute responses if
1615                 * appropriate.
1616                 */
1617                t_flag = curthread->t_flag & T_WOULDBLOCK;
1618                do {
1619                        rp->thread->t_flag |= t_flag;
1620                        /* check for overflows */
1621                        if (!error) {
1622                                error  = vattr_to_nattr(&va, &rp->ns->ns_attr);
1623                        }
1624                        rp->ns->ns_status = puterrno(error);
1625                        rp = rp->list;
1626                } while (rp != lrp);
1627        } while (rp != NULL);

1629        /*
1630         * If any data was written at all, then we need to flush
1631         * the data and metadata to stable storage.
1632         */
1633        if (data_written) {
1634                error = VOP_PUTPAGE(vp, (u_offset_t)off, len, 0, cr, &ct);

1636                if (!error) {
1637                        error = VOP_FSYNC(vp, FNODSYNC, cr, &ct);
1638                }
1639        }

1641        VOP_RWUNLOCK(vp, V_WRITELOCK_TRUE, &ct);

1643        if (in_crit)
1644                nbl_end_crit(vp);
1645        VN_RELE(vp);

1647        t_flag = curthread->t_flag & T_WOULDBLOCK;
1648        mutex_enter(&rfs_async_write_lock);
1649        for (rp = nlp->list; rp != NULL; rp = rp->list) {
1650                if (rp->ns->ns_status == RFSWRITE_INITVAL) {
1651                        rp->ns->ns_status = puterrno(error);
1652                        rp->thread->t_flag |= t_flag;
1653                }
1654        }
1655        cv_broadcast(&nlp->cv);
1656        mutex_exit(&rfs_async_write_lock);

1658 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  130899 Fri May  8 18:10:26 2015
new/usr/src/uts/common/fs/nfs/nfs_vnops.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

4640 /*
4641  * Setup and add an address space callback to do the work of the delmap call.
4642  * The callback will (and must be) deleted in the actual callback function.
4643  *
4644  * This is done in order to take care of the problem that we have with holding
4645  * the address space's a_lock for a long period of time (e.g. if the NFS server
4646  * is down).  Callbacks will be executed in the address space code while the
4647  * a_lock is not held.  Holding the address space's a_lock causes things such
4648  * as ps and fork to hang because they are trying to acquire this lock as well.
4649  */
4650 /* ARGSUSED */
4651 static int
4652 nfs_delmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
4653         size_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *cr,
4654         caller_context_t *ct)
4655 {
4656         int                     caller_found;
4657         int                     error;
4658         rnode_t                 *rp;
4659         nfs_delmap_args_t       *dmapp;
4660         nfs_delmapcall_t        *delmap_call;

4662         if (vp->v_flag & VNOMAP)
4663                 return (ENOSYS);
4664         /*
4665          * A process may not change zones if it has NFS pages mmap'ed
4666          * in, so we can't legitimately get here from the wrong zone.
4667          */
4668         ASSERT(nfs_zone() == VTOMI(vp)->mi_zone);

4670         rp = VTOR(vp);

4672         /*
4673          * The way that the address space of this process deletes its mapping
4674          * of this file is via the following call chains:
4675          * - as_free()->segop_unmap()/segvn_unmap()->VOP_DELMAP()/nfs_delmap()
4676          * - as_unmap()->segop_unmap()/segvn_unmap()->VOP_DELMAP()/nfs_delmap()
4675          * - as_free()->SEGOP_UNMAP()/segvn_unmap()->VOP_DELMAP()/nfs_delmap()
4676          * - as_unmap()->SEGOP_UNMAP()/segvn_unmap()->VOP_DELMAP()/nfs_delmap()
4677          *
4678          * With the use of address space callbacks we are allowed to drop the
4679          * address space lock, a_lock, while executing the NFS operations that
4680          * need to go over the wire.  Returning EAGAIN to the caller of this
4681          * function is what drives the execution of the callback that we add
4682          * below.  The callback will be executed by the address space code
4683          * after dropping the a_lock.  When the callback is finished, since
4684          * we dropped the a_lock, it must be re-acquired and segvn_unmap()
4685          * is called again on the same segment to finish the rest of the work
4686          * that needs to happen during unmapping.
4687          *
4688          * This action of calling back into the segment driver causes
4689          * nfs_delmap() to get called again, but since the callback was
4690          * already executed at this point, it already did the work and there
4691          * is nothing left for us to do.
4692          *
4693          * To Summarize:
4694          * - The first time nfs_delmap is called by the current thread is when
4695          * we add the caller associated with this delmap to the delmap caller
4696          * list, add the callback, and return EAGAIN.
```



```
4697          * - The second time in this call chain when nfs_delmap is called we
4698          * will find this caller in the delmap caller list and realize there
4699          * is no more work to do thus removing this caller from the list and
4700          * returning the error that was set in the callback execution.
4701          */
4702         caller_found = nfs_find_and_delete_delmapcall(rp, &error);
4703         if (caller_found) {
4704                 /*
4705                  * 'error' is from the actual delmap operations.  To avoid
4706                  * hangs, we need to handle the return of EAGAIN differently
4707                  * since this is what drives the callback execution.
4708                  * In this case, we don't want to return EAGAIN and do the
4709                  * callback execution because there are none to execute.
4710                  */
4711                 if (error == EAGAIN)
4712                         return (0);
4713                 else
4714                         return (error);
4715         }

4717         /* current caller was not in the list */
4718         delmap_call = nfs_init_delmapcall();

4720         mutex_enter(&rp->r_statelock);
4721         list_insert_tail(&rp->r_indelmap, delmap_call);
4722         mutex_exit(&rp->r_statelock);

4724         dmapp = kmem_alloc(sizeof (nfs_delmap_args_t), KM_SLEEP);

4726         dmapp->vp = vp;
4727         dmapp->off = off;
4728         dmapp->addr = addr;
4729         dmapp->len = len;
4730         dmapp->prot = prot;
4731         dmapp->maxprot = maxprot;
4732         dmapp->flags = flags;
4733         dmapp->cr = cr;
4734         dmapp->caller = delmap_call;

4736         error = as_add_callback(as, nfs_delmap_callback, dmapp,
4737             AS_UNMAP_EVENT, addr, len, KM_SLEEP);

4739         return (error ? error : EAGAIN);
4740 }
______unchanged_portion_omitted_
```

```
**********************************************************
   93906 Fri May  8 18:10:26 2015
new/usr/src/uts/common/fs/proc/prioctl.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

3117 /*
3118  * Common code for PIOCOPENM
3119  * Returns with the process unlocked.
3120  */
3121 static int
3122 propenm(prnode_t *pnp, caddr_t cmaddr, caddr_t va, int *rvalp, cred_t *cr)
3123 {
3124         proc_t *p = pnp->pr_common->prc_proc;
3125         struct as *as = p->p_as;
3126         int error = 0;
3127         struct seg *seg;
3128         struct vnode *xvp;
3129         int n;

3131         /*
3132          * By fiat, a system process has no address space.
3133          */
3134         if ((p->p_flag & SSYS) || as == &kas) {
3135                 error = EINVAL;
3136         } else if (cmaddr) {
3137                 /*
3138                  * We drop p_lock before grabbing the address
3139                  * space lock in order to avoid a deadlock with
3140                  * the clock thread.  The process will not
3141                  * disappear and its address space will not
3142                  * change because it is marked P_PR_LOCK.
3143                  */
3144                 mutex_exit(&p->p_lock);
3145                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
3146                 seg = as_segat(as, va);
3147                 if (seg != NULL &&
3148                     seg->s_ops == &segvn_ops &&
3149                     segop_getvp(seg, va, &xvp) == 0 &&
3149                     SEGOP_GETVP(seg, va, &xvp) == 0 &&
3150                     xvp != NULL &&
3151                     xvp->v_type == VREG) {
3152                         VN_HOLD(xvp);
3153                 } else {
3154                         error = EINVAL;
3155                 }
3156                 AS_LOCK_EXIT(as, &as->a_lock);
3157                 mutex_enter(&p->p_lock);
3158         } else if ((xvp = p->p_exec) == NULL) {
3159                 error = EINVAL;
3160         } else {
3161                 VN_HOLD(xvp);
3162         }

3164         prunlock(pnp);

3166         if (error == 0) {
3167                 if ((error = VOP_ACCESS(xvp, VREAD, 0, cr, NULL)) == 0)
3168                         error = fassign(&xvp, FREAD, &n);
3169                 if (error) {
3170                         VN_RELE(xvp);
3171                 } else {
3172                         *rvalp = n;
3173                 }
3174         }
```



```
3176         return (error);
3177 }
______unchanged_portion_omitted_

3511 /*
3512  * Return an array of structures with memory map information.
3513  * We allocate here; the caller must deallocate.
3514  * The caller is also responsible to append the zero-filled entry
3515  * that terminates the PIOCMAP output buffer.
3516  */
3517 static int
3518 oprgetmap(proc_t *p, list_t *iolhead)
3519 {
3520         struct as *as = p->p_as;
3521         prmap_t *mp;
3522         struct seg *seg;
3523         struct seg *brkseg, *stkseg;
3524         uint_t prot;

3526         ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

3528         /*
3529          * Request an initial buffer size that doesn't waste memory
3530          * if the address space has only a small number of segments.
3531          */
3532         pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

3534         if ((seg = AS_SEGFIRST(as)) == NULL)
3535                 return (0);

3537         brkseg = break_seg(p);
3538         stkseg = as_segat(as, prgetstackbase(p));

3540         do {
3541                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3542                 caddr_t saddr, naddr;
3543                 void *tmp = NULL;

3545                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3546                         prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3547                         if (saddr == naddr)
3548                                 continue;

3550                         mp = pr_iol_newbuf(iolhead, sizeof (*mp));

3552                         mp->pr_vaddr = saddr;
3553                         mp->pr_size = naddr - saddr;
3554                         mp->pr_off = segop_getoffset(seg, saddr);
3554                         mp->pr_off = SEGOP_GETOFFSET(seg, saddr);
3555                         mp->pr_mflags = 0;
3556                         if (prot & PROT_READ)
3557                                 mp->pr_mflags |= MA_READ;
3558                         if (prot & PROT_WRITE)
3559                                 mp->pr_mflags |= MA_WRITE;
3560                         if (prot & PROT_EXEC)
3561                                 mp->pr_mflags |= MA_EXEC;
3562                         if (segop_gettype(seg, saddr) & MAP_SHARED)
3562                         if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
3563                                 mp->pr_mflags |= MA_SHARED;
3564                         if (seg == brkseg)
3565                                 mp->pr_mflags |= MA_BREAK;
3566                         else if (seg == stkseg)
3567                                 mp->pr_mflags |= MA_STACK;
3568                         mp->pr_pagesize = PAGESIZE;
3569                 }
```

```
3570                 ASSERT(tmp == NULL);
3571         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

3573         return (0);
3574 }

3576 #ifdef _SYSCALL32_IMPL
3577 static int
3578 oprgetmap32(proc_t *p, list_t *iolhead)
3579 {
3580         struct as *as = p->p_as;
3581         ioc_prmap32_t *mp;
3582         struct seg *seg;
3583         struct seg *brkseg, *stkseg;
3584         uint_t prot;

3586         ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

3588         /*
3589          * Request an initial buffer size that doesn't waste memory
3590          * if the address space has only a small number of segments.
3591          */
3592         pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

3594         if ((seg = AS_SEGFIRST(as)) == NULL)
3595                 return (0);

3597         brkseg = break_seg(p);
3598         stkseg = as_segat(as, prgetstackbase(p));

3600         do {
3601                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3602                 caddr_t saddr, naddr;
3603                 void *tmp = NULL;

3605                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3606                         prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3607                         if (saddr == naddr)
3608                                 continue;

3610                         mp = pr_iol_newbuf(iolhead, sizeof (*mp));

3612                         mp->pr_vaddr = (caddr32_t)(uintptr_t)saddr;
3613                         mp->pr_size = (size32_t)(naddr - saddr);
3614                         mp->pr_off = (off32_t)segop_getoffset(seg, saddr);
3614                         mp->pr_off = (off32_t)SEGOP_GETOFFSET(seg, saddr);
3615                         mp->pr_mflags = 0;
3616                         if (prot & PROT_READ)
3617                                 mp->pr_mflags |= MA_READ;
3618                         if (prot & PROT_WRITE)
3619                                 mp->pr_mflags |= MA_WRITE;
3620                         if (prot & PROT_EXEC)
3621                                 mp->pr_mflags |= MA_EXEC;
3622                         if (segop_gettype(seg, saddr) & MAP_SHARED)
3622                         if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
3623                                 mp->pr_mflags |= MA_SHARED;
3624                         if (seg == brkseg)
3625                                 mp->pr_mflags |= MA_BREAK;
3626                         else if (seg == stkseg)
3627                                 mp->pr_mflags |= MA_STACK;
3628                         mp->pr_pagesize = PAGESIZE;
3629                 }
3630                 ASSERT(tmp == NULL);
3631         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

3633         return (0);
```



```
3634 }
_____unchanged_portion_omitted_
3698 #endif  /* _SYSCALL32_IMPL */

3700 /*
3701  * Read old /proc page data information.
3702  */
3703 int
3704 oprpdread(struct as *as, uint_t hatid, struct uio *uiop)
3705 {
3706         caddr_t buf;
3707         size_t size;
3708         prpageheader_t *php;
3709         prasmap_t *pmp;
3710         struct seg *seg;
3711         int error;

3713 again:
3714         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

3716         if ((seg = AS_SEGFIRST(as)) == NULL) {
3717                 AS_LOCK_EXIT(as, &as->a_lock);
3718                 return (0);
3719         }
3720         size = oprpdsize(as);
3721         if (uiop->uio_resid < size) {
3722                 AS_LOCK_EXIT(as, &as->a_lock);
3723                 return (E2BIG);
3724         }

3726         buf = kmem_zalloc(size, KM_SLEEP);
3727         php = (prpageheader_t *)buf;
3728         pmp = (prasmap_t *)(buf + sizeof (prpageheader_t));

3730         hrt2ts(gethrtime(), &php->pr_tstamp);
3731         php->pr_nmap = 0;
3732         php->pr_npage = 0;
3733         do {
3734                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3735                 caddr_t saddr, naddr;
3736                 void *tmp = NULL;

3738                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3739                         size_t len;
3740                         size_t npage;
3741                         uint_t prot;
3742                         uintptr_t next;

3744                         prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3745                         if ((len = naddr - saddr) == 0)
3746                                 continue;
3747                         npage = len / PAGESIZE;
3748                         next = (uintptr_t)(pmp + 1) + roundlong(npage);
3749                         /*
3750                          * It's possible that the address space can change
3751                          * subtlely even though we're holding as->a_lock
3752                          * due to the nondeterminism of page_exists() in
3753                          * the presence of asychronously flushed pages or
3754                          * mapped files whose sizes are changing.
3755                          * page_exists() may be called indirectly from
3756                          * pr_getprot() by a segop_incore() routine.
3756                          * pr_getprot() by a SEGOP_INCORE() routine.
3757                          * If this happens we need to make sure we don't
3758                          * overrun the buffer whose size we computed based
3759                          * on the initial iteration through the segments.
3760                          * Once we've detected an overflow, we need to clean
```

```
3761                          * up the temporary memory allocated in pr_getprot()
3762                          * and retry. If there's a pending signal, we return
3763                          * EINTR so that this thread can be dislodged if
3764                          * a latent bug causes us to spin indefinitely.
3765                          */
3766                         if (next > (uintptr_t)buf + size) {
3767                                 pr_getprot_done(&tmp);
3768                                 AS_LOCK_EXIT(as, &as->a_lock);

3770                                 kmem_free(buf, size);

3772                                 if (ISSIG(curthread, JUSTLOOKING))
3773                                         return (EINTR);

3775                                 goto again;
3776                         }

3778                         php->pr_nmap++;
3779                         php->pr_npage += npage;
3780                         pmp->pr_vaddr = saddr;
3781                         pmp->pr_npage = npage;
3782                         pmp->pr_off = segop_getoffset(seg, saddr);
3782                         pmp->pr_off = SEGOP_GETOFFSET(seg, saddr);
3783                         pmp->pr_mflags = 0;
3784                         if (prot & PROT_READ)
3785                                 pmp->pr_mflags |= MA_READ;
3786                         if (prot & PROT_WRITE)
3787                                 pmp->pr_mflags |= MA_WRITE;
3788                         if (prot & PROT_EXEC)
3789                                 pmp->pr_mflags |= MA_EXEC;
3790                         if (segop_gettype(seg, saddr) & MAP_SHARED)
3790                         if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
3791                                 pmp->pr_mflags |= MA_SHARED;
3792                         pmp->pr_pagesize = PAGESIZE;
3793                         hat_getstat(as, saddr, len, hatid,
3794                             (char *)(pmp + 1), HAT_SYNC_ZERORM);
3795                         pmp = (prasmap_t *)next;
3796                 }
3797                 ASSERT(tmp == NULL);
3798         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

3800         AS_LOCK_EXIT(as, &as->a_lock);

3802         ASSERT((uintptr_t)pmp <= (uintptr_t)buf + size);
3803         error = uiomove(buf, (caddr_t)pmp - buf, UIO_READ, uiop);
3804         kmem_free(buf, size);

3806         return (error);
3807 }

3809 #ifdef _SYSCALL32_IMPL
3810 int
3811 oprpdread32(struct as *as, uint_t hatid, struct uio *uiop)
3812 {
3813         caddr_t buf;
3814         size_t size;
3815         ioc_prpageheader32_t *php;
3816         ioc_prasmap32_t *pmp;
3817         struct seg *seg;
3818         int error;

3820 again:
3821         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

3823         if ((seg = AS_SEGFIRST(as)) == NULL) {
3824                 AS_LOCK_EXIT(as, &as->a_lock);
```



```
3825                 return (0);
3826         }
3827         size = oprpdsize32(as);
3828         if (uiop->uio_resid < size) {
3829                 AS_LOCK_EXIT(as, &as->a_lock);
3830                 return (E2BIG);
3831         }

3833         buf = kmem_zalloc(size, KM_SLEEP);
3834         php = (ioc_prpageheader32_t *)buf;
3835         pmp = (ioc_prasmap32_t *)(buf + sizeof (ioc_prpageheader32_t));

3837         hrt2ts32(gethrtime(), &php->pr_tstamp);
3838         php->pr_nmap = 0;
3839         php->pr_npage = 0;
3840         do {
3841                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3842                 caddr_t saddr, naddr;
3843                 void *tmp = NULL;

3845                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3846                         size_t len;
3847                         size_t npage;
3848                         uint_t prot;
3849                         uintptr_t next;

3851                         prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3852                         if ((len = naddr - saddr) == 0)
3853                                 continue;
3854                         npage = len / PAGESIZE;
3855                         next = (uintptr_t)(pmp + 1) + round4(npage);
3856                         /*
3857                          * It's possible that the address space can change
3858                          * subtlely even though we're holding as->a_lock
3859                          * due to the nondeterminism of page_exists() in
3860                          * the presence of asychronously flushed pages or
3861                          * mapped files whose sizes are changing.
3862                          * page_exists() may be called indirectly from
3863                          * pr_getprot() by a segop_incore() routine.
3863                          * pr_getprot() by a SEGOP_INCORE() routine.
3864                          * If this happens we need to make sure we don't
3865                          * overrun the buffer whose size we computed based
3866                          * on the initial iteration through the segments.
3867                          * Once we've detected an overflow, we need to clean
3868                          * up the temporary memory allocated in pr_getprot()
3869                          * and retry. If there's a pending signal, we return
3870                          * EINTR so that this thread can be dislodged if
3871                          * a latent bug causes us to spin indefinitely.
3872                          */
3873                         if (next > (uintptr_t)buf + size) {
3874                                 pr_getprot_done(&tmp);
3875                                 AS_LOCK_EXIT(as, &as->a_lock);

3877                                 kmem_free(buf, size);

3879                                 if (ISSIG(curthread, JUSTLOOKING))
3880                                         return (EINTR);

3882                                 goto again;
3883                         }

3885                         php->pr_nmap++;
3886                         php->pr_npage += npage;
3887                         pmp->pr_vaddr = (uint32_t)(uintptr_t)saddr;
3888                         pmp->pr_npage = (uint32_t)npage;
3889                         pmp->pr_off = (int32_t)segop_getoffset(seg, saddr);
```

```
3889                        pmp->pr_off = (int32_t)SEGOP_GETOFFSET(seg, saddr);
3890                        pmp->pr_mflags = 0;
3891                        if (prot & PROT_READ)
3892                                pmp->pr_mflags |= MA_READ;
3893                        if (prot & PROT_WRITE)
3894                                pmp->pr_mflags |= MA_WRITE;
3895                        if (prot & PROT_EXEC)
3896                                pmp->pr_mflags |= MA_EXEC;
3897                        if (segop_gettype(seg, saddr) & MAP_SHARED)
3897                        if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
3898                                pmp->pr_mflags |= MA_SHARED;
3899                        pmp->pr_pagesize = PAGESIZE;
3900                        hat_getstat(as, saddr, len, hatid,
3901                            (char *)(pmp + 1), HAT_SYNC_ZERORM);
3902                        pmp = (ioc_prasmap32_t *)next;
3903                }
3904                ASSERT(tmp == NULL);
3905        } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

3907        AS_LOCK_EXIT(as, &as->a_lock);

3909        ASSERT((uintptr_t)pmp == (uintptr_t)buf + size);
3910        error = uiomove(buf, (caddr_t)pmp - buf, UIO_READ, uiop);
3911        kmem_free(buf, size);

3913        return (error);
3914 }
```

_____unchanged_portion_omitted_

```
**********************************************************
  112647 Fri May  8 18:10:27 2015
new/usr/src/uts/common/fs/proc/prsubr.c
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

  98 size_t pagev_lim = 256 * 1024;  /* limit on number of pages in prpagev_t */

 100 extern const struct seg_ops segdev_ops; /* needs a header file */
 101 extern const struct seg_ops segspt_shmops;      /* needs a header file */
 100 extern struct seg_ops segdev_ops;       /* needs a header file */
 101 extern struct seg_ops segspt_shmops;    /* needs a header file */

 103 static  int     set_watched_page(proc_t *, caddr_t, caddr_t, ulong_t, ulong_t);
 104 static  void    clear_watched_page(proc_t *, caddr_t, caddr_t, ulong_t);

 106 /*
 107  * Choose an lwp from the complete set of lwps for the process.
 108  * This is called for any operation applied to the process
 109  * file descriptor that requires an lwp to operate upon.
 110  *
 111  * Returns a pointer to the thread for the selected LWP,
 112  * and with the dispatcher lock held for the thread.
 113  *
 114  * The algorithm for choosing an lwp is critical for /proc semantics;
 115  * don't touch this code unless you know all of the implications.
 116  */
 117 kthread_t *
 118 prchoose(proc_t *p)
 119 {
 120         kthread_t *t;
 121         kthread_t *t_onproc = NULL;     /* running on processor */
 122         kthread_t *t_run = NULL;        /* runnable, on disp queue */
 123         kthread_t *t_sleep = NULL;      /* sleeping */
 124         kthread_t *t_hold = NULL;       /* sleeping, performing hold */
 125         kthread_t *t_susp = NULL;       /* suspended stop */
 126         kthread_t *t_jstop = NULL;      /* jobcontrol stop, w/o directed stop */
 127         kthread_t *t_jdstop = NULL;     /* jobcontrol stop with directed stop */
 128         kthread_t *t_req = NULL;        /* requested stop */
 129         kthread_t *t_istop = NULL;      /* event-of-interest stop */
 130         kthread_t *t_dtrace = NULL;     /* DTrace stop */

 132         ASSERT(MUTEX_HELD(&p->p_lock));

 134         /*
 135          * If the agent lwp exists, it takes precedence over all others.
 136          */
 137         if ((t = p->p_agenttp) != NULL) {
 138                 thread_lock(t);
 139                 return (t);
 140         }

 142         if ((t = p->p_tlist) == NULL)   /* start at the head of the list */
 143                 return (t);
 144         do {            /* for eacn lwp in the process */
 145                 if (VSTOPPED(t)) {      /* virtually stopped */
 146                         if (t_req == NULL)
 147                                 t_req = t;
 148                         continue;
 149                 }

 151                 thread_lock(t);         /* make sure thread is in good state */
 152                 switch (t->t_state) {
```



```
 153                 default:
 154                         panic("prchoose: bad thread state %d, thread 0x%p",
 155                             t->t_state, (void *)t);
 156                         /*NOTREACHED*/
 157                 case TS_SLEEP:
 158                         /* this is filthy */
 159                         if (t->t_wchan == (caddr_t)&p->p_holdlwps &&
 160                             t->t_wchan0 == NULL) {
 161                                 if (t_hold == NULL)
 162                                         t_hold = t;
 163                         } else {
 164                                 if (t_sleep == NULL)
 165                                         t_sleep = t;
 166                         }
 167                         break;
 168                 case TS_RUN:
 169                 case TS_WAIT:
 170                         if (t_run == NULL)
 171                                 t_run = t;
 172                         break;
 173                 case TS_ONPROC:
 174                         if (t_onproc == NULL)
 175                                 t_onproc = t;
 176                         break;
 177                 case TS_ZOMB:           /* last possible choice */
 178                         break;
 179                 case TS_STOPPED:
 180                         switch (t->t_whystop) {
 181                         case PR_SUSPENDED:
 182                                 if (t_susp == NULL)
 183                                         t_susp = t;
 184                                 break;
 185                         case PR_JOBCONTROL:
 186                                 if (t->t_proc_flag & TP_PRSTOP) {
 187                                         if (t_jdstop == NULL)
 188                                                 t_jdstop = t;
 189                                 } else {
 190                                         if (t_jstop == NULL)
 191                                                 t_jstop = t;
 192                                 }
 193                                 break;
 194                         case PR_REQUESTED:
 195                                 if (t->t_dtrace_stop && t_dtrace == NULL)
 196                                         t_dtrace = t;
 197                                 else if (t_req == NULL)
 198                                         t_req = t;
 199                                 break;
 200                         case PR_SYSENTRY:
 201                         case PR_SYSEXIT:
 202                         case PR_SIGNALLED:
 203                         case PR_FAULTED:
 204                                 /*
 205                                  * Make an lwp calling exit() be the
 206                                  * last lwp seen in the process.
 207                                  */
 208                                 if (t_istop == NULL ||
 209                                     (t_istop->t_whystop == PR_SYSENTRY &&
 210                                     t_istop->t_whatstop == SYS_exit))
 211                                         t_istop = t;
 212                                 break;
 213                         case PR_CHECKPOINT:     /* can't happen? */
 214                                 break;
 215                         default:
 216                                 panic("prchoose: bad t_whystop %d, thread 0x%p",
 217                                     t->t_whystop, (void *)t);
 218                                 /*NOTREACHED*/
```

```
 219                        }
 220                        break;
 221                }
 222                thread_unlock(t);
 223        } while ((t = t->t_forw) != p->p_tlist);

 225        if (t_onproc)
 226                t = t_onproc;
 227        else if (t_run)
 228                t = t_run;
 229        else if (t_sleep)
 230                t = t_sleep;
 231        else if (t_jstop)
 232                t = t_jstop;
 233        else if (t_jdstop)
 234                t = t_jdstop;
 235        else if (t_istop)
 236                t = t_istop;
 237        else if (t_dtrace)
 238                t = t_dtrace;
 239        else if (t_req)
 240                t = t_req;
 241        else if (t_hold)
 242                t = t_hold;
 243        else if (t_susp)
 244                t = t_susp;
 245        else                        /* TS_ZOMB */
 246                t = p->p_tlist;

 248        if (t != NULL)
 249                thread_lock(t);
 250        return (t);
 251 }
```

*_____unchanged_portion_omitted_*

```
1475 struct seg *
1476 break_seg(proc_t *p)
1477 {
1478        caddr_t addr = p->p_brkbase;
1479        struct seg *seg;
1480        struct vnode *vp;

1482        if (p->p_brksize != 0)
1483                addr += p->p_brksize - 1;
1484        seg = as_segat(p->p_as, addr);
1485        if (seg != NULL && seg->s_ops == &segvn_ops &&
1486            (segop_getvp(seg, seg->s_base, &vp) != 0 || vp == NULL))
1486            (SEGOP_GETVP(seg, seg->s_base, &vp) != 0 || vp == NULL))
1487                return (seg);
1488        return (NULL);
1489 }
```

*_____unchanged_portion_omitted_*

```
1607 /*
1608  * Return an array of structures with memory map information.
1609  * We allocate here; the caller must deallocate.
1610  */
1611 int
1612 prgetmap(proc_t *p, int reserved, list_t *iolhead)
1613 {
1614        struct as *as = p->p_as;
1615        prmap_t *mp;
1616        struct seg *seg;
1617        struct seg *brkseg, *stkseg;
1618        struct vnode *vp;
1619        struct vattr vattr;
```



```
1620        uint_t prot;

1622        ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

1624        /*
1625         * Request an initial buffer size that doesn't waste memory
1626         * if the address space has only a small number of segments.
1627         */
1628        pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

1630        if ((seg = AS_SEGFIRST(as)) == NULL)
1631                return (0);

1633        brkseg = break_seg(p);
1634        stkseg = as_segat(as, prgetstackbase(p));

1636        do {
1637                caddr_t eaddr = seg->s_base + pr_getsegsize(seg, reserved);
1638                caddr_t saddr, naddr;
1639                void *tmp = NULL;

1641                for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
1642                        prot = pr_getprot(seg, reserved, &tmp,
1643                            &saddr, &naddr, eaddr);
1644                        if (saddr == naddr)
1645                                continue;

1647                        mp = pr_iol_newbuf(iolhead, sizeof (*mp));

1649                        mp->pr_vaddr = (uintptr_t)saddr;
1650                        mp->pr_size = naddr - saddr;
1651                        mp->pr_offset = segop_getoffset(seg, saddr);
1651                        mp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
1652                        mp->pr_mflags = 0;
1653                        if (prot & PROT_READ)
1654                                mp->pr_mflags |= MA_READ;
1655                        if (prot & PROT_WRITE)
1656                                mp->pr_mflags |= MA_WRITE;
1657                        if (prot & PROT_EXEC)
1658                                mp->pr_mflags |= MA_EXEC;
1659                        if (segop_gettype(seg, saddr) & MAP_SHARED)
1659                        if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
1660                                mp->pr_mflags |= MA_SHARED;
1661                        if (segop_gettype(seg, saddr) & MAP_NORESERVE)
1661                        if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
1662                                mp->pr_mflags |= MA_NORESERVE;
1663                        if (seg->s_ops == &segspt_shmops ||
1664                            (seg->s_ops == &segvn_ops &&
1665                            (segop_getvp(seg, saddr, &vp) != 0 || vp == NULL)))
1665                            (SEGOP_GETVP(seg, saddr, &vp) != 0 || vp == NULL)))
1666                                mp->pr_mflags |= MA_ANON;
1667                        if (seg == brkseg)
1668                                mp->pr_mflags |= MA_BREAK;
1669                        else if (seg == stkseg) {
1670                                mp->pr_mflags |= MA_STACK;
1671                                if (reserved) {
1672                                        size_t maxstack =
1673                                            ((size_t)p->p_stk_ctl +
1674                                            PAGEOFFSET) & PAGEMASK;
1675                                        mp->pr_vaddr =
1676                                            (uintptr_t)prgetstackbase(p) +
1677                                            p->p_stksize - maxstack;
1678                                        mp->pr_size = (uintptr_t)naddr -
1679                                            mp->pr_vaddr;
1680                                }
1681                        }
```

```
1682                        if (seg->s_ops == &segspt_shmops)
1683                                mp->pr_mflags |= MA_ISM | MA_SHM;
1684                        mp->pr_pagesize = PAGESIZE;

1686                        /*
1687                         * Manufacture a filename for the "object" directory.
1688                         */
1689                        vattr.va_mask = AT_FSID|AT_NODEID;
1690                        if (seg->s_ops == &segvn_ops &&
1691                            segop_getvp(seg, saddr, &vp) == 0 &&
1691                            SEGOP_GETVP(seg, saddr, &vp) == 0 &&
1692                            vp != NULL && vp->v_type == VREG &&
1693                            VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
1694                                if (vp == p->p_exec)
1695                                        (void) strcpy(mp->pr_mapname, "a.out");
1696                                else
1697                                        pr_object_name(mp->pr_mapname,
1698                                            vp, &vattr);
1699                        }

1701                        /*
1702                         * Get the SysV shared memory id, if any.
1703                         */
1704                        if ((mp->pr_mflags & MA_SHARED) && p->p_segacct &&
1705                            (mp->pr_shmid = shmgetid(p, seg->s_base)) !=
1706                            SHMID_NONE) {
1707                                if (mp->pr_shmid == SHMID_FREE)
1708                                        mp->pr_shmid = -1;

1710                                mp->pr_mflags |= MA_SHM;
1711                        } else {
1712                                mp->pr_shmid = -1;
1713                        }
1714                }
1715                ASSERT(tmp == NULL);
1716        } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

1718        return (0);
1719 }

1721 #ifdef _SYSCALL32_IMPL
1722 int
1723 prgetmap32(proc_t *p, int reserved, list_t *iolhead)
1724 {
1725        struct as *as = p->p_as;
1726        prmap32_t *mp;
1727        struct seg *seg;
1728        struct seg *brkseg, *stkseg;
1729        struct vnode *vp;
1730        struct vattr vattr;
1731        uint_t prot;

1733        ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

1735        /*
1736         * Request an initial buffer size that doesn't waste memory
1737         * if the address space has only a small number of segments.
1738         */
1739        pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

1741        if ((seg = AS_SEGFIRST(as)) == NULL)
1742                return (0);

1744        brkseg = break_seg(p);
1745        stkseg = as_segat(as, prgetstackbase(p));
```



```
1747        do {
1748                caddr_t eaddr = seg->s_base + pr_getsegsize(seg, reserved);
1749                caddr_t saddr, naddr;
1750                void *tmp = NULL;

1752                for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
1753                        prot = pr_getprot(seg, reserved, &tmp,
1754                            &saddr, &naddr, eaddr);
1755                        if (saddr == naddr)
1756                                continue;

1758                        mp = pr_iol_newbuf(iolhead, sizeof (*mp));

1760                        mp->pr_vaddr = (caddr32_t)(uintptr_t)saddr;
1761                        mp->pr_size = (size32_t)(naddr - saddr);
1762                        mp->pr_offset = segop_getoffset(seg, saddr);
1762                        mp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
1763                        mp->pr_mflags = 0;
1764                        if (prot & PROT_READ)
1765                                mp->pr_mflags |= MA_READ;
1766                        if (prot & PROT_WRITE)
1767                                mp->pr_mflags |= MA_WRITE;
1768                        if (prot & PROT_EXEC)
1769                                mp->pr_mflags |= MA_EXEC;
1770                        if (segop_gettype(seg, saddr) & MAP_SHARED)
1770                        if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
1771                                mp->pr_mflags |= MA_SHARED;
1772                        if (segop_gettype(seg, saddr) & MAP_NORESERVE)
1772                        if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
1773                                mp->pr_mflags |= MA_NORESERVE;
1774                        if (seg->s_ops == &segspt_shmops ||
1775                            (seg->s_ops == &segvn_ops &&
1776                            (segop_getvp(seg, saddr, &vp) != 0 || vp == NULL)))
1776                            (SEGOP_GETVP(seg, saddr, &vp) != 0 || vp == NULL)))
1777                                mp->pr_mflags |= MA_ANON;
1778                        if (seg == brkseg)
1779                                mp->pr_mflags |= MA_BREAK;
1780                        else if (seg == stkseg) {
1781                                mp->pr_mflags |= MA_STACK;
1782                                if (reserved) {
1783                                        size_t maxstack =
1784                                            ((size_t)p->p_stk_ctl +
1785                                            PAGEOFFSET) & PAGEMASK;
1786                                        uintptr_t vaddr =
1787                                            (uintptr_t)prgetstackbase(p) +
1788                                            p->p_stksize - maxstack;
1789                                        mp->pr_vaddr = (caddr32_t)vaddr;
1790                                        mp->pr_size = (size32_t)
1791                                            ((uintptr_t)naddr - vaddr);
1792                                }
1793                        }
1794                        if (seg->s_ops == &segspt_shmops)
1795                                mp->pr_mflags |= MA_ISM | MA_SHM;
1796                        mp->pr_pagesize = PAGESIZE;

1798                        /*
1799                         * Manufacture a filename for the "object" directory.
1800                         */
1801                        vattr.va_mask = AT_FSID|AT_NODEID;
1802                        if (seg->s_ops == &segvn_ops &&
1803                            segop_getvp(seg, saddr, &vp) == 0 &&
1803                            SEGOP_GETVP(seg, saddr, &vp) == 0 &&
1804                            vp != NULL && vp->v_type == VREG &&
1805                            VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
1806                                if (vp == p->p_exec)
1807                                        (void) strcpy(mp->pr_mapname, "a.out");
```

```
1808                                    else
1809                                            pr_object_name(mp->pr_mapname,
1810                                                vp, &vattr);
1811                            }
1813                            /*
1814                             * Get the SysV shared memory id, if any.
1815                             */
1816                            if ((mp->pr_mflags & MA_SHARED) && p->p_segacct &&
1817                                (mp->pr_shmid = shmgetid(p, seg->s_base)) !=
1818                                SHMID_NONE) {
1819                                    if (mp->pr_shmid == SHMID_FREE)
1820                                            mp->pr_shmid = -1;
1822                                    mp->pr_mflags |= MA_SHM;
1823                            } else {
1824                                    mp->pr_shmid = -1;
1825                            }
1826                    }
1827                    ASSERT(tmp == NULL);
1828            } while ((seg = AS_SEGNEXT(as, seg)) != NULL);
1830            return (0);
1831 }
_____unchanged_portion_omitted_
1895 #endif  /* _SYSCALL32_IMPL */
1897 /*
1898  * Read page data information.
1899  */
1900 int
1901 prpdread(proc_t *p, uint_t hatid, struct uio *uiop)
1902 {
1903            struct as *as = p->p_as;
1904            caddr_t buf;
1905            size_t size;
1906            prpageheader_t *php;
1907            prasmap_t *pmp;
1908            struct seg *seg;
1909            int error;
1911 again:
1912            AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1914            if ((seg = AS_SEGFIRST(as)) == NULL) {
1915                    AS_LOCK_EXIT(as, &as->a_lock);
1916                    return (0);
1917            }
1918            size = prpdsize(as);
1919            if (uiop->uio_resid < size) {
1920                    AS_LOCK_EXIT(as, &as->a_lock);
1921                    return (E2BIG);
1922            }
1924            buf = kmem_zalloc(size, KM_SLEEP);
1925            php = (prpageheader_t *)buf;
1926            pmp = (prasmap_t *)(buf + sizeof (prpageheader_t));
1928            hrt2ts(gethrtime(), &php->pr_tstamp);
1929            php->pr_nmap = 0;
1930            php->pr_npage = 0;
1931            do {
1932                    caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
1933                    caddr_t saddr, naddr;
1934                    void *tmp = NULL;
```



```
1936                    for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
1937                            struct vnode *vp;
1938                            struct vattr vattr;
1939                            size_t len;
1940                            size_t npage;
1941                            uint_t prot;
1942                            uintptr_t next;
1944                            prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
1945                            if ((len = (size_t)(naddr - saddr)) == 0)
1946                                    continue;
1947                            npage = len / PAGESIZE;
1948                            next = (uintptr_t)(pmp + 1) + round8(npage);
1949                            /*
1950                             * It's possible that the address space can change
1951                             * subtlely even though we're holding as->a_lock
1952                             * due to the nondeterminism of page_exists() in
1953                             * the presence of asychronously flushed pages or
1954                             * mapped files whose sizes are changing.
1955                             * page_exists() may be called indirectly from
1956                             * pr_getprot() by a segop_incore() routine.
1956                             * pr_getprot() by a SEGOP_INCORE() routine.
1957                             * If this happens we need to make sure we don't
1958                             * overrun the buffer whose size we computed based
1959                             * on the initial iteration through the segments.
1960                             * Once we've detected an overflow, we need to clean
1961                             * up the temporary memory allocated in pr_getprot()
1962                             * and retry. If there's a pending signal, we return
1963                             * EINTR so that this thread can be dislodged if
1964                             * a latent bug causes us to spin indefinitely.
1965                             */
1966                            if (next > (uintptr_t)buf + size) {
1967                                    pr_getprot_done(&tmp);
1968                                    AS_LOCK_EXIT(as, &as->a_lock);
1970                                    kmem_free(buf, size);
1972                                    if (ISSIG(curthread, JUSTLOOKING))
1973                                            return (EINTR);
1975                                    goto again;
1976                            }
1978                            php->pr_nmap++;
1979                            php->pr_npage += npage;
1980                            pmp->pr_vaddr = (uintptr_t)saddr;
1981                            pmp->pr_npage = npage;
1982                            pmp->pr_offset = segop_getoffset(seg, saddr);
1982                            pmp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
1983                            pmp->pr_mflags = 0;
1984                            if (prot & PROT_READ)
1985                                    pmp->pr_mflags |= MA_READ;
1986                            if (prot & PROT_WRITE)
1987                                    pmp->pr_mflags |= MA_WRITE;
1988                            if (prot & PROT_EXEC)
1989                                    pmp->pr_mflags |= MA_EXEC;
1990                            if (segop_gettype(seg, saddr) & MAP_SHARED)
1990                            if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
1991                                    pmp->pr_mflags |= MA_SHARED;
1992                            if (segop_gettype(seg, saddr) & MAP_NORESERVE)
1992                            if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
1993                                    pmp->pr_mflags |= MA_NORESERVE;
1994                            if (seg->s_ops == &segspt_shmops ||
1995                                (seg->s_ops == &segvn_ops &&
1996                                (segop_getvp(seg, saddr, &vp) != 0 || vp == NULL)))
1996                                (SEGOP_GETVP(seg, saddr, &vp) != 0 || vp == NULL)))
```

```
1997                                    pmp->pr_mflags |= MA_ANON;
1998                            if (seg->s_ops == &segspt_shmops)
1999                                    pmp->pr_mflags |= MA_ISM | MA_SHM;
2000                            pmp->pr_pagesize = PAGESIZE;
2001                            /*
2002                             * Manufacture a filename for the "object" directory.
2003                             */
2004                            vattr.va_mask = AT_FSID|AT_NODEID;
2005                            if (seg->s_ops == &segvn_ops &&
2006                                segop_getvp(seg, saddr, &vp) == 0 &&
2006                                SEGOP_GETVP(seg, saddr, &vp) == 0 &&
2007                                vp != NULL && vp->v_type == VREG &&
2008                                VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
2009                                    if (vp == p->p_exec)
2010                                            (void) strcpy(pmp->pr_mapname, "a.out");
2011                                    else
2012                                            pr_object_name(pmp->pr_mapname,
2013                                                vp, &vattr);
2014                            }

2016                            /*
2017                             * Get the SysV shared memory id, if any.
2018                             */
2019                            if ((pmp->pr_mflags & MA_SHARED) && p->p_segacct &&
2020                                (pmp->pr_shmid = shmgetid(p, seg->s_base)) !=
2021                                SHMID_NONE) {
2022                                    if (pmp->pr_shmid == SHMID_FREE)
2023                                            pmp->pr_shmid = -1;

2025                                    pmp->pr_mflags |= MA_SHM;
2026                            } else {
2027                                    pmp->pr_shmid = -1;
2028                            }

2030                            hat_getstat(as, saddr, len, hatid,
2031                                (char *)(pmp + 1), HAT_SYNC_ZERORM);
2032                            pmp = (prasmap_t *)next;
2033                    }
2034                    ASSERT(tmp == NULL);
2035            } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

2037            AS_LOCK_EXIT(as, &as->a_lock);

2039            ASSERT((uintptr_t)pmp <= (uintptr_t)buf + size);
2040            error = uiomove(buf, (caddr_t)pmp - buf, UIO_READ, uiop);
2041            kmem_free(buf, size);

2043            return (error);
2044    }

2046    #ifdef _SYSCALL32_IMPL
2047    int
2048    prpdread32(proc_t *p, uint_t hatid, struct uio *uiop)
2049    {
2050            struct as *as = p->p_as;
2051            caddr_t buf;
2052            size_t size;
2053            prpageheader32_t *php;
2054            prasmap32_t *pmp;
2055            struct seg *seg;
2056            int error;

2058    again:
2059            AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

2061            if ((seg = AS_SEGFIRST(as)) == NULL) {
```



```
2062                    AS_LOCK_EXIT(as, &as->a_lock);
2063                    return (0);
2064            }
2065            size = prpdsize32(as);
2066            if (uiop->uio_resid < size) {
2067                    AS_LOCK_EXIT(as, &as->a_lock);
2068                    return (E2BIG);
2069            }

2071            buf = kmem_zalloc(size, KM_SLEEP);
2072            php = (prpageheader32_t *)buf;
2073            pmp = (prasmap32_t *)(buf + sizeof (prpageheader32_t));

2075            hrt2ts32(gethrtime(), &php->pr_tstamp);
2076            php->pr_nmap = 0;
2077            php->pr_npage = 0;
2078            do {
2079                    caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
2080                    caddr_t saddr, naddr;
2081                    void *tmp = NULL;

2083                    for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
2084                            struct vnode *vp;
2085                            struct vattr vattr;
2086                            size_t len;
2087                            size_t npage;
2088                            uint_t prot;
2089                            uintptr_t next;

2091                            prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
2092                            if ((len = (size_t)(naddr - saddr)) == 0)
2093                                    continue;
2094                            npage = len / PAGESIZE;
2095                            next = (uintptr_t)(pmp + 1) + round8(npage);
2096                            /*
2097                             * It's possible that the address space can change
2098                             * subtlely even though we're holding as->a_lock
2099                             * due to the nondeterminism of page_exists() in
2100                             * the presence of asychronously flushed pages or
2101                             * mapped files whose sizes are changing.
2102                             * page_exists() may be called indirectly from
2103                             * pr_getprot() by a segop_incore() routine.
2103                             * pr_getprot() by a SEGOP_INCORE() routine.
2104                             * If this happens we need to make sure we don't
2105                             * overrun the buffer whose size we computed based
2106                             * on the initial iteration through the segments.
2107                             * Once we've detected an overflow, we need to clean
2108                             * up the temporary memory allocated in pr_getprot()
2109                             * and retry. If there's a pending signal, we return
2110                             * EINTR so that this thread can be dislodged if
2111                             * a latent bug causes us to spin indefinitely.
2112                             */
2113                            if (next > (uintptr_t)buf + size) {
2114                                    pr_getprot_done(&tmp);
2115                                    AS_LOCK_EXIT(as, &as->a_lock);

2117                                    kmem_free(buf, size);

2119                                    if (ISSIG(curthread, JUSTLOOKING))
2120                                            return (EINTR);

2122                                    goto again;
2123                            }

2125                            php->pr_nmap++;
2126                            php->pr_npage += npage;
```

```
2127                        pmp->pr_vaddr = (caddr32_t)(uintptr_t)saddr;
2128                        pmp->pr_npage = (size32_t)npage;
2129                        pmp->pr_offset = segop_getoffset(seg, saddr);
2129                        pmp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
2130                        pmp->pr_mflags = 0;
2131                        if (prot & PROT_READ)
2132                                pmp->pr_mflags |= MA_READ;
2133                        if (prot & PROT_WRITE)
2134                                pmp->pr_mflags |= MA_WRITE;
2135                        if (prot & PROT_EXEC)
2136                                pmp->pr_mflags |= MA_EXEC;
2137                        if (segop_gettype(seg, saddr) & MAP_SHARED)
2137                        if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
2138                                pmp->pr_mflags |= MA_SHARED;
2139                        if (segop_gettype(seg, saddr) & MAP_NORESERVE)
2139                        if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
2140                                pmp->pr_mflags |= MA_NORESERVE;
2141                        if (seg->s_ops == &segspt_shmops ||
2142                            (seg->s_ops == &segvn_ops &&
2143                            (segop_getvp(seg, saddr, &vp) != 0 || vp == NULL)))
2143                            (SEGOP_GETVP(seg, saddr, &vp) != 0 || vp == NULL)))
2144                                pmp->pr_mflags |= MA_ANON;
2145                        if (seg->s_ops == &segspt_shmops)
2146                                pmp->pr_mflags |= MA_ISM | MA_SHM;
2147                        pmp->pr_pagesize = PAGESIZE;
2148                        /*
2149                         * Manufacture a filename for the "object" directory.
2150                         */
2151                        vattr.va_mask = AT_FSID|AT_NODEID;
2152                        if (seg->s_ops == &segvn_ops &&
2153                            segop_getvp(seg, saddr, &vp) == 0 &&
2153                            SEGOP_GETVP(seg, saddr, &vp) == 0 &&
2154                            vp != NULL && vp->v_type == VREG &&
2155                            VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
2156                                if (vp == p->p_exec)
2157                                        (void) strcpy(pmp->pr_mapname, "a.out");
2158                                else
2159                                        pr_object_name(pmp->pr_mapname,
2160                                            vp, &vattr);
2161                        }

2163                        /*
2164                         * Get the SysV shared memory id, if any.
2165                         */
2166                        if ((pmp->pr_mflags & MA_SHARED) && p->p_segacct &&
2167                            (pmp->pr_shmid = shmgetid(p, seg->s_base)) !=
2168                            SHMID_NONE) {
2169                                if (pmp->pr_shmid == SHMID_FREE)
2170                                        pmp->pr_shmid = -1;

2172                                pmp->pr_mflags |= MA_SHM;
2173                        } else {
2174                                pmp->pr_shmid = -1;
2175                        }

2177                        hat_getstat(as, saddr, len, hatid,
2178                            (char *)(pmp + 1), HAT_SYNC_ZERORM);
2179                        pmp = (prasmap32_t *)next;
2180                }
2181                ASSERT(tmp == NULL);
2182        } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

2184        AS_LOCK_EXIT(as, &as->a_lock);

2186        ASSERT((uintptr_t)pmp <= (uintptr_t)buf + size);
2187        error = uiomove(buf, (caddr_t)pmp - buf, UIO_READ, uiop);
```



```
2188        kmem_free(buf, size);

2190        return (error);
2191 }
_____unchanged_portion_omitted_

3299 /*
3300  * This one is called by the traced process to unwatch all the
3301  * pages while deallocating the list of watched_page structs.
3302  */
3303 void
3304 pr_free_watched_pages(proc_t *p)
3305 {
3306        struct as *as = p->p_as;
3307        struct watched_page *pwp;
3308        uint_t prot;
3309        int    retrycnt, err;
3310        void *cookie;

3312        if (as == NULL || avl_numnodes(&as->a_wpage) == 0)
3313                return;

3315        ASSERT(MUTEX_NOT_HELD(&curproc->p_lock));
3316        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

3318        pwp = avl_first(&as->a_wpage);

3320        cookie = NULL;
3321        while ((pwp = avl_destroy_nodes(&as->a_wpage, &cookie)) != NULL) {
3322                retrycnt = 0;
3323                if ((prot = pwp->wp_oprot) != 0) {
3324                        caddr_t addr = pwp->wp_vaddr;
3325                        struct seg *seg;
3326                retry:

3328                        if ((pwp->wp_prot != prot ||
3329                            (pwp->wp_flags & WP_NOWATCH)) &&
3330                            (seg = as_segat(as, addr)) != NULL) {
3331                                err = segop_setprot(seg, addr, PAGESIZE, prot);
3331                                err = SEGOP_SETPROT(seg, addr, PAGESIZE, prot);
3332                                if (err == IE_RETRY) {
3333                                        ASSERT(retrycnt == 0);
3334                                        retrycnt++;
3335                                        goto retry;
3336                                }
3337                        }
3338                }
3339                kmem_free(pwp, sizeof (struct watched_page));
3340        }

3342        avl_destroy(&as->a_wpage);
3343        p->p_wprot = NULL;

3345        AS_LOCK_EXIT(as, &as->a_lock);
3346 }

3348 /*
3349  * Insert a watched area into the list of watched pages.
3350  * If oflags is zero then we are adding a new watched area.
3351  * Otherwise we are changing the flags of an existing watched area.
3352  */
3353 static int
3354 set_watched_page(proc_t *p, caddr_t vaddr, caddr_t eaddr,
3355        ulong_t flags, ulong_t oflags)
3356 {
3357        struct as *as = p->p_as;
```

```
3358         avl_tree_t *pwp_tree;
3359         struct watched_page *pwp, *newpwp;
3360         struct watched_page tpw;
3361         avl_index_t where;
3362         struct seg *seg;
3363         uint_t prot;
3364         caddr_t addr;

3366         /*
3367          * We need to pre-allocate a list of structures before we grab the
3368          * address space lock to avoid calling kmem_alloc(KM_SLEEP) with locks
3369          * held.
3370          */
3371         newpwp = NULL;
3372         for (addr = (caddr_t)((uintptr_t)vaddr & (uintptr_t)PAGEMASK);
3373             addr < eaddr; addr += PAGESIZE) {
3374                 pwp = kmem_zalloc(sizeof (struct watched_page), KM_SLEEP);
3375                 pwp->wp_list = newpwp;
3376                 newpwp = pwp;
3377         }

3379         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

3381         /*
3382          * Search for an existing watched page to contain the watched area.
3383          * If none is found, grab a new one from the available list
3384          * and insert it in the active list, keeping the list sorted
3385          * by user-level virtual address.
3386          */
3387         if (p->p_flag & SVFWAIT)
3388                 pwp_tree = &p->p_wpage;
3389         else
3390                 pwp_tree = &as->a_wpage;

3392 again:
3393         if (avl_numnodes(pwp_tree) > prnwatch) {
3394                 AS_LOCK_EXIT(as, &as->a_lock);
3395                 while (newpwp != NULL) {
3396                         pwp = newpwp->wp_list;
3397                         kmem_free(newpwp, sizeof (struct watched_page));
3398                         newpwp = pwp;
3399                 }
3400                 return (E2BIG);
3401         }

3403         tpw.wp_vaddr = (caddr_t)((uintptr_t)vaddr & (uintptr_t)PAGEMASK);
3404         if ((pwp = avl_find(pwp_tree, &tpw, &where)) == NULL) {
3405                 pwp = newpwp;
3406                 newpwp = newpwp->wp_list;
3407                 pwp->wp_list = NULL;
3408                 pwp->wp_vaddr = (caddr_t)((uintptr_t)vaddr &
3409                     (uintptr_t)PAGEMASK);
3410                 avl_insert(pwp_tree, pwp, where);
3411         }

3413         ASSERT(vaddr >= pwp->wp_vaddr && vaddr < pwp->wp_vaddr + PAGESIZE);

3415         if (oflags & WA_READ)
3416                 pwp->wp_read--;
3417         if (oflags & WA_WRITE)
3418                 pwp->wp_write--;
3419         if (oflags & WA_EXEC)
3420                 pwp->wp_exec--;

3422         ASSERT(pwp->wp_read >= 0);
3423         ASSERT(pwp->wp_write >= 0);
```



```
3424         ASSERT(pwp->wp_exec >= 0);

3426         if (flags & WA_READ)
3427                 pwp->wp_read++;
3428         if (flags & WA_WRITE)
3429                 pwp->wp_write++;
3430         if (flags & WA_EXEC)
3431                 pwp->wp_exec++;

3433         if (!(p->p_flag & SVFWAIT)) {
3434                 vaddr = pwp->wp_vaddr;
3435                 if (pwp->wp_oprot == 0 &&
3436                     (seg = as_segat(as, vaddr)) != NULL) {
3437                         segop_getprot(seg, vaddr, 0, &prot);
3437                         SEGOP_GETPROT(seg, vaddr, 0, &prot);
3438                         pwp->wp_oprot = (uchar_t)prot;
3439                         pwp->wp_prot = (uchar_t)prot;
3440                 }
3441                 if (pwp->wp_oprot != 0) {
3442                         prot = pwp->wp_oprot;
3443                         if (pwp->wp_read)
3444                                 prot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3445                         if (pwp->wp_write)
3446                                 prot &= ~PROT_WRITE;
3447                         if (pwp->wp_exec)
3448                                 prot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3449                         if (!(pwp->wp_flags & WP_NOWATCH) &&
3450                             pwp->wp_prot != prot &&
3451                             (pwp->wp_flags & WP_SETPROT) == 0) {
3452                                 pwp->wp_flags |= WP_SETPROT;
3453                                 pwp->wp_list = p->p_wprot;
3454                                 p->p_wprot = pwp;
3455                         }
3456                         pwp->wp_prot = (uchar_t)prot;
3457                 }
3458         }

3460         /*
3461          * If the watched area extends into the next page then do
3462          * it over again with the virtual address of the next page.
3463          */
3464         if ((vaddr = pwp->wp_vaddr + PAGESIZE) < eaddr)
3465                 goto again;

3467         AS_LOCK_EXIT(as, &as->a_lock);

3469         /*
3470          * Free any pages we may have over-allocated
3471          */
3472         while (newpwp != NULL) {
3473                 pwp = newpwp->wp_list;
3474                 kmem_free(newpwp, sizeof (struct watched_page));
3475                 newpwp = pwp;
3476         }

3478         return (0);
3479 }
```

*_____unchanged_portion_omitted_*

```
3614 static caddr_t
3615 pr_pagev_fill(prpagev_t *pagev, struct seg *seg, caddr_t addr, caddr_t eaddr)
3616 {
3617         ulong_t lastpg = seg_page(seg, eaddr - 1);
3618         ulong_t pn, pnlim;
3619         caddr_t saddr;
3620         size_t len;
```

```
3622         ASSERT(addr >= seg->s_base && addr <= eaddr);

3624         if (addr == eaddr)
3625                 return (eaddr);

3627 refill:
3628         ASSERT(addr < eaddr);
3629         pagev->pg_pnbase = seg_page(seg, addr);
3630         pnlim = pagev->pg_pnbase + pagev->pg_npages;
3631         saddr = addr;

3633         if (lastpg < pnlim)
3634                 len = (size_t)(eaddr - addr);
3635         else
3636                 len = pagev->pg_npages * PAGESIZE;

3638         if (pagev->pg_incore != NULL) {
3639                 /*
3640                  * INCORE cleverly has different semantics than GETPROT:
3641                  * it returns info on pages up to but NOT including addr + len.
3642                  */
3643                 segop_incore(seg, addr, len, pagev->pg_incore);
3643                 SEGOP_INCORE(seg, addr, len, pagev->pg_incore);
3644                 pn = pagev->pg_pnbase;

3646                 do {
3647                         /*
3648                          * Guilty knowledge here:  We know that segvn_incore
3649                          * returns more than just the low-order bit that
3650                          * indicates the page is actually in memory.  If any
3651                          * bits are set, then the page has backing store.
3652                          */
3653                         if (pagev->pg_incore[pn++ - pagev->pg_pnbase])
3654                                 goto out;

3656                 } while ((addr += PAGESIZE) < eaddr && pn < pnlim);

3658                 /*
3659                  * If we examined all the pages in the vector but we're not
3660                  * at the end of the segment, take another lap.
3661                  */
3662                 if (addr < eaddr)
3663                         goto refill;
3664         }

3666         /*
3667          * Need to take len - 1 because addr + len is the address of the
3668          * first byte of the page just past the end of what we want.
3669          */
3670 out:
3671         segop_getprot(seg, saddr, len - 1, pagev->pg_protv);
3671         SEGOP_GETPROT(seg, saddr, len - 1, pagev->pg_protv);
3672         return (addr);
3673 }
```

*unchanged_portion_omitted_*

```
3768 size_t
3769 pr_getsegsize(struct seg *seg, int reserved)
3770 {
3771         size_t size = seg->s_size;

3773         /*
3774          * If we're interested in the reserved space, return the size of the
3775          * segment itself.  Everything else in this function is a special case
3776          * to determine the actual underlying size of various segment types.
```



```
3777          */
3778         if (reserved)
3779                 return (size);

3781         /*
3782          * If this is a segvn mapping of a regular file, return the smaller
3783          * of the segment size and the remaining size of the file beyond
3784          * the file offset corresponding to seg->s_base.
3785          */
3786         if (seg->s_ops == &segvn_ops) {
3787                 vattr_t vattr;
3788                 vnode_t *vp;

3790                 vattr.va_mask = AT_SIZE;

3792                 if (segop_getvp(seg, seg->s_base, &vp) == 0 &&
3792                 if (SEGOP_GETVP(seg, seg->s_base, &vp) == 0 &&
3793                     vp != NULL && vp->v_type == VREG &&
3794                     VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {

3796                         u_offset_t fsize = vattr.va_size;
3797                         u_offset_t offset = segop_getoffset(seg, seg->s_base);
3797                         u_offset_t offset = SEGOP_GETOFFSET(seg, seg->s_base);

3799                         if (fsize < offset)
3800                                 fsize = 0;
3801                         else
3802                                 fsize -= offset;

3804                         fsize = roundup(fsize, (u_offset_t)PAGESIZE);

3806                         if (fsize < (u_offset_t)size)
3807                                 size = (size_t)fsize;
3808                 }

3810                 return (size);
3811         }

3813         /*
3814          * If this is an ISM shared segment, don't include pages that are
3815          * beyond the real size of the spt segment that backs it.
3816          */
3817         if (seg->s_ops == &segspt_shmops)
3818                 return (MIN(spt_realsize(seg), size));

3820         /*
3821          * If this is segment is a mapping from /dev/null, then this is a
3822          * reservation of virtual address space and has no actual size.
3823          * Such segments are backed by segdev and have type set to neither
3824          * MAP_SHARED nor MAP_PRIVATE.
3825          */
3826         if (seg->s_ops == &segdev_ops &&
3827             ((segop_gettype(seg, seg->s_base) &
3827             ((SEGOP_GETTYPE(seg, seg->s_base) &
3828             (MAP_SHARED | MAP_PRIVATE)) == 0))
3829                 return (0);

3831         /*
3832          * If this segment doesn't match one of the special types we handle,
3833          * just return the size of the segment itself.
3834          */
3835         return (size);
3836 }
```

*unchanged_portion_omitted_*

```
3997 /*
```

```
3998  * Return an array of structures with extended memory map information.
3999  * We allocate here; the caller must deallocate.
4000  */
4001 int
4002 prgetxmap(proc_t *p, list_t *iolhead)
4003 {
4004         struct as *as = p->p_as;
4005         prxmap_t *mp;
4006         struct seg *seg;
4007         struct seg *brkseg, *stkseg;
4008         struct vnode *vp;
4009         struct vattr vattr;
4010         uint_t prot;

4012         ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

4014         /*
4015          * Request an initial buffer size that doesn't waste memory
4016          * if the address space has only a small number of segments.
4017          */
4018         pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

4020         if ((seg = AS_SEGFIRST(as)) == NULL)
4021                 return (0);

4023         brkseg = break_seg(p);
4024         stkseg = as_segat(as, prgetstackbase(p));

4026         do {
4027                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
4028                 caddr_t saddr, naddr, baddr;
4029                 void *tmp = NULL;
4030                 ssize_t psz;
4031                 char *parr;
4032                 uint64_t npages;
4033                 uint64_t pagenum;

4035                 /*
4036                  * Segment loop part one: iterate from the base of the segment
4037                  * to its end, pausing at each address boundary (baddr) between
4038                  * ranges that have different virtual memory protections.
4039                  */
4040                 for (saddr = seg->s_base; saddr < eaddr; saddr = baddr) {
4041                         prot = pr_getprot(seg, 0, &tmp, &saddr, &baddr, eaddr);
4042                         ASSERT(baddr >= saddr && baddr <= eaddr);

4044                         /*
4045                          * Segment loop part two: iterate from the current
4046                          * position to the end of the protection boundary,
4047                          * pausing at each address boundary (naddr) between
4048                          * ranges that have different underlying page sizes.
4049                          */
4050                         for (; saddr < baddr; saddr = naddr) {
4051                                 psz = pr_getpagesize(seg, saddr, &naddr, baddr);
4052                                 ASSERT(naddr >= saddr && naddr <= baddr);

4054                                 mp = pr_iol_newbuf(iolhead, sizeof (*mp));

4056                                 mp->pr_vaddr = (uintptr_t)saddr;
4057                                 mp->pr_size = naddr - saddr;
4058                                 mp->pr_offset = segop_getoffset(seg, saddr);
4058                                 mp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
4059                                 mp->pr_mflags = 0;
4060                                 if (prot & PROT_READ)
4061                                         mp->pr_mflags |= MA_READ;
4062                                 if (prot & PROT_WRITE)
```



```
4063                                         mp->pr_mflags |= MA_WRITE;
4064                                 if (prot & PROT_EXEC)
4065                                         mp->pr_mflags |= MA_EXEC;
4066                                 if (segop_gettype(seg, saddr) & MAP_SHARED)
4066                                 if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
4067                                         mp->pr_mflags |= MA_SHARED;
4068                                 if (segop_gettype(seg, saddr) & MAP_NORESERVE)
4068                                 if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
4069                                         mp->pr_mflags |= MA_NORESERVE;
4070                                 if (seg->s_ops == &segspt_shmops ||
4071                                     (seg->s_ops == &segvn_ops &&
4072                                         (segop_getvp(seg, saddr, &vp) != 0 ||
4072                                         (SEGOP_GETVP(seg, saddr, &vp) != 0 ||
4073                                         vp == NULL)))
4074                                         mp->pr_mflags |= MA_ANON;
4075                                 if (seg == brkseg)
4076                                         mp->pr_mflags |= MA_BREAK;
4077                                 else if (seg == stkseg)
4078                                         mp->pr_mflags |= MA_STACK;
4079                                 if (seg->s_ops == &segspt_shmops)
4080                                         mp->pr_mflags |= MA_ISM | MA_SHM;

4082                                 mp->pr_pagesize = PAGESIZE;
4083                                 if (psz == -1) {
4084                                         mp->pr_hatpagesize = 0;
4085                                 } else {
4086                                         mp->pr_hatpagesize = psz;
4087                                 }

4089                                 /*
4090                                  * Manufacture a filename for the "object" dir.
4091                                  */
4092                                 mp->pr_dev = PRNODEV;
4093                                 vattr.va_mask = AT_FSID|AT_NODEID;
4094                                 if (seg->s_ops == &segvn_ops &&
4095                                     segop_getvp(seg, saddr, &vp) == 0 &&
4095                                     SEGOP_GETVP(seg, saddr, &vp) == 0 &&
4096                                     vp != NULL && vp->v_type == VREG &&
4097                                     VOP_GETATTR(vp, &vattr, 0, CRED(),
4098                                     NULL) == 0) {
4099                                         mp->pr_dev = vattr.va_fsid;
4100                                         mp->pr_ino = vattr.va_nodeid;
4101                                         if (vp == p->p_exec)
4102                                                 (void) strcpy(mp->pr_mapname,
4103                                                     "a.out");
4104                                         else
4105                                                 pr_object_name(mp->pr_mapname,
4106                                                     vp, &vattr);
4107                                 }

4109                                 /*
4110                                  * Get the SysV shared memory id, if any.
4111                                  */
4112                                 if ((mp->pr_mflags & MA_SHARED) &&
4113                                     p->p_segacct && (mp->pr_shmid = shmgetid(p,
4114                                     seg->s_base)) != SHMID_NONE) {
4115                                         if (mp->pr_shmid == SHMID_FREE)
4116                                                 mp->pr_shmid = -1;

4118                                         mp->pr_mflags |= MA_SHM;
4119                                 } else {
4120                                         mp->pr_shmid = -1;
4121                                 }

4123                                 npages = ((uintptr_t)(naddr - saddr)) >>
4124                                     PAGESHIFT;
```

```
4125                                parr = kmem_zalloc(npages, KM_SLEEP);

4127                                segop_incore(seg, saddr, naddr - saddr, parr);
4127                                SEGOP_INCORE(seg, saddr, naddr - saddr, parr);

4129                                for (pagenum = 0; pagenum < npages; pagenum++) {
4130                                        if (parr[pagenum] & SEG_PAGE_INCORE)
4131                                                mp->pr_rss++;
4132                                        if (parr[pagenum] & SEG_PAGE_ANON)
4133                                                mp->pr_anon++;
4134                                        if (parr[pagenum] & SEG_PAGE_LOCKED)
4135                                                mp->pr_locked++;
4136                                }
4137                                kmem_free(parr, npages);
4138                        }
4139                }
4140                ASSERT(tmp == NULL);
4141        } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

4143        return (0);
4144 }
```

*unchanged_portion_omitted_*

```
4180 #ifdef _SYSCALL32_IMPL
4181 /*
4182  * Return an array of structures with HAT memory map information.
4183  * We allocate here; the caller must deallocate.
4184  */
4185 int
4186 prgetxmap32(proc_t *p, list_t *iolhead)
4187 {
4188        struct as *as = p->p_as;
4189        prxmap32_t *mp;
4190        struct seg *seg;
4191        struct seg *brkseg, *stkseg;
4192        struct vnode *vp;
4193        struct vattr vattr;
4194        uint_t prot;

4196        ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

4198        /*
4199         * Request an initial buffer size that doesn't waste memory
4200         * if the address space has only a small number of segments.
4201         */
4202        pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

4204        if ((seg = AS_SEGFIRST(as)) == NULL)
4205                return (0);

4207        brkseg = break_seg(p);
4208        stkseg = as_segat(as, prgetstackbase(p));

4210        do {
4211                caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
4212                caddr_t saddr, naddr, baddr;
4213                void *tmp = NULL;
4214                ssize_t psz;
4215                char *parr;
4216                uint64_t npages;
4217                uint64_t pagenum;

4219                /*
4220                 * Segment loop part one: iterate from the base of the segment
4221                 * to its end, pausing at each address boundary (baddr) between
4222                 * ranges that have different virtual memory protections.
```



```
4223                 */
4224                for (saddr = seg->s_base; saddr < eaddr; saddr = baddr) {
4225                        prot = pr_getprot(seg, 0, &tmp, &saddr, &baddr, eaddr);
4226                        ASSERT(baddr >= saddr && baddr <= eaddr);

4228                        /*
4229                         * Segment loop part two: iterate from the current
4230                         * position to the end of the protection boundary,
4231                         * pausing at each address boundary (naddr) between
4232                         * ranges that have different underlying page sizes.
4233                         */
4234                        for (; saddr < baddr; saddr = naddr) {
4235                                psz = pr_getpagesize(seg, saddr, &naddr, baddr);
4236                                ASSERT(naddr >= saddr && naddr <= baddr);

4238                                mp = pr_iol_newbuf(iolhead, sizeof (*mp));

4240                                mp->pr_vaddr = (caddr32_t)(uintptr_t)saddr;
4241                                mp->pr_size = (size32_t)(naddr - saddr);
4242                                mp->pr_offset = segop_getoffset(seg, saddr);
4242                                mp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
4243                                mp->pr_mflags = 0;
4244                                if (prot & PROT_READ)
4245                                        mp->pr_mflags |= MA_READ;
4246                                if (prot & PROT_WRITE)
4247                                        mp->pr_mflags |= MA_WRITE;
4248                                if (prot & PROT_EXEC)
4249                                        mp->pr_mflags |= MA_EXEC;
4250                                if (segop_gettype(seg, saddr) & MAP_SHARED)
4250                                if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
4251                                        mp->pr_mflags |= MA_SHARED;
4252                                if (segop_gettype(seg, saddr) & MAP_NORESERVE)
4252                                if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
4253                                        mp->pr_mflags |= MA_NORESERVE;
4254                                if (seg->s_ops == &segspt_shmops ||
4255                                    (seg->s_ops == &segvn_ops &&
4256                                    (segop_getvp(seg, saddr, &vp) != 0 ||
4256                                    (SEGOP_GETVP(seg, saddr, &vp) != 0 ||
4257                                    vp == NULL)))
4258                                        mp->pr_mflags |= MA_ANON;
4259                                if (seg == brkseg)
4260                                        mp->pr_mflags |= MA_BREAK;
4261                                else if (seg == stkseg)
4262                                        mp->pr_mflags |= MA_STACK;
4263                                if (seg->s_ops == &segspt_shmops)
4264                                        mp->pr_mflags |= MA_ISM | MA_SHM;

4266                                mp->pr_pagesize = PAGESIZE;
4267                                if (psz == -1) {
4268                                        mp->pr_hatpagesize = 0;
4269                                } else {
4270                                        mp->pr_hatpagesize = psz;
4271                                }

4273                                /*
4274                                 * Manufacture a filename for the "object" dir.
4275                                 */
4276                                mp->pr_dev = PRNODEV32;
4277                                vattr.va_mask = AT_FSID|AT_NODEID;
4278                                if (seg->s_ops == &segvn_ops &&
4279                                    segop_getvp(seg, saddr, &vp) == 0 &&
4279                                    SEGOP_GETVP(seg, saddr, &vp) == 0 &&
4280                                    vp != NULL && vp->v_type == VREG &&
4281                                    VOP_GETATTR(vp, &vattr, 0, CRED(),
4282                                    NULL) == 0) {
4283                                        (void) cmpldev(&mp->pr_dev,
```

```
4284                                        vattr.va_fsid);
4285                                mp->pr_ino = vattr.va_nodeid;
4286                                if (vp == p->p_exec)
4287                                        (void) strcpy(mp->pr_mapname,
4288                                            "a.out");
4289                                else
4290                                        pr_object_name(mp->pr_mapname,
4291                                            vp, &vattr);
4292                        }

4294                        /*
4295                         * Get the SysV shared memory id, if any.
4296                         */
4297                        if ((mp->pr_mflags & MA_SHARED) &&
4298                            p->p_segacct && (mp->pr_shmid = shmgetid(p,
4299                            seg->s_base)) != SHMID_NONE) {
4300                                if (mp->pr_shmid == SHMID_FREE)
4301                                        mp->pr_shmid = -1;

4303                                mp->pr_mflags |= MA_SHM;
4304                        } else {
4305                                mp->pr_shmid = -1;
4306                        }

4308                        npages = ((uintptr_t)(naddr - saddr)) >>
4309                            PAGESHIFT;
4310                        parr = kmem_zalloc(npages, KM_SLEEP);

4312                        segop_incore(seg, saddr, naddr - saddr, parr);
4312                        SEGOP_INCORE(seg, saddr, naddr - saddr, parr);

4314                        for (pagenum = 0; pagenum < npages; pagenum++) {
4315                                if (parr[pagenum] & SEG_PAGE_INCORE)
4316                                        mp->pr_rss++;
4317                                if (parr[pagenum] & SEG_PAGE_ANON)
4318                                        mp->pr_anon++;
4319                                if (parr[pagenum] & SEG_PAGE_LOCKED)
4320                                        mp->pr_locked++;
4321                        }
4322                        kmem_free(parr, npages);
4323                }
4324        }
4325        ASSERT(tmp == NULL);
4326    } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

4328    return (0);
4329 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  142900 Fri May  8 18:10:27 2015
new/usr/src/uts/common/fs/proc/prvnops.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

3663 static vnode_t *
3664 pr_lookup_objectdir(vnode_t *dp, char *comp)
3665 {
3666         prnode_t *dpnp = VTOP(dp);
3667         prnode_t *pnp;
3668         proc_t *p;
3669         struct seg *seg;
3670         struct as *as;
3671         vnode_t *vp;
3672         vattr_t vattr;

3674         ASSERT(dpnp->pr_type == PR_OBJECTDIR);

3676         pnp = prgetnode(dp, PR_OBJECT);

3678         if (prlock(dpnp, ZNO) != 0) {
3679                 prfreenode(pnp);
3680                 return (NULL);
3681         }
3682         p = dpnp->pr_common->prc_proc;
3683         if ((p->p_flag & SSYS) || (as = p->p_as) == &kas) {
3684                 prunlock(dpnp);
3685                 prfreenode(pnp);
3686                 return (NULL);
3687         }

3689         /*
3690          * We drop p_lock before grabbing the address space lock
3691          * in order to avoid a deadlock with the clock thread.
3692          * The process will not disappear and its address space
3693          * will not change because it is marked P_PR_LOCK.
3694          */
3695         mutex_exit(&p->p_lock);
3696         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
3697         if ((seg = AS_SEGFIRST(as)) == NULL) {
3698                 vp = NULL;
3699                 goto out;
3700         }
3701         if (strcmp(comp, "a.out") == 0) {
3702                 vp = p->p_exec;
3703                 goto out;
3704         }
3705         do {
3706                 /*
3707                  * Manufacture a filename for the "object" directory.
3708                  */
3709                 vattr.va_mask = AT_FSID|AT_NODEID;
3710                 if (seg->s_ops == &segvn_ops &&
3711                     segop_getvp(seg, seg->s_base, &vp) == 0 &&
3711                     SEGOP_GETVP(seg, seg->s_base, &vp) == 0 &&
3712                     vp != NULL && vp->v_type == VREG &&
3713                     VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
3714                         char name[64];

3716                         if (vp == p->p_exec)    /* "a.out" */
3717                                 continue;
3718                         pr_object_name(name, vp, &vattr);
3719                         if (strcmp(name, comp) == 0)
3720                                 goto out;
```



```
3721                 }
3722         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

3724         vp = NULL;
3725 out:
3726         if (vp != NULL) {
3727                 VN_HOLD(vp);
3728         }
3729         AS_LOCK_EXIT(as, &as->a_lock);
3730         mutex_enter(&p->p_lock);
3731         prunlock(dpnp);

3733         if (vp == NULL)
3734                 prfreenode(pnp);
3735         else {
3736                 /*
3737                  * Fill in the prnode so future references will
3738                  * be able to find the underlying object's vnode.
3739                  * Don't link this prnode into the list of all
3740                  * prnodes for the process; this is a one-use node.
3741                  * Its use is entirely to catch and fail opens for writing.
3742                  */
3743                 pnp->pr_realvp = vp;
3744                 vp = PTOV(pnp);
3745         }

3747         return (vp);
3748 }
______unchanged_portion_omitted_

4051 static vnode_t *
4052 pr_lookup_pathdir(vnode_t *dp, char *comp)
4053 {
4054         prnode_t *dpnp = VTOP(dp);
4055         prnode_t *pnp;
4056         vnode_t *vp = NULL;
4057         proc_t *p;
4058         uint_t fd, flags = 0;
4059         int c;
4060         uf_entry_t *ufp;
4061         uf_info_t *fip;
4062         enum { NAME_FD, NAME_OBJECT, NAME_ROOT, NAME_CWD, NAME_UNKNOWN } type;
4063         char *tmp;
4064         int idx;
4065         struct seg *seg;
4066         struct as *as = NULL;
4067         vattr_t vattr;

4069         ASSERT(dpnp->pr_type == PR_PATHDIR);

4071         /*
4072          * First, check if this is a numeric entry, in which case we have a
4073          * file descriptor.
4074          */
4075         fd = 0;
4076         type = NAME_FD;
4077         tmp = comp;
4078         while ((c = *tmp++) != '\0') {
4079                 int ofd;
4080                 if (c < '0' || c > '9') {
4081                         type = NAME_UNKNOWN;
4082                         break;
4083                 }
4084                 ofd = fd;
4085                 fd = 10*fd + c - '0';
4086                 if (fd/10 != ofd) {     /* integer overflow */
```

```
4087                        type = NAME_UNKNOWN;
4088                        break;
4089                }
4090        }

4092        /*
4093         * Next, see if it is one of the special values {root, cwd}.
4094         */
4095        if (type == NAME_UNKNOWN) {
4096                if (strcmp(comp, "root") == 0)
4097                        type = NAME_ROOT;
4098                else if (strcmp(comp, "cwd") == 0)
4099                        type = NAME_CWD;
4100        }

4102        /*
4103         * Grab the necessary data from the process
4104         */
4105        if (prlock(dpnp, ZNO) != 0)
4106                return (NULL);
4107        p = dpnp->pr_common->prc_proc;

4109        fip = P_FINFO(p);

4111        switch (type) {
4112        case NAME_ROOT:
4113                if ((vp = PTOU(p)->u_rdir) == NULL)
4114                        vp = p->p_zone->zone_rootvp;
4115                VN_HOLD(vp);
4116                break;
4117        case NAME_CWD:
4118                vp = PTOU(p)->u_cdir;
4119                VN_HOLD(vp);
4120                break;
4121        default:
4122                if ((p->p_flag & SSYS) || (as = p->p_as) == &kas) {
4123                        prunlock(dpnp);
4124                        return (NULL);
4125                }
4126        }
4127        mutex_exit(&p->p_lock);

4129        /*
4130         * Determine if this is an object entry
4131         */
4132        if (type == NAME_UNKNOWN) {
4133                /*
4134                 * Start with the inode index immediately after the number of
4135                 * files.
4136                 */
4137                mutex_enter(&fip->fi_lock);
4138                idx = fip->fi_nfiles + 4;
4139                mutex_exit(&fip->fi_lock);

4141                if (strcmp(comp, "a.out") == 0) {
4142                        if (p->p_execdir != NULL) {
4143                                vp = p->p_execdir;
4144                                VN_HOLD(vp);
4145                                type = NAME_OBJECT;
4146                                flags |= PR_AOUT;
4147                        } else {
4148                                vp = p->p_exec;
4149                                VN_HOLD(vp);
4150                                type = NAME_OBJECT;
4151                        }
4152                } else {
```



```
4153                        AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
4154                        if ((seg = AS_SEGFIRST(as)) != NULL) {
4155                                do {
4156                                        /*
4157                                         * Manufacture a filename for the
4158                                         * "object" directory.
4159                                         */
4160                                        vattr.va_mask = AT_FSID|AT_NODEID;
4161                                        if (seg->s_ops == &segvn_ops &&
4162                                            segop_getvp(seg, seg->s_base, &vp)
4162                                            SEGOP_GETVP(seg, seg->s_base, &vp)
4163                                            == 0 &&
4164                                            vp != NULL && vp->v_type == VREG &&
4165                                            VOP_GETATTR(vp, &vattr, 0, CRED(),
4166                                            NULL) == 0) {
4167                                                char name[64];

4169                                                if (vp == p->p_exec)
4170                                                        continue;
4171                                                idx++;
4172                                                pr_object_name(name, vp,
4173                                                    &vattr);
4174                                                if (strcmp(name, comp) == 0)
4175                                                        break;
4176                                        }
4177                                } while ((seg = AS_SEGNEXT(as, seg)) != NULL);
4178                        }

4180                        if (seg == NULL) {
4181                                vp = NULL;
4182                        } else {
4183                                VN_HOLD(vp);
4184                                type = NAME_OBJECT;
4185                        }

4187                        AS_LOCK_EXIT(as, &as->a_lock);
4188                }
4189        }


4192        switch (type) {
4193        case NAME_FD:
4194                mutex_enter(&fip->fi_lock);
4195                if (fd < fip->fi_nfiles) {
4196                        UF_ENTER(ufp, fip, fd);
4197                        if (ufp->uf_file != NULL) {
4198                                vp = ufp->uf_file->f_vnode;
4199                                VN_HOLD(vp);
4200                        }
4201                        UF_EXIT(ufp);
4202                }
4203                mutex_exit(&fip->fi_lock);
4204                idx = fd + 4;
4205                break;
4206        case NAME_ROOT:
4207                idx = 2;
4208                break;
4209        case NAME_CWD:
4210                idx = 3;
4211                break;
4212        case NAME_OBJECT:
4213        case NAME_UNKNOWN:
4214                /* Nothing to do */
4215                break;
4216        }
```

```
4218         mutex_enter(&p->p_lock);
4219         prunlock(dpnp);

4221         if (vp != NULL) {
4222                 pnp = prgetnode(dp, PR_PATH);

4224                 pnp->pr_flags |= flags;
4225                 pnp->pr_common = dpnp->pr_common;
4226                 pnp->pr_pcommon = dpnp->pr_pcommon;
4227                 pnp->pr_realvp = vp;
4228                 pnp->pr_parent = dp;            /* needed for prlookup */
4229                 pnp->pr_ino = pmkino(idx, dpnp->pr_common->prc_slot, PR_PATH);
4230                 VN_HOLD(dp);
4231                 vp = PTOV(pnp);
4232                 vp->v_type = VLNK;
4233         }

4235         return (vp);
4236 }
_____unchanged_portion_omitted_

4829 static void
4830 rebuild_objdir(struct as *as)
4831 {
4832         struct seg *seg;
4833         vnode_t *vp;
4834         vattr_t vattr;
4835         vnode_t **dir;
4836         ulong_t nalloc;
4837         ulong_t nentries;
4838         int i, j;
4839         ulong_t nold, nnew;

4841         ASSERT(AS_WRITE_HELD(as, &as->a_lock));

4843         if (as->a_updatedir == 0 && as->a_objectdir != NULL)
4844                 return;
4845         as->a_updatedir = 0;

4847         if ((nalloc = avl_numnodes(&as->a_segtree)) == 0 ||
4848             (seg = AS_SEGFIRST(as)) == NULL)    /* can't happen? */
4849                 return;

4851         /*
4852          * Allocate space for the new object directory.
4853          * (This is usually about two times too many entries.)
4854          */
4855         nalloc = (nalloc + 0xf) & ~0xf;         /* multiple of 16 */
4856         dir = kmem_zalloc(nalloc * sizeof (vnode_t *), KM_SLEEP);

4858         /* fill in the new directory with desired entries */
4859         nentries = 0;
4860         do {
4861                 vattr.va_mask = AT_FSID|AT_NODEID;
4862                 if (seg->s_ops == &segvn_ops &&
4863                     segop_getvp(seg, seg->s_base, &vp) == 0 &&
4863                     SEGOP_GETVP(seg, seg->s_base, &vp) == 0 &&
4864                     vp != NULL && vp->v_type == VREG &&
4865                     VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
4866                         for (i = 0; i < nentries; i++)
4867                                 if (vp == dir[i])
4868                                         break;
4869                         if (i == nentries) {
4870                                 ASSERT(nentries < nalloc);
4871                                 dir[nentries++] = vp;
4872                         }
```



```
4873                 }
4874         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

4876         if (as->a_objectdir == NULL) {  /* first time */
4877                 as->a_objectdir = dir;
4878                 as->a_sizedir = nalloc;
4879                 return;
4880         }

4882         /*
4883          * Null out all of the defunct entries in the old directory.
4884          */
4885         nold = 0;
4886         nnew = nentries;
4887         for (i = 0; i < as->a_sizedir; i++) {
4888                 if ((vp = as->a_objectdir[i]) != NULL) {
4889                         for (j = 0; j < nentries; j++) {
4890                                 if (vp == dir[j]) {
4891                                         dir[j] = NULL;
4892                                         nnew--;
4893                                         break;
4894                                 }
4895                         }
4896                         if (j == nentries)
4897                                 as->a_objectdir[i] = NULL;
4898                         else
4899                                 nold++;
4900                 }
4901         }

4903         if (nold + nnew > as->a_sizedir) {
4904                 /*
4905                  * Reallocate the old directory to have enough
4906                  * space for the old and new entries combined.
4907                  * Round up to the next multiple of 16.
4908                  */
4909                 ulong_t newsize = (nold + nnew + 0xf) & ~0xf;
4910                 vnode_t **newdir = kmem_zalloc(newsize * sizeof (vnode_t *),
4911                     KM_SLEEP);
4912                 bcopy(as->a_objectdir, newdir,
4913                     as->a_sizedir * sizeof (vnode_t *));
4914                 kmem_free(as->a_objectdir, as->a_sizedir * sizeof (vnode_t *));
4915                 as->a_objectdir = newdir;
4916                 as->a_sizedir = newsize;
4917         }

4919         /*
4920          * Move all new entries to the old directory and
4921          * deallocate the space used by the new directory.
4922          */
4923         if (nnew) {
4924                 for (i = 0, j = 0; i < nentries; i++) {
4925                         if ((vp = dir[i]) == NULL)
4926                                 continue;
4927                         for (; j < as->a_sizedir; j++) {
4928                                 if (as->a_objectdir[j] != NULL)
4929                                         continue;
4930                                 as->a_objectdir[j++] = vp;
4931                                 break;
4932                         }
4933                 }
4934         }
4935         kmem_free(dir, nalloc * sizeof (vnode_t *));
4936 }
______unchanged_portion_omitted_
```

```
**********************************************************
   23668 Fri May  8 18:10:27 2015
new/usr/src/uts/common/io/mem.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

 285 static int
 286 mmpagelock(struct as *as, caddr_t va)
 287 {
 288         struct seg *seg;
 289         int i;

 291         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
 292         seg = as_segat(as, va);
 293         i = (seg != NULL)? segop_capable(seg, S_CAPABILITY_NOMINFLT) : 0;
 293         i = (seg != NULL)? SEGOP_CAPABLE(seg, S_CAPABILITY_NOMINFLT) : 0;
 294         AS_LOCK_EXIT(as, &as->a_lock);

 296         return (i);
 297 }
______unchanged_portion_omitted_
```

```
**********************************************************
   73921 Fri May  8 18:10:28 2015
new/usr/src/uts/common/os/clock.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

 371 /*
 372  * test hook for tod broken detection in tod_validate
 373  */
 374 int tod_unit_test = 0;
 375 time_t tod_test_injector;

 377 #define CLOCK_ADJ_HIST_SIZE     4

 379 static int      adj_hist_entry;

 381 int64_t clock_adj_hist[CLOCK_ADJ_HIST_SIZE];

 383 static void calcloadavg(int, uint64_t *);
 384 static int genloadavg(struct loadavg_s *);
 385 static void loadavg_update();

 387 void (*cmm_clock_callout)() = NULL;
 388 void (*cpucaps_clock_callout)() = NULL;

 390 extern clock_t clock_tick_proc_max;

 392 static int64_t deadman_counter = 0;

 394 static void
 395 clock(void)
 396 {
 397         kthread_t       *t;
 398         uint_t  nrunnable;
 399         uint_t  w_io;
 400         cpu_t   *cp;
 401         cpupart_t *cpupart;
 402         extern  void    set_freemem();
 403         void    (*funcp)();
 404         int32_t ltemp;
 405         int64_t lltemp;
 406         int s;
 407         int do_lgrp_load;
 408         int i;
 409         clock_t now = LBOLT_NO_ACCOUNT; /* current tick */

 411         if (panicstr)
 412                 return;

 414         /*
 415          * Make sure that 'freemem' do not drift too far from the truth
 416          */
 417         set_freemem();


 420         /*
 421          * Before the section which is repeated is executed, we do
 422          * the time delta processing which occurs every clock tick
 423          *
```



```
 424          * There is additional processing which happens every time
 425          * the nanosecond counter rolls over which is described
 426          * below - see the section which begins with : if (one_sec)
 427          *
 428          * This section marks the beginning of the precision-kernel
 429          * code fragment.
 430          *
 431          * First, compute the phase adjustment. If the low-order bits
 432          * (time_phase) of the update overflow, bump the higher order
 433          * bits (time_update).
 434          */
 435         time_phase += time_adj;
 436         if (time_phase <= -FINEUSEC) {
 437                 ltemp = -time_phase / SCALE_PHASE;
 438                 time_phase += ltemp * SCALE_PHASE;
 439                 s = hr_clock_lock();
 440                 timedelta -= ltemp * (NANOSEC/MICROSEC);
 441                 hr_clock_unlock(s);
 442         } else if (time_phase >= FINEUSEC) {
 443                 ltemp = time_phase / SCALE_PHASE;
 444                 time_phase -= ltemp * SCALE_PHASE;
 445                 s = hr_clock_lock();
 446                 timedelta += ltemp * (NANOSEC/MICROSEC);
 447                 hr_clock_unlock(s);
 448         }

 450         /*
 451          * End of precision-kernel code fragment which is processed
 452          * every timer interrupt.
 453          *
 454          * Continue with the interrupt processing as scheduled.
 455          */
 456         /*
 457          * Count the number of runnable threads and the number waiting
 458          * for some form of I/O to complete -- gets added to
 459          * sysinfo.waiting.  To know the state of the system, must add
 460          * wait counts from all CPUs.  Also add up the per-partition
 461          * statistics.
 462          */
 463         w_io = 0;
 464         nrunnable = 0;

 466         /*
 467          * keep track of when to update lgrp/part loads
 468          */

 470         do_lgrp_load = 0;
 471         if (lgrp_ticks++ >= hz / 10) {
 472                 lgrp_ticks = 0;
 473                 do_lgrp_load = 1;
 474         }

 476         if (one_sec) {
 477                 loadavg_update();
 478                 deadman_counter++;
 479         }

 481         /*
 482          * First count the threads waiting on kpreempt queues in each
 483          * CPU partition.
 484          */

 486         cpupart = cp_list_head;
 487         do {
 488                 uint_t cpupart_nrunnable = cpupart->cp_kp_queue.disp_nrunnable;
```

```
490                 cpupart->cp_updates++;
491                 nrunnable += cpupart_nrunnable;
492                 cpupart->cp_nrunnable_cum += cpupart_nrunnable;
493                 if (one_sec) {
494                         cpupart->cp_nrunning = 0;
495                         cpupart->cp_nrunnable = cpupart_nrunnable;
496                 }
497         } while ((cpupart = cpupart->cp_next) != cp_list_head);


500         /* Now count the per-CPU statistics. */
501         cp = cpu_list;
502         do {
503                 uint_t cpu_nrunnable = cp->cpu_disp->disp_nrunnable;

505                 nrunnable += cpu_nrunnable;
506                 cpupart = cp->cpu_part;
507                 cpupart->cp_nrunnable_cum += cpu_nrunnable;
508                 if (one_sec) {
509                         cpupart->cp_nrunnable += cpu_nrunnable;
510                         /*
511                          * Update user, system, and idle cpu times.
512                          */
513                         cpupart->cp_nrunning++;
514                         /*
515                          * w_io is used to update sysinfo.waiting during
516                          * one_second processing below.  Only gather w_io
517                          * information when we walk the list of cpus if we're
518                          * going to perform one_second processing.
519                          */
520                         w_io += CPU_STATS(cp, sys.iowait);
521                 }

523                 if (one_sec && (cp->cpu_flags & CPU_EXISTS)) {
524                         int i, load, change;
525                         hrtime_t intracct, intrused;
526                         const hrtime_t maxnsec = 1000000000;
527                         const int precision = 100;

529                         /*
530                          * Estimate interrupt load on this cpu each second.
531                          * Computes cpu_intrload as %utilization (0-99).
532                          */

534                         /* add up interrupt time from all micro states */
535                         for (intracct = 0, i = 0; i < NCMSTATES; i++)
536                                 intracct += cp->cpu_intracct[i];
537                         scalehrtime(&intracct);

539                         /* compute nsec used in the past second */
540                         intrused = intracct - cp->cpu_intrlast;
541                         cp->cpu_intrlast = intracct;

543                         /* limit the value for safety (and the first pass) */
544                         if (intrused >= maxnsec)
545                                 intrused = maxnsec - 1;

547                         /* calculate %time in interrupt */
548                         load = (precision * intrused) / maxnsec;
549                         ASSERT(load >= 0 && load < precision);
550                         change = cp->cpu_intrload - load;

552                         /* jump to new max, or decay the old max */
553                         if (change < 0)
554                                 cp->cpu_intrload = load;
555                         else if (change > 0)
```



```
556                                 cp->cpu_intrload -= (change + 3) / 4;

558                         DTRACE_PROBE3(cpu_intrload,
559                             cpu_t *, cp,
560                             hrtime_t, intracct,
561                             hrtime_t, intrused);
562                 }

564                 if (do_lgrp_load &&
565                     (cp->cpu_flags & CPU_EXISTS)) {
566                         /*
567                          * When updating the lgroup's load average,
568                          * account for the thread running on the CPU.
569                          * If the CPU is the current one, then we need
570                          * to account for the underlying thread which
571                          * got the clock interrupt not the thread that is
572                          * handling the interrupt and caculating the load
573                          * average
574                          */
575                         t = cp->cpu_thread;
576                         if (CPU == cp)
577                                 t = t->t_intr;

579                         /*
580                          * Account for the load average for this thread if
581                          * it isn't the idle thread or it is on the interrupt
582                          * stack and not the current CPU handling the clock
583                          * interrupt
584                          */
585                         if ((t && t != cp->cpu_idle_thread) || (CPU != cp &&
586                             CPU_ON_INTR(cp))) {
587                                 if (t->t_lpl == cp->cpu_lpl) {
588                                         /* local thread */
589                                         cpu_nrunnable++;
590                                 } else {
591                                         /*
592                                          * This is a remote thread, charge it
593                                          * against its home lgroup.  Note that
594                                          * we notice that a thread is remote
595                                          * only if it's currently executing.
596                                          * This is a reasonable approximation,
597                                          * since queued remote threads are rare.
598                                          * Note also that if we didn't charge
599                                          * it to its home lgroup, remote
600                                          * execution would often make a system
601                                          * appear balanced even though it was
602                                          * not, and thread placement/migration
603                                          * would often not be done correctly.
604                                          */
605                                         lgrp_loadavg(t->t_lpl,
606                                             LGRP_LOADAVG_IN_THREAD_MAX, 0);
607                                 }
608                         }
609                         lgrp_loadavg(cp->cpu_lpl,
610                             cpu_nrunnable * LGRP_LOADAVG_IN_THREAD_MAX, 1);
611                 }
612         } while ((cp = cp->cpu_next) != cpu_list);

614         clock_tick_schedule(one_sec);

616         /*
617          * Check for a callout that needs be called from the clock
618          * thread to support the membership protocol in a clustered
619          * system.  Copy the function pointer so that we can reset
620          * this to NULL if needed.
621          */
```

```
622         if ((funcp = cmm_clock_callout) != NULL)
623                 (*funcp)();
624
625         if ((funcp = cpucaps_clock_callout) != NULL)
626                 (*funcp)();
627
628         /*
629          * Wakeup the cageout thread waiters once per second.
630          */
631         if (one_sec)
632                 kcage_tick();
633
634         if (one_sec) {
635
636                 int drift, absdrift;
637                 timestruc_t tod;
638                 int s;
639
640                 /*
641                  * Beginning of precision-kernel code fragment executed
642                  * every second.
643                  *
644                  * On rollover of the second the phase adjustment to be
645                  * used for the next second is calculated.  Also, the
646                  * maximum error is increased by the tolerance.  If the
647                  * PPS frequency discipline code is present, the phase is
648                  * increased to compensate for the CPU clock oscillator
649                  * frequency error.
650                  *
651                  * On a 32-bit machine and given parameters in the timex.h
652                  * header file, the maximum phase adjustment is +-512 ms
653                  * and maximum frequency offset is (a tad less than)
654                  * +-512 ppm. On a 64-bit machine, you shouldn't need to ask.
655                  */
656                 time_maxerror += time_tolerance / SCALE_USEC;
657
658                 /*
659                  * Leap second processing. If in leap-insert state at
660                  * the end of the day, the system clock is set back one
661                  * second; if in leap-delete state, the system clock is
662                  * set ahead one second. The microtime() routine or
663                  * external clock driver will insure that reported time
664                  * is always monotonic. The ugly divides should be
665                  * replaced.
666                  */
667                 switch (time_state) {
668
669                 case TIME_OK:
670                         if (time_status & STA_INS)
671                                 time_state = TIME_INS;
672                         else if (time_status & STA_DEL)
673                                 time_state = TIME_DEL;
674                         break;
675
676                 case TIME_INS:
677                         if (hrestime.tv_sec % 86400 == 0) {
678                                 s = hr_clock_lock();
679                                 hrestime.tv_sec--;
680                                 hr_clock_unlock(s);
681                                 time_state = TIME_OOP;
682                         }
683                         break;
684
685                 case TIME_DEL:
686                         if ((hrestime.tv_sec + 1) % 86400 == 0) {
687                                 s = hr_clock_lock();
```



```
688                                 hrestime.tv_sec++;
689                                 hr_clock_unlock(s);
690                                 time_state = TIME_WAIT;
691                         }
692                         break;
693
694                 case TIME_OOP:
695                         time_state = TIME_WAIT;
696                         break;
697
698                 case TIME_WAIT:
699                         if (!(time_status & (STA_INS | STA_DEL)))
700                                 time_state = TIME_OK;
701                 default:
702                         break;
703                 }
704
705                 /*
706                  * Compute the phase adjustment for the next second. In
707                  * PLL mode, the offset is reduced by a fixed factor
708                  * times the time constant. In FLL mode the offset is
709                  * used directly. In either mode, the maximum phase
710                  * adjustment for each second is clamped so as to spread
711                  * the adjustment over not more than the number of
712                  * seconds between updates.
713                  */
714                 if (time_offset == 0)
715                         time_adj = 0;
716                 else if (time_offset < 0) {
717                         lltemp = -time_offset;
718                         if (!(time_status & STA_FLL)) {
719                                 if ((1 << time_constant) >= SCALE_KG)
720                                         lltemp *= (1 << time_constant) /
721                                             SCALE_KG;
722                                 else
723                                         lltemp = (lltemp / SCALE_KG) >>
724                                             time_constant;
725                         }
726                         if (lltemp > (MAXPHASE / MINSEC) * SCALE_UPDATE)
727                                 lltemp = (MAXPHASE / MINSEC) * SCALE_UPDATE;
728                         time_offset += lltemp;
729                         time_adj = -(lltemp * SCALE_PHASE) / hz / SCALE_UPDATE;
730                 } else {
731                         lltemp = time_offset;
732                         if (!(time_status & STA_FLL)) {
733                                 if ((1 << time_constant) >= SCALE_KG)
734                                         lltemp *= (1 << time_constant) /
735                                             SCALE_KG;
736                                 else
737                                         lltemp = (lltemp / SCALE_KG) >>
738                                             time_constant;
739                         }
740                         if (lltemp > (MAXPHASE / MINSEC) * SCALE_UPDATE)
741                                 lltemp = (MAXPHASE / MINSEC) * SCALE_UPDATE;
742                         time_offset -= lltemp;
743                         time_adj = (lltemp * SCALE_PHASE) / hz / SCALE_UPDATE;
744                 }
745
746                 /*
747                  * Compute the frequency estimate and additional phase
748                  * adjustment due to frequency error for the next
749                  * second. When the PPS signal is engaged, gnaw on the
750                  * watchdog counter and update the frequency computed by
751                  * the pll and the PPS signal.
752                  */
753                 pps_valid++;
```

```
754                 if (pps_valid == PPS_VALID) {
755                         pps_jitter = MAXTIME;
756                         pps_stabil = MAXFREQ;
757                         time_status &= ~(STA_PPSSIGNAL | STA_PPSJITTER |
758                             STA_PPSWANDER | STA_PPSERROR);
759                 }
760                 lltemp = time_freq + pps_freq;
761
762                 if (lltemp)
763                         time_adj += (lltemp * SCALE_PHASE) / (SCALE_USEC * hz);
764
765                 /*
766                  * End of precision kernel-code fragment
767                  *
768                  * The section below should be modified if we are planning
769                  * to use NTP for synchronization.
770                  *
771                  * Note: the clock synchronization code now assumes
772                  * the following:
773                  *   - if dosynctodr is 1, then compute the drift between
774                  *      the tod chip and software time and adjust one or
775                  *      the other depending on the circumstances
776                  *
777                  *   - if dosynctodr is 0, then the tod chip is independent
778                  *      of the software clock and should not be adjusted,
779                  *      but allowed to free run.  this allows NTP to sync.
780                  *      hrestime without any interference from the tod chip.
781                  */
782
783                 tod_validate_deferred = B_FALSE;
784                 mutex_enter(&tod_lock);
785                 tod = tod_get();
786                 drift = tod.tv_sec - hrestime.tv_sec;
787                 absdrift = (drift >= 0) ? drift : -drift;
788                 if (tod_needsync || absdrift > 1) {
789                         int s;
790                         if (absdrift > 2) {
791                                 if (!tod_broken && tod_faulted == TOD_NOFAULT) {
792                                         s = hr_clock_lock();
793                                         hrestime = tod;
794                                         membar_enter(); /* hrestime visible */
795                                         timedelta = 0;
796                                         timechanged++;
797                                         tod_needsync = 0;
798                                         hr_clock_unlock(s);
799                                         callout_hrestime();
800
801                                 }
802                         } else {
803                                 if (tod_needsync || !dosynctodr) {
804                                         gethrestime(&tod);
805                                         tod_set(tod);
806                                         s = hr_clock_lock();
807                                         if (timedelta == 0)
808                                                 tod_needsync = 0;
809                                         hr_clock_unlock(s);
810                                 } else {
811                                         /*
812                                          * If the drift is 2 seconds on the
813                                          * money, then the TOD is adjusting
814                                          * the clock;  record that.
815                                          */
816                                         clock_adj_hist[adj_hist_entry++ %
817                                             CLOCK_ADJ_HIST_SIZE] = now;
818                                         s = hr_clock_lock();
819                                         timedelta = (int64_t)drift*NANOSEC;
```



```
820                                         hr_clock_unlock(s);
821                                 }
822                         }
823                 }
824                 one_sec = 0;
825                 time = gethrestime_sec();  /* for crusty old kmem readers */
826                 mutex_exit(&tod_lock);
827
828                 /*
829                  * Some drivers still depend on this... XXX
830                  */
831                 cv_broadcast(&lbolt_cv);
832
833                 vminfo.freemem += freemem;
834                 {
835                         pgcnt_t maxswap, resv, free;
836                         pgcnt_t avail =
837                             MAX((spgcnt_t)(availrmem - swapfs_minfree), 0);
838
839                         maxswap = k_anoninfo.ani_mem_resv +
840                             k_anoninfo.ani_max +avail;
841                         /* Update ani_free */
842                         set_anoninfo();
843                         free = k_anoninfo.ani_free + avail;
844                         resv = k_anoninfo.ani_phys_resv +
845                             k_anoninfo.ani_mem_resv;
846
847                         vminfo.swap_resv += resv;
848                         /* number of reserved and allocated pages */
849 #ifdef  DEBUG
850                         if (maxswap < free)
851                                 cmn_err(CE_WARN, "clock: maxswap < free");
852                         if (maxswap < resv)
853                                 cmn_err(CE_WARN, "clock: maxswap < resv");
854 #endif
855                         vminfo.swap_alloc += maxswap - free;
856                         vminfo.swap_avail += maxswap - resv;
857                         vminfo.swap_free += free;
858                 }
859                 vminfo.updates++;
860                 if (nrunnable) {
861                         sysinfo.runque += nrunnable;
862                         sysinfo.runocc++;
863                 }
864                 if (nswapped) {
865                         sysinfo.swpque += nswapped;
866                         sysinfo.swpocc++;
867                 }
868                 sysinfo.waiting += w_io;
869                 sysinfo.updates++;
870
871                 /*
872                  * Wake up fsflush to write out DELWRI
873                  * buffers, dirty pages and other cached
874                  * administrative data, e.g. inodes.
875                  */
876                 if (--fsflushcnt <= 0) {
877                         fsflushcnt = tune.t_fsflushr;
878                         cv_signal(&fsflush_cv);
879                 }
880
881                 vmmeter();
882                 calcloadavg(genloadavg(&loadavg), hp_avenrun);
883                 for (i = 0; i < 3; i++)
884                         /*
885                          * At the moment avenrun[] can only hold 31
```

```
886                          * bits of load average as it is a signed
887                          * int in the API. We need to ensure that
888                          * hp_avenrun[i] >> (16 - FSHIFT) will not be
889                          * too large. If it is, we put the largest value
890                          * that we can use into avenrun[i]. This is
891                          * kludgey, but about all we can do until we
892                          * avenrun[] is declared as an array of uint64[]
893                          */
894                         if (hp_avenrun[i] < ((uint64_t)1<<(31+16-FSHIFT)))
895                                 avenrun[i] = (int32_t)(hp_avenrun[i] >>
896                                     (16 - FSHIFT));
897                         else
898                                 avenrun[i] = 0x7fffffff;

900                 cpupart = cp_list_head;
901                 do {
902                         calcloadavg(genloadavg(&cpupart->cp_loadavg),
903                             cpupart->cp_hp_avenrun);
904                 } while ((cpupart = cpupart->cp_next) != cp_list_head);

906                 /*
907                  * Wake up the swapper thread if necessary.
908                  */
909                 if (runin ||
910                     (runout && (avefree < desfree || wake_sched_sec))) {
911                         t = &t0;
912                         thread_lock(t);
913                         if (t->t_state == TS_STOPPED) {
914                                 runin = runout = 0;
915                                 wake_sched_sec = 0;
916                                 t->t_whystop = 0;
917                                 t->t_whatstop = 0;
918                                 t->t_schedflag &= ~TS_ALLSTART;
919                                 THREAD_TRANSITION(t);
920                                 setfrontdq(t);
921                         }
922                         thread_unlock(t);
923                 }
924         }

926         /*
927          * Wake up the swapper if any high priority swapped-out threads
928          * became runable during the last tick.
929          */
930         if (wake_sched) {
931                 t = &t0;
932                 thread_lock(t);
933                 if (t->t_state == TS_STOPPED) {
934                         runin = runout = 0;
935                         wake_sched = 0;
936                         t->t_whystop = 0;
937                         t->t_whatstop = 0;
938                         t->t_schedflag &= ~TS_ALLSTART;
939                         THREAD_TRANSITION(t);
940                         setfrontdq(t);
941                 }
942                 thread_unlock(t);
905         }
906 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   21374 Fri May  8 18:10:28 2015
new/usr/src/uts/common/os/condvar.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

 182 #define cv_block_sig(t, cvp)    \
 183         { (t)->t_flag |= T_WAKEABLE; cv_block(cvp); }

 185 /*
 186  * Block on the indicated condition variable and release the
 187  * associated kmutex while blocked.
 188  */
 189 void
 190 cv_wait(kcondvar_t *cvp, kmutex_t *mp)
 191 {
 192         if (panicstr)
 193                 return;
 194         ASSERT(!quiesce_active);

 196         ASSERT(curthread->t_schedflag & TS_DONT_SWAP);
 196         thread_lock(curthread);                 /* lock the thread */
 197         cv_block((condvar_impl_t *)cvp);
 198         thread_unlock_nopreempt(curthread);     /* unlock the waiters field */
 199         mutex_exit(mp);
 200         swtch();
 201         mutex_enter(mp);
 202 }
______unchanged_portion_omitted_

 303 int
 304 cv_wait_sig(kcondvar_t *cvp, kmutex_t *mp)
 305 {
 306         kthread_t *t = curthread;
 307         proc_t *p = ttoproc(t);
 308         klwp_t *lwp = ttolwp(t);
 309         int cancel_pending;
 310         int rval = 1;
 311         int signalled = 0;

 313         if (panicstr)
 314                 return (rval);
 315         ASSERT(!quiesce_active);

 317         /*
 318          * Threads in system processes don't process signals.  This is
 319          * true both for standard threads of system processes and for
 320          * interrupt threads which have borrowed their pinned thread's LWP.
 321          */
 322         if (lwp == NULL || (p->p_flag & SSYS)) {
 323                 cv_wait(cvp, mp);
 324                 return (rval);
 325         }
 326         ASSERT(t->t_intr == NULL);

 329         ASSERT(curthread->t_schedflag & TS_DONT_SWAP);
 328         cancel_pending = schedctl_cancel_pending();
 329         lwp->lwp_asleep = 1;
 330         lwp->lwp_sysabort = 0;
```



```
 331         thread_lock(t);
 332         cv_block_sig(t, (condvar_impl_t *)cvp);
 333         thread_unlock_nopreempt(t);
 334         mutex_exit(mp);
 335         if (ISSIG(t, JUSTLOOKING) || MUSTRETURN(p, t) || cancel_pending)
 336                 setrun(t);
 337         /* ASSERT(no locks are held) */
 338         swtch();
 339         signalled = (t->t_schedflag & TS_SIGNALLED);
 340         t->t_flag &= ~T_WAKEABLE;
 341         mutex_enter(mp);
 342         if (ISSIG_PENDING(t, lwp, p)) {
 343                 mutex_exit(mp);
 344                 if (issig(FORREAL))
 345                         rval = 0;
 346                 mutex_enter(mp);
 347         }
 348         if (lwp->lwp_sysabort || MUSTRETURN(p, t))
 349                 rval = 0;
 350         if (rval != 0 && cancel_pending) {
 351                 schedctl_cancel_eintr();
 352                 rval = 0;
 353         }
 354         lwp->lwp_asleep = 0;
 355         lwp->lwp_sysabort = 0;
 356         if (rval == 0 && signalled)     /* avoid consuming the cv_signal() */
 357                 cv_signal(cvp);
 358         return (rval);
 359 }
______unchanged_portion_omitted_

 517 /*
 518  * Like cv_wait_sig_swap but allows the caller to indicate (with a
 519  * non-NULL sigret) that they will take care of signalling the cv
 520  * after wakeup, if necessary.  This is a vile hack that should only
 521  * be used when no other option is available; almost all callers
 522  * should just use cv_wait_sig_swap (which takes care of the cv_signal
 523  * stuff automatically) instead.
 524  */
 525 int
 526 cv_wait_sig_swap_core(kcondvar_t *cvp, kmutex_t *mp, int *sigret)
 527 {
 528         kthread_t *t = curthread;
 529         proc_t *p = ttoproc(t);
 530         klwp_t *lwp = ttolwp(t);
 531         int cancel_pending;
 532         int rval = 1;
 533         int signalled = 0;

 535         if (panicstr)
 536                 return (rval);

 538         /*
 539          * Threads in system processes don't process signals.  This is
 540          * true both for standard threads of system processes and for
 541          * interrupt threads which have borrowed their pinned thread's LWP.
 542          */
 543         if (lwp == NULL || (p->p_flag & SSYS)) {
 544                 cv_wait(cvp, mp);
 545                 return (rval);
 546         }
 547         ASSERT(t->t_intr == NULL);

 549         cancel_pending = schedctl_cancel_pending();
 550         lwp->lwp_asleep = 1;
 551         lwp->lwp_sysabort = 0;
```

```
552         thread_lock(t);
553         t->t_kpri_req = 0;      /* don't need kernel priority */
554         cv_block_sig(t, (condvar_impl_t *)cvp);
557         /* I can be swapped now */
558         curthread->t_schedflag &= ~TS_DONT_SWAP;
555         thread_unlock_nopreempt(t);
556         mutex_exit(mp);
557         if (ISSIG(t, JUSTLOOKING) || MUSTRETURN(p, t) || cancel_pending)
558                 setrun(t);
559         /* ASSERT(no locks are held) */
560         swtch();
561         signalled = (t->t_schedflag & TS_SIGNALLED);
562         t->t_flag &= ~T_WAKEABLE;
567         /* TS_DONT_SWAP set by disp() */
568         ASSERT(curthread->t_schedflag & TS_DONT_SWAP);
563         mutex_enter(mp);
564         if (ISSIG_PENDING(t, lwp, p)) {
565                 mutex_exit(mp);
566                 if (issig(FORREAL))
567                         rval = 0;
568                 mutex_enter(mp);
569         }
570         if (lwp->lwp_sysabort || MUSTRETURN(p, t))
571                 rval = 0;
572         if (rval != 0 && cancel_pending) {
573                 schedctl_cancel_eintr();
574                 rval = 0;
575         }
576         lwp->lwp_asleep = 0;
577         lwp->lwp_sysabort = 0;
578         if (rval == 0) {
579                 if (sigret != NULL)
580                         *sigret = signalled;    /* just tell the caller */
581                 else if (signalled)
582                         cv_signal(cvp); /* avoid consuming the cv_signal() */
583         }
584         return (rval);
585 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   93874 Fri May  8 18:10:28 2015
new/usr/src/uts/common/os/cpu.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

 297 static struct cpu_vm_stats_ks_data {
 298         kstat_named_t pgrec;
 299         kstat_named_t pgfrec;
 300         kstat_named_t pgin;
 301         kstat_named_t pgpgin;
 302         kstat_named_t pgout;
 303         kstat_named_t pgpgout;
 304         kstat_named_t swapin;
 305         kstat_named_t pgswapin;
 306         kstat_named_t swapout;
 307         kstat_named_t pgswapout;
 304         kstat_named_t zfod;
 305         kstat_named_t dfree;
 306         kstat_named_t scan;
 307         kstat_named_t rev;
 308         kstat_named_t hat_fault;
 309         kstat_named_t as_fault;
 310         kstat_named_t maj_fault;
 311         kstat_named_t cow_fault;
 312         kstat_named_t prot_fault;
 313         kstat_named_t softlock;
 314         kstat_named_t kernel_asflt;
 315         kstat_named_t pgrrun;
 316         kstat_named_t execpgin;
 317         kstat_named_t execpgout;
 318         kstat_named_t execfree;
 319         kstat_named_t anonpgin;
 320         kstat_named_t anonpgout;
 321         kstat_named_t anonfree;
 322         kstat_named_t fspgin;
 323         kstat_named_t fspgout;
 324         kstat_named_t fsfree;
 325 } cpu_vm_stats_ks_data_template = {
 326         { "pgrec",              KSTAT_DATA_UINT64 },
 327         { "pgfrec",             KSTAT_DATA_UINT64 },
 328         { "pgin",               KSTAT_DATA_UINT64 },
 329         { "pgpgin",             KSTAT_DATA_UINT64 },
 330         { "pgout",              KSTAT_DATA_UINT64 },
 331         { "pgpgout",            KSTAT_DATA_UINT64 },
 336         { "swapin",             KSTAT_DATA_UINT64 },
 337         { "pgswapin",           KSTAT_DATA_UINT64 },
 338         { "swapout",            KSTAT_DATA_UINT64 },
 339         { "pgswapout",          KSTAT_DATA_UINT64 },
 332         { "zfod",               KSTAT_DATA_UINT64 },
 333         { "dfree",              KSTAT_DATA_UINT64 },
 334         { "scan",               KSTAT_DATA_UINT64 },
 335         { "rev",                KSTAT_DATA_UINT64 },
 336         { "hat_fault",          KSTAT_DATA_UINT64 },
 337         { "as_fault",           KSTAT_DATA_UINT64 },
 338         { "maj_fault",          KSTAT_DATA_UINT64 },
 339         { "cow_fault",          KSTAT_DATA_UINT64 },
 340         { "prot_fault",         KSTAT_DATA_UINT64 },
 341         { "softlock",           KSTAT_DATA_UINT64 },
```



```
 342         { "kernel_asflt",       KSTAT_DATA_UINT64 },
 343         { "pgrrun",             KSTAT_DATA_UINT64 },
 344         { "execpgin",           KSTAT_DATA_UINT64 },
 345         { "execpgout",          KSTAT_DATA_UINT64 },
 346         { "execfree",           KSTAT_DATA_UINT64 },
 347         { "anonpgin",           KSTAT_DATA_UINT64 },
 348         { "anonpgout",          KSTAT_DATA_UINT64 },
 349         { "anonfree",           KSTAT_DATA_UINT64 },
 350         { "fspgin",             KSTAT_DATA_UINT64 },
 351         { "fspgout",            KSTAT_DATA_UINT64 },
 352         { "fsfree",             KSTAT_DATA_UINT64 },
 353 };
______unchanged_portion_omitted_

2515 /*
2516  * Bind a thread to a CPU as requested.
2517  */
2518 int
2519 cpu_bind_thread(kthread_id_t tp, processorid_t bind, processorid_t *obind,
2520     int *error)
2521 {
2522         processorid_t   binding;
2523         cpu_t           *cp = NULL;

2525         ASSERT(MUTEX_HELD(&cpu_lock));
2526         ASSERT(MUTEX_HELD(&ttoproc(tp)->p_lock));

2528         thread_lock(tp);

2530         /*
2531          * Record old binding, but change the obind, which was initialized
2532          * to PBIND_NONE, only if this thread has a binding.  This avoids
2533          * reporting PBIND_NONE for a process when some LWPs are bound.
2534          */
2535         binding = tp->t_bind_cpu;
2536         if (binding != PBIND_NONE)
2537                 *obind = binding;       /* record old binding */

2539         switch (bind) {
2540         case PBIND_QUERY:
2541                 /* Just return the old binding */
2542                 thread_unlock(tp);
2543                 return (0);

2545         case PBIND_QUERY_TYPE:
2546                 /* Return the binding type */
2547                 *obind = TB_CPU_IS_SOFT(tp) ? PBIND_SOFT : PBIND_HARD;
2548                 thread_unlock(tp);
2549                 return (0);

2551         case PBIND_SOFT:
2552                 /*
2553                  *  Set soft binding for this thread and return the actual
2554                  *  binding
2555                  */
2556                 TB_CPU_SOFT_SET(tp);
2557                 thread_unlock(tp);
2558                 return (0);

2560         case PBIND_HARD:
2561                 /*
2562                  *  Set hard binding for this thread and return the actual
2563                  *  binding
2564                  */
2565                 TB_CPU_HARD_SET(tp);
2566                 thread_unlock(tp);
```

```
2567                return (0);

2569        default:
2570                break;
2571        }

2573        /*
2574         * If this thread/LWP cannot be bound because of permission
2575         * problems, just note that and return success so that the
2576         * other threads/LWPs will be bound.  This is the way
2577         * processor_bind() is defined to work.
2578         *
2579         * Binding will get EPERM if the thread is of system class
2580         * or hasprocperm() fails.
2581         */
2582        if (tp->t_cid == 0 || !hasprocperm(tp->t_cred, CRED())) {
2583                *error = EPERM;
2584                thread_unlock(tp);
2585                return (0);
2586        }

2588        binding = bind;
2589        if (binding != PBIND_NONE) {
2590                cp = cpu_get((processorid_t)binding);
2591                /*
2592                 * Make sure binding is valid and is in right partition.
2593                 */
2594                if (cp == NULL || tp->t_cpupart != cp->cpu_part) {
2595                        *error = EINVAL;
2596                        thread_unlock(tp);
2597                        return (0);
2598                }
2599        }
2600        tp->t_bind_cpu = binding;       /* set new binding */

2602        /*
2603         * If there is no system-set reason for affinity, set
2604         * the t_bound_cpu field to reflect the binding.
2605         */
2606        if (tp->t_affinitycnt == 0) {
2607                if (binding == PBIND_NONE) {
2608                        /*
2609                         * We may need to adjust disp_max_unbound_pri
2610                         * since we're becoming unbound.
2611                         */
2612                        disp_adjust_unbound_pri(tp);

2614                        tp->t_bound_cpu = NULL; /* set new binding */

2616                        /*
2617                         * Move thread to lgroup with strongest affinity
2618                         * after unbinding
2619                         */
2620                        if (tp->t_lgrp_affinity)
2621                                lgrp_move_thread(tp,
2622                                    lgrp_choose(tp, tp->t_cpupart), 1);

2624                        if (tp->t_state == TS_ONPROC &&
2625                            tp->t_cpu->cpu_part != tp->t_cpupart)
2626                                cpu_surrender(tp);
2627                } else {
2628                        lpl_t   *lpl;

2630                        tp->t_bound_cpu = cp;
2631                        ASSERT(cp->cpu_lpl != NULL);
```



```
2633                        /*
2634                         * Set home to lgroup with most affinity containing CPU
2635                         * that thread is being bound or minimum bounding
2636                         * lgroup if no affinities set
2637                         */
2638                        if (tp->t_lgrp_affinity)
2639                                lpl = lgrp_affinity_best(tp, tp->t_cpupart,
2640                                    LGRP_NONE, B_FALSE);
2641                        else
2642                                lpl = cp->cpu_lpl;

2644                        if (tp->t_lpl != lpl) {
2645                                /* can't grab cpu_lock */
2646                                lgrp_move_thread(tp, lpl, 1);
2647                        }

2649                        /*
2650                         * Make the thread switch to the bound CPU.
2651                         * If the thread is runnable, we need to
2652                         * requeue it even if t_cpu is already set
2653                         * to the right CPU, since it may be on a
2654                         * kpreempt queue and need to move to a local
2655                         * queue.  We could check t_disp_queue to
2656                         * avoid unnecessary overhead if it's already
2657                         * on the right queue, but since this isn't
2658                         * a performance-critical operation it doesn't
2659                         * seem worth the extra code and complexity.
2660                         *
2661                         * If the thread is weakbound to the cpu then it will
2662                         * resist the new binding request until the weak
2663                         * binding drops.  The cpu_surrender or requeueing
2664                         * below could be skipped in such cases (since it
2665                         * will have no effect), but that would require
2666                         * thread_allowmigrate to acquire thread_lock so
2667                         * we'll take the very occasional hit here instead.
2668                         */
2669                        if (tp->t_state == TS_ONPROC) {
2670                                cpu_surrender(tp);
2671                        } else if (tp->t_state == TS_RUN) {
2672                                cpu_t *ocp = tp->t_cpu;

2674                                (void) dispdeq(tp);
2675                                setbackdq(tp);
2676                                /*
2677                                 * On the bound CPU's disp queue now.
2685                                 * Either on the bound CPU's disp queue now,
2686                                 * or swapped out or on the swap queue.
2678                                 */
2679                                ASSERT(tp->t_disp_queue == cp->cpu_disp ||
2680                                    tp->t_weakbound_cpu == ocp);
2689                                    tp->t_weakbound_cpu == ocp ||
2690                                    (tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ))
2691                                    != TS_LOAD);
2681                        }
2682                }
2683        }

2685        /*
2686         * Our binding has changed; set TP_CHANGEBIND.
2687         */
2688        tp->t_proc_flag |= TP_CHANGEBIND;
2689        aston(tp);

2691        thread_unlock(tp);

2693        return (0);
```

```
2694 }
_____unchanged_portion_omitted_

3262 static int
3263 cpu_vm_stats_ks_update(kstat_t *ksp, int rw)
3264 {
3265         cpu_t *cp = (cpu_t *)ksp->ks_private;
3266         struct cpu_vm_stats_ks_data *cvskd;
3267         cpu_vm_stats_t *cvs;

3269         if (rw == KSTAT_WRITE)
3270                 return (EACCES);

3272         cvs = &cp->cpu_stats.vm;
3273         cvskd = ksp->ks_data;

3275         bcopy(&cpu_vm_stats_ks_data_template, ksp->ks_data,
3276             sizeof (cpu_vm_stats_ks_data_template));
3277         cvskd->pgrec.value.ui64 = cvs->pgrec;
3278         cvskd->pgfrec.value.ui64 = cvs->pgfrec;
3279         cvskd->pgin.value.ui64 = cvs->pgin;
3280         cvskd->pgpgin.value.ui64 = cvs->pgpgin;
3281         cvskd->pgout.value.ui64 = cvs->pgout;
3282         cvskd->pgpgout.value.ui64 = cvs->pgpgout;
3294         cvskd->swapin.value.ui64 = cvs->swapin;
3295         cvskd->pgswapin.value.ui64 = cvs->pgswapin;
3296         cvskd->swapout.value.ui64 = cvs->swapout;
3297         cvskd->pgswapout.value.ui64 = cvs->pgswapout;
3283         cvskd->zfod.value.ui64 = cvs->zfod;
3284         cvskd->dfree.value.ui64 = cvs->dfree;
3285         cvskd->scan.value.ui64 = cvs->scan;
3286         cvskd->rev.value.ui64 = cvs->rev;
3287         cvskd->hat_fault.value.ui64 = cvs->hat_fault;
3288         cvskd->as_fault.value.ui64 = cvs->as_fault;
3289         cvskd->maj_fault.value.ui64 = cvs->maj_fault;
3290         cvskd->cow_fault.value.ui64 = cvs->cow_fault;
3291         cvskd->prot_fault.value.ui64 = cvs->prot_fault;
3292         cvskd->softlock.value.ui64 = cvs->softlock;
3293         cvskd->kernel_asflt.value.ui64 = cvs->kernel_asflt;
3294         cvskd->pgrrun.value.ui64 = cvs->pgrrun;
3295         cvskd->execpgin.value.ui64 = cvs->execpgin;
3296         cvskd->execpgout.value.ui64 = cvs->execpgout;
3297         cvskd->execfree.value.ui64 = cvs->execfree;
3298         cvskd->anonpgin.value.ui64 = cvs->anonpgin;
3299         cvskd->anonpgout.value.ui64 = cvs->anonpgout;
3300         cvskd->anonfree.value.ui64 = cvs->anonfree;
3301         cvskd->fspgin.value.ui64 = cvs->fspgin;
3302         cvskd->fspgout.value.ui64 = cvs->fspgout;
3303         cvskd->fsfree.value.ui64 = cvs->fsfree;

3305         return (0);
3306 }

3308 static int
3309 cpu_stat_ks_update(kstat_t *ksp, int rw)
3310 {
3311         cpu_stat_t *cso;
3312         cpu_t *cp;
3313         int i;
3314         hrtime_t msnsecs[NCMSTATES];

3316         cso = (cpu_stat_t *)ksp->ks_data;
3317         cp = (cpu_t *)ksp->ks_private;

3319         if (rw == KSTAT_WRITE)
3320                 return (EACCES);
```



```
3322         /*
3323          * Read CPU mstate, but compare with the last values we
3324          * received to make sure that the returned kstats never
3325          * decrease.
3326          */

3328         get_cpu_mstate(cp, msnsecs);
3329         msnsecs[CMS_IDLE] = NSEC_TO_TICK(msnsecs[CMS_IDLE]);
3330         msnsecs[CMS_USER] = NSEC_TO_TICK(msnsecs[CMS_USER]);
3331         msnsecs[CMS_SYSTEM] = NSEC_TO_TICK(msnsecs[CMS_SYSTEM]);
3332         if (cso->cpu_sysinfo.cpu[CPU_IDLE] < msnsecs[CMS_IDLE])
3333                 cso->cpu_sysinfo.cpu[CPU_IDLE] = msnsecs[CMS_IDLE];
3334         if (cso->cpu_sysinfo.cpu[CPU_USER] < msnsecs[CMS_USER])
3335                 cso->cpu_sysinfo.cpu[CPU_USER] = msnsecs[CMS_USER];
3336         if (cso->cpu_sysinfo.cpu[CPU_KERNEL] < msnsecs[CMS_SYSTEM])
3337                 cso->cpu_sysinfo.cpu[CPU_KERNEL] = msnsecs[CMS_SYSTEM];
3338         cso->cpu_sysinfo.cpu[CPU_WAIT]  = 0;
3339         cso->cpu_sysinfo.wait[W_IO]     = 0;
3340         cso->cpu_sysinfo.wait[W_SWAP]   = 0;
3341         cso->cpu_sysinfo.wait[W_PIO]    = 0;
3342         cso->cpu_sysinfo.bread          = CPU_STATS(cp, sys.bread);
3343         cso->cpu_sysinfo.bwrite         = CPU_STATS(cp, sys.bwrite);
3344         cso->cpu_sysinfo.lread          = CPU_STATS(cp, sys.lread);
3345         cso->cpu_sysinfo.lwrite         = CPU_STATS(cp, sys.lwrite);
3346         cso->cpu_sysinfo.phread         = CPU_STATS(cp, sys.phread);
3347         cso->cpu_sysinfo.phwrite        = CPU_STATS(cp, sys.phwrite);
3348         cso->cpu_sysinfo.pswitch        = CPU_STATS(cp, sys.pswitch);
3349         cso->cpu_sysinfo.trap           = CPU_STATS(cp, sys.trap);
3350         cso->cpu_sysinfo.intr           = 0;
3351         for (i = 0; i < PIL_MAX; i++)
3352                 cso->cpu_sysinfo.intr += CPU_STATS(cp, sys.intr[i]);
3353         cso->cpu_sysinfo.syscall        = CPU_STATS(cp, sys.syscall);
3354         cso->cpu_sysinfo.sysread        = CPU_STATS(cp, sys.sysread);
3355         cso->cpu_sysinfo.syswrite       = CPU_STATS(cp, sys.syswrite);
3356         cso->cpu_sysinfo.sysfork        = CPU_STATS(cp, sys.sysfork);
3357         cso->cpu_sysinfo.sysvfork       = CPU_STATS(cp, sys.sysvfork);
3358         cso->cpu_sysinfo.sysexec        = CPU_STATS(cp, sys.sysexec);
3359         cso->cpu_sysinfo.readch         = CPU_STATS(cp, sys.readch);
3360         cso->cpu_sysinfo.writech        = CPU_STATS(cp, sys.writech);
3361         cso->cpu_sysinfo.rcvint         = CPU_STATS(cp, sys.rcvint);
3362         cso->cpu_sysinfo.xmtint         = CPU_STATS(cp, sys.xmtint);
3363         cso->cpu_sysinfo.mdmint         = CPU_STATS(cp, sys.mdmint);
3364         cso->cpu_sysinfo.rawch          = CPU_STATS(cp, sys.rawch);
3365         cso->cpu_sysinfo.canch          = CPU_STATS(cp, sys.canch);
3366         cso->cpu_sysinfo.outch          = CPU_STATS(cp, sys.outch);
3367         cso->cpu_sysinfo.msg            = CPU_STATS(cp, sys.msg);
3368         cso->cpu_sysinfo.sema           = CPU_STATS(cp, sys.sema);
3369         cso->cpu_sysinfo.namei          = CPU_STATS(cp, sys.namei);
3370         cso->cpu_sysinfo.ufsiget        = CPU_STATS(cp, sys.ufsiget);
3371         cso->cpu_sysinfo.ufsdirblk      = CPU_STATS(cp, sys.ufsdirblk);
3372         cso->cpu_sysinfo.ufsipage       = CPU_STATS(cp, sys.ufsipage);
3373         cso->cpu_sysinfo.ufsinopage     = CPU_STATS(cp, sys.ufsinopage);
3374         cso->cpu_sysinfo.inodeovf       = 0;
3375         cso->cpu_sysinfo.fileovf        = 0;
3376         cso->cpu_sysinfo.procovf        = CPU_STATS(cp, sys.procovf);
3377         cso->cpu_sysinfo.intrthread     = 0;
3378         for (i = 0; i < LOCK_LEVEL - 1; i++)
3379                 cso->cpu_sysinfo.intrthread += CPU_STATS(cp, sys.intr[i]);
3380         cso->cpu_sysinfo.intrblk        = CPU_STATS(cp, sys.intrblk);
3381         cso->cpu_sysinfo.idlethread     = CPU_STATS(cp, sys.idlethread);
3382         cso->cpu_sysinfo.inv_swtch      = CPU_STATS(cp, sys.inv_swtch);
3383         cso->cpu_sysinfo.nthreads       = CPU_STATS(cp, sys.nthreads);
3384         cso->cpu_sysinfo.cpumigrate     = CPU_STATS(cp, sys.cpumigrate);
3385         cso->cpu_sysinfo.xcalls         = CPU_STATS(cp, sys.xcalls);
3386         cso->cpu_sysinfo.mutex_adenters = CPU_STATS(cp, sys.mutex_adenters);
```

```
3387         cso->cpu_sysinfo.rw_rdfails     = CPU_STATS(cp, sys.rw_rdfails);
3388         cso->cpu_sysinfo.rw_wrfails     = CPU_STATS(cp, sys.rw_wrfails);
3389         cso->cpu_sysinfo.modload        = CPU_STATS(cp, sys.modload);
3390         cso->cpu_sysinfo.modunload      = CPU_STATS(cp, sys.modunload);
3391         cso->cpu_sysinfo.bawrite        = CPU_STATS(cp, sys.bawrite);
3392         cso->cpu_sysinfo.rw_enters      = 0;
3393         cso->cpu_sysinfo.win_uo_cnt     = 0;
3394         cso->cpu_sysinfo.win_uu_cnt     = 0;
3395         cso->cpu_sysinfo.win_so_cnt     = 0;
3396         cso->cpu_sysinfo.win_su_cnt     = 0;
3397         cso->cpu_sysinfo.win_suo_cnt    = 0;

3399         cso->cpu_syswait.iowait         = CPU_STATS(cp, sys.iowait);
3400         cso->cpu_syswait.swap           = 0;
3401         cso->cpu_syswait.physio         = 0;

3403         cso->cpu_vminfo.pgrec           = CPU_STATS(cp, vm.pgrec);
3404         cso->cpu_vminfo.pgfrec          = CPU_STATS(cp, vm.pgfrec);
3405         cso->cpu_vminfo.pgin            = CPU_STATS(cp, vm.pgin);
3406         cso->cpu_vminfo.pgpgin          = CPU_STATS(cp, vm.pgpgin);
3407         cso->cpu_vminfo.pgout           = CPU_STATS(cp, vm.pgout);
3408         cso->cpu_vminfo.pgpgout         = CPU_STATS(cp, vm.pgpgout);
3424         cso->cpu_vminfo.swapin          = CPU_STATS(cp, vm.swapin);
3425         cso->cpu_vminfo.pgswapin        = CPU_STATS(cp, vm.pgswapin);
3426         cso->cpu_vminfo.swapout         = CPU_STATS(cp, vm.swapout);
3427         cso->cpu_vminfo.pgswapout       = CPU_STATS(cp, vm.pgswapout);
3409         cso->cpu_vminfo.zfod            = CPU_STATS(cp, vm.zfod);
3410         cso->cpu_vminfo.dfree           = CPU_STATS(cp, vm.dfree);
3411         cso->cpu_vminfo.scan            = CPU_STATS(cp, vm.scan);
3412         cso->cpu_vminfo.rev             = CPU_STATS(cp, vm.rev);
3413         cso->cpu_vminfo.hat_fault       = CPU_STATS(cp, vm.hat_fault);
3414         cso->cpu_vminfo.as_fault        = CPU_STATS(cp, vm.as_fault);
3415         cso->cpu_vminfo.maj_fault       = CPU_STATS(cp, vm.maj_fault);
3416         cso->cpu_vminfo.cow_fault       = CPU_STATS(cp, vm.cow_fault);
3417         cso->cpu_vminfo.prot_fault      = CPU_STATS(cp, vm.prot_fault);
3418         cso->cpu_vminfo.softlock        = CPU_STATS(cp, vm.softlock);
3419         cso->cpu_vminfo.kernel_asflt    = CPU_STATS(cp, vm.kernel_asflt);
3420         cso->cpu_vminfo.pgrrun          = CPU_STATS(cp, vm.pgrrun);
3421         cso->cpu_vminfo.execpgin        = CPU_STATS(cp, vm.execpgin);
3422         cso->cpu_vminfo.execpgout       = CPU_STATS(cp, vm.execpgout);
3423         cso->cpu_vminfo.execfree        = CPU_STATS(cp, vm.execfree);
3424         cso->cpu_vminfo.anonpgin        = CPU_STATS(cp, vm.anonpgin);
3425         cso->cpu_vminfo.anonpgout       = CPU_STATS(cp, vm.anonpgout);
3426         cso->cpu_vminfo.anonfree        = CPU_STATS(cp, vm.anonfree);
3427         cso->cpu_vminfo.fspgin          = CPU_STATS(cp, vm.fspgin);
3428         cso->cpu_vminfo.fspgout         = CPU_STATS(cp, vm.fspgout);
3429         cso->cpu_vminfo.fsfree          = CPU_STATS(cp, vm.fsfree);

3431         return (0);
3432 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   35777 Fri May  8 18:10:28 2015
new/usr/src/uts/common/os/dumpsubr.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

 661 /*
 662  * Dump the <as, va, pfn> information for a given address space.
 663  * segop_dump() will call dump_addpage() for each page in the segment.
 663  * SEGOP_DUMP() will call dump_addpage() for each page in the segment.
 664  */
 665 static void
 666 dump_as(struct as *as)
 667 {
 668         struct seg *seg;

 670         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
 671         for (seg = AS_SEGFIRST(as); seg; seg = AS_SEGNEXT(as, seg)) {
 672                 if (seg->s_as != as)
 673                         break;
 674                 if (seg->s_ops == NULL)
 675                         continue;
 676                 segop_dump(seg);
 676                 SEGOP_DUMP(seg);
 677         }
 678         AS_LOCK_EXIT(as, &as->a_lock);

 680         if (seg != NULL)
 681                 cmn_err(CE_WARN, "invalid segment %p in address space %p",
 682                     (void *)seg, (void *)as);
 683 }
______unchanged_portion_omitted_
```

```
**********************************************************
   52326 Fri May  8 18:10:29 2015
new/usr/src/uts/common/os/exec.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

1144 /*
1145  * Map a section of an executable file into the user's
1146  * address space.
1147  */
1148 int
1149 execmap(struct vnode *vp, caddr_t addr, size_t len, size_t zfodlen,
1150     off_t offset, int prot, int page, uint_t szc)
1151 {
1152         int error = 0;
1153         off_t oldoffset;
1154         caddr_t zfodbase, oldaddr;
1155         size_t end, oldlen;
1156         size_t zfoddiff;
1157         label_t ljb;
1158         proc_t *p = ttoproc(curthread);

1160         oldaddr = addr;
1161         addr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1162         if (len) {
1163                 oldlen = len;
1164                 len += ((size_t)oldaddr - (size_t)addr);
1165                 oldoffset = offset;
1166                 offset = (off_t)((uintptr_t)offset & PAGEMASK);
1167                 if (page) {
1168                         spgcnt_t  prefltmem, availm, npages;
1169                         int preread;
1170                         uint_t mflag = MAP_PRIVATE | MAP_FIXED;

1172                         if ((prot & (PROT_WRITE | PROT_EXEC)) == PROT_EXEC) {
1173                                 mflag |= MAP_TEXT;
1174                         } else {
1175                                 mflag |= MAP_INITDATA;
1176                         }

1178                         if (valid_usr_range(addr, len, prot, p->p_as,
1179                             p->p_as->a_userlimit) != RANGE_OKAY) {
1180                                 error = ENOMEM;
1181                                 goto bad;
1182                         }
1183                         if (error = VOP_MAP(vp, (offset_t)offset,
1184                             p->p_as, &addr, len, prot, PROT_ALL,
1185                             mflag, CRED(), NULL))
1186                                 goto bad;

1188                         /*
1189                          * If the segment can fit, then we prefault
1190                          * the entire segment in.  This is based on the
1191                          * model that says the best working set of a
1192                          * small program is all of its pages.
1193                          */
1194                         npages = (spgcnt_t)btopr(len);
1195                         prefltmem = freemem - desfree;
1196                         preread =
1197                             (npages < prefltmem && len < PGTHRESH) ? 1 : 0;

1199                         /*
1200                          * If we aren't prefaulting the segment,
1201                          * increment "deficit", if necessary to ensure
1202                          * that pages will become available when this
```



```
1203                          * process starts executing.
1204                          */
1205                         availm = freemem - lotsfree;
1206                         if (preread == 0 && npages > availm &&
1207                             deficit < lotsfree) {
1208                                 deficit += MIN((pgcnt_t)(npages - availm),
1209                                     lotsfree - deficit);
1210                         }

1212                         if (preread) {
1213                                 TRACE_2(TR_FAC_PROC, TR_EXECMAP_PREREAD,
1214                                     "execmap preread:freemem %d size %lu",
1215                                     freemem, len);
1216                                 (void) as_fault(p->p_as->a_hat, p->p_as,
1217                                     (caddr_t)addr, len, F_INVAL, S_READ);
1218                         }
1219                 } else {
1220                         if (valid_usr_range(addr, len, prot, p->p_as,
1221                             p->p_as->a_userlimit) != RANGE_OKAY) {
1222                                 error = ENOMEM;
1223                                 goto bad;
1224                         }

1226                         if (error = as_map(p->p_as, addr, len,
1227                             segvn_create, zfod_argsp))
1228                                 goto bad;
1229                         /*
1230                          * Read in the segment in one big chunk.
1231                          */
1232                         if (error = vn_rdwr(UIO_READ, vp, (caddr_t)oldaddr,
1233                             oldlen, (offset_t)oldoffset, UIO_USERSPACE, 0,
1234                             (rlim64_t)0, CRED(), (ssize_t *)0))
1235                                 goto bad;
1236                         /*
1237                          * Now set protections.
1238                          */
1239                         if (prot != PROT_ZFOD) {
1240                                 (void) as_setprot(p->p_as, (caddr_t)addr,
1241                                     len, prot);
1242                         }
1243                 }
1244         }

1246         if (zfodlen) {
1247                 struct as *as = curproc->p_as;
1248                 struct seg *seg;
1249                 uint_t zprot = 0;

1251                 end = (size_t)addr + len;
1252                 zfodbase = (caddr_t)roundup(end, PAGESIZE);
1253                 zfoddiff = (uintptr_t)zfodbase - end;
1254                 if (zfoddiff) {
1255                         /*
1256                          * Before we go to zero the remaining space on the last
1257                          * page, make sure we have write permission.
1258                          *
1259                          * Normal illumos binaries don't even hit the case
1260                          * where we have to change permission on the last page
1261                          * since their protection is typically either
1262                          *    PROT_USER | PROT_WRITE | PROT_READ
1263                          * or
1264                          *    PROT_ZFOD (same as PROT_ALL).
1265                          *
1266                          * We need to be careful how we zero-fill the last page
1267                          * if the segment protection does not include
1268                          * PROT_WRITE. Using as_setprot() can cause the VM
```

```
1269                         * segment code to call segvn_vpage(), which must
1270                         * allocate a page struct for each page in the segment.
1271                         * If we have a very large segment, this may fail, so
1272                         * we have to check for that, even though we ignore
1273                         * other return values from as_setprot.
1274                         */
1276                        AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1277                        seg = as_segat(curproc->p_as, (caddr_t)end);
1278                        if (seg != NULL)
1279                                segop_getprot(seg, (caddr_t)end, zfoddiff - 1,
1279                                SEGOP_GETPROT(seg, (caddr_t)end, zfoddiff - 1,
1280                                    &zprot);
1281                        AS_LOCK_EXIT(as, &as->a_lock);
1283                        if (seg != NULL && (zprot & PROT_WRITE) == 0) {
1284                                if (as_setprot(as, (caddr_t)end, zfoddiff - 1,
1285                                    zprot | PROT_WRITE) == ENOMEM) {
1286                                        error = ENOMEM;
1287                                        goto bad;
1288                                }
1289                        }
1291                        if (on_fault(&ljb)) {
1292                                no_fault();
1293                                if (seg != NULL && (zprot & PROT_WRITE) == 0)
1294                                        (void) as_setprot(as, (caddr_t)end,
1295                                            zfoddiff - 1, zprot);
1296                                error = EFAULT;
1297                                goto bad;
1298                        }
1299                        uzero((void *)end, zfoddiff);
1300                        no_fault();
1301                        if (seg != NULL && (zprot & PROT_WRITE) == 0)
1302                                (void) as_setprot(as, (caddr_t)end,
1303                                    zfoddiff - 1, zprot);
1304                }
1305                if (zfodlen > zfoddiff) {
1306                        struct segvn_crargs crargs =
1307                            SEGVN_ZFOD_ARGS(PROT_ZFOD, PROT_ALL);
1309                        zfodlen -= zfoddiff;
1310                        if (valid_usr_range(zfodbase, zfodlen, prot, p->p_as,
1311                            p->p_as->a_userlimit) != RANGE_OKAY) {
1312                                error = ENOMEM;
1313                                goto bad;
1314                        }
1315                        if (szc > 0) {
1316                                /*
1317                                 * ASSERT alignment because the mapelfexec()
1318                                 * caller for the szc > 0 case extended zfod
1319                                 * so it's end is pgsz aligned.
1320                                 */
1321                                size_t pgsz = page_get_pagesize(szc);
1322                                ASSERT(IS_P2ALIGNED(zfodbase + zfodlen, pgsz));
1324                                if (IS_P2ALIGNED(zfodbase, pgsz)) {
1325                                        crargs.szc = szc;
1326                                } else {
1327                                        crargs.szc = AS_MAP_HEAP;
1328                                }
1329                        } else {
1330                                crargs.szc = AS_MAP_NO_LPOOB;
1331                        }
1332                        if (error = as_map(p->p_as, (caddr_t)zfodbase,
1333                            zfodlen, segvn_create, &crargs))
```



```
1334                                        goto bad;
1335                                if (prot != PROT_ZFOD) {
1336                                        (void) as_setprot(p->p_as, (caddr_t)zfodbase,
1337                                            zfodlen, prot);
1338                                }
1339                        }
1340                }
1341                return (0);
1342 bad:
1343                return (error);
1344 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  118977 Fri May  8 18:10:29 2015
new/usr/src/uts/common/os/lgrp.c
lgrp: getpolicy seg op has been around long enough
This special casing has been around for more than 10 years.  It's time for
it to go.  (There are no third party segment drivers anyway.)
segop_getpolicy already checks for a NULL op
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

3498 /*
3499  * Get memory allocation policy for this segment
3500  */
3501 lgrp_mem_policy_info_t *
3502 lgrp_mem_policy_get(struct seg *seg, caddr_t vaddr)
3503 {
3504         return (segop_getpolicy(seg, vaddr));
3504         lgrp_mem_policy_info_t  *policy_info;
3505         extern struct seg_ops   segspt_ops;
3506         extern struct seg_ops   segspt_shmops;

3508         /*
3509          * This is for binary compatibility to protect against third party
3510          * segment drivers which haven't recompiled to allow for
3511          * SEGOP_GETPOLICY()
3512          */
3513         if (seg->s_ops != &segvn_ops && seg->s_ops != &segspt_ops &&
3514             seg->s_ops != &segspt_shmops)
3515                 return (NULL);

3517         policy_info = NULL;
3518         if (seg->s_ops->getpolicy != NULL)
3519                 policy_info = SEGOP_GETPOLICY(seg, vaddr);

3521         return (policy_info);
3505 }
______unchanged_portion_omitted_
```

```
**********************************************************
   69060 Fri May  8 18:10:29 2015
new/usr/src/uts/common/os/mmapobj.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

1434 /*
1435  * Check the address space to see if the virtual addresses to be used are
1436  * available.  If they are not, return errno for failure.  On success, 0
1437  * will be returned, and the virtual addresses for each mmapobj_result_t
1438  * will be reserved.  Note that a reservation could have earlier been made
1439  * for a given segment via a /dev/null mapping.  If that is the case, then
1440  * we can use that VA space for our mappings.
1441  * Note: this function will only be used for ET_EXEC binaries.
1442  */
1443 int
1444 check_exec_addrs(int loadable, mmapobj_result_t *mrp, caddr_t start_addr)
1445 {
1446         int i;
1447         struct as *as = curproc->p_as;
1448         struct segvn_crargs crargs = SEGVN_ZFOD_ARGS(PROT_ZFOD, PROT_ALL);
1449         int ret;
1450         caddr_t myaddr;
1451         size_t mylen;
1452         struct seg *seg;

1454         /* No need to reserve swap space now since it will be reserved later */
1455         crargs.flags |= MAP_NORESERVE;
1456         as_rangelock(as);
1457         for (i = 0; i < loadable; i++) {

1459                 myaddr = start_addr + (size_t)mrp[i].mr_addr;
1460                 mylen = mrp[i].mr_msize;

1462                 /* See if there is a hole in the as for this range */
1463                 if (as_gap(as, mylen, &myaddr, &mylen, 0, NULL) == 0) {
1464                         ASSERT(myaddr == start_addr + (size_t)mrp[i].mr_addr);
1465                         ASSERT(mylen == mrp[i].mr_msize);

1467 #ifdef DEBUG
1468                         if (MR_GET_TYPE(mrp[i].mr_flags) == MR_PADDING) {
1469                                 MOBJ_STAT_ADD(exec_padding);
1470                         }
1471 #endif
1472                         ret = as_map(as, myaddr, mylen, segvn_create, &crargs);
1473                         if (ret) {
1474                                 as_rangeunlock(as);
1475                                 mmapobj_unmap_exec(mrp, i, start_addr);
1476                                 return (ret);
1477                         }
1478                 } else {
1479                         /*
1480                          * There is a mapping that exists in the range
1481                          * so check to see if it was a "reservation"
1482                          * from /dev/null.  The mapping is from
1483                          * /dev/null if the mapping comes from
1484                          * segdev and the type is neither MAP_SHARED
1485                          * nor MAP_PRIVATE.
1486                          */
1487                         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1488                         seg = as_findseg(as, myaddr, 0);
1489                         MOBJ_STAT_ADD(exec_addr_mapped);
1490                         if (seg && seg->s_ops == &segdev_ops &&
1491                             ((segop_gettype(seg, myaddr) &
1491                             ((SEGOP_GETTYPE(seg, myaddr) &
```



```
1492                             (MAP_SHARED | MAP_PRIVATE)) == 0) &&
1493                             myaddr >= seg->s_base &&
1494                             myaddr + mylen <=
1495                             seg->s_base + seg->s_size) {
1496                                 MOBJ_STAT_ADD(exec_addr_devnull);
1497                                 AS_LOCK_EXIT(as, &as->a_lock);
1498                                 (void) as_unmap(as, myaddr, mylen);
1499                                 ret = as_map(as, myaddr, mylen, segvn_create,
1500                                     &crargs);
1501                                 mrp[i].mr_flags |= MR_RESV;
1502                                 if (ret) {
1503                                         as_rangeunlock(as);
1504                                         /* Need to remap what we unmapped */
1505                                         mmapobj_unmap_exec(mrp, i + 1,
1506                                             start_addr);
1507                                         return (ret);
1508                                 }
1509                         } else {
1510                                 AS_LOCK_EXIT(as, &as->a_lock);
1511                                 as_rangeunlock(as);
1512                                 mmapobj_unmap_exec(mrp, i, start_addr);
1513                                 MOBJ_STAT_ADD(exec_addr_in_use);
1514                                 return (EADDRINUSE);
1515                         }
1516                 }
1517         }
1518         as_rangeunlock(as);
1519         return (0);
1520 }
______unchanged_portion_omitted_
```

```
**********************************************************
   15482 Fri May  8 18:10:29 2015
new/usr/src/uts/common/os/panic.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
```

```
/*
 * 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) 1999, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright (c) 2011, Joyent, Inc. All rights reserved.
 */

/*
 * When the operating system detects that it is in an invalid state, a panic
 * is initiated in order to minimize potential damage to user data and to
 * facilitate debugging.  There are three major tasks to be performed in
 * a system panic: recording information about the panic in memory (and thus
 * making it part of the crash dump), synchronizing the file systems to
 * preserve user file data, and generating the crash dump.  We define the
 * system to be in one of four states with respect to the panic code:
 *
 * CALM    - the state of the system prior to any thread initiating a panic
 *
 * QUIESCE - the state of the system when the first thread to initiate
 *           a system panic records information about the cause of the panic
 *           and renders the system quiescent by stopping other processors
 *
 * SYNC    - the state of the system when we synchronize the file systems
 * DUMP    - the state when we generate the crash dump.
 *
 * The transitions between these states are irreversible: once we begin
 * panicking, we only make one attempt to perform the actions associated with
 * each state.
 *
 * The panic code itself must be re-entrant because actions taken during any
 * state may lead to another system panic.  Additionally, any Solaris
 * thread may initiate a panic at any time, and so we must have synchronization
 * between threads which attempt to initiate a state transition simultaneously.
 * The panic code makes use of a special locking primitive, a trigger, to
```



```
 * perform this synchronization.  A trigger is simply a word which is set
 * atomically and can only be set once.  We declare three triggers, one for
 * each transition between the four states.  When a thread enters the panic
 * code it attempts to set each trigger; if it fails it moves on to the
 * next trigger.  A special case is the first trigger: if two threads race
 * to perform the transition to QUIESCE, the losing thread may execute before
 * the winner has a chance to stop its CPU.  To solve this problem, we have
 * the loser look ahead to see if any other triggers are set; if not, it
 * presumes a panic is underway and simply spins.  Unfortunately, since we
 * are panicking, it is not possible to know this with absolute certainty.
 *
 * There are two common reasons for re-entering the panic code once a panic
 * has been initiated: (1) after we debug_enter() at the end of QUIESCE,
 * the operator may type "sync" instead of "go", and the PROM's sync callback
 * routine will invoke panic(); (2) if the clock routine decides that sync
 * or dump is not making progress, it will invoke panic() to force a timeout.
 * The design assumes that a third possibility, another thread causing an
 * unrelated panic while sync or dump is still underway, is extremely unlikely.
 * If this situation occurs, we may end up triggering dump while sync is
 * still in progress.  This third case is considered extremely unlikely because
 * all other CPUs are stopped and low-level interrupts have been blocked.
 *
 * The panic code is entered via a call directly to the vpanic() function,
 * or its varargs wrappers panic() and cmn_err(9F).  The vpanic routine
 * is implemented in assembly language to record the current machine
 * registers, attempt to set the trigger for the QUIESCE state, and
 * if successful, switch stacks on to the panic_stack before calling into
 * the common panicsys() routine.  The first thread to initiate a panic
 * is allowed to make use of the reserved panic_stack so that executing
 * the panic code itself does not overwrite valuable data on that thread's
 * stack *ahead* of the current stack pointer.  This data will be preserved
 * in the crash dump and may prove invaluable in determining what this
 * thread has previously been doing.  The first thread, saved in panic_thread,
 * is also responsible for stopping the other CPUs as quickly as possible,
 * and then setting the various panic_* variables.  Most important among
 * these is panicstr, which allows threads to subsequently bypass held
 * locks so that we can proceed without ever blocking.  We must stop the
 * other CPUs *prior* to setting panicstr in case threads running there are
 * currently spinning to acquire a lock; we want that state to be preserved.
 * Every thread which initiates a panic has its T_PANIC flag set so we can
 * identify all such threads in the crash dump.
 *
 * The panic_thread is also allowed to make use of the special memory buffer
 * panicbuf, which on machines with appropriate hardware is preserved across
 * reboots.  We allow the panic_thread to store its register set and panic
 * message in this buffer, so even if we fail to obtain a crash dump we will
 * be able to examine the machine after reboot and determine some of the
 * state at the time of the panic.  If we do get a dump, the panic buffer
 * data is structured so that a debugger can easily consume the information
 * therein (see <sys/panic.h>).
 *
 * Each platform or architecture is required to implement the functions
 * panic_savetrap() to record trap-specific information to panicbuf,
 * panic_saveregs() to record a register set to panicbuf, panic_stopcpus()
 * to halt all CPUs but the panicking CPU, panic_quiesce_hw() to perform
 * miscellaneous platform-specific tasks *after* panicstr is set,
 * panic_showtrap() to print trap-specific information to the console,
 * and panic_dump_hw() to perform platform tasks prior to calling dumpsys().
 *
 * A Note on Word Formation, courtesy of the Oxford Guide to English Usage:
 *
 * Words ending in -c interpose k before suffixes which otherwise would
 * indicate a soft c, and thus the verb and adjective forms of 'panic' are
 * spelled "panicked", "panicking", and "panicky" respectively.  Use of
 * the ill-conceived "panicing" and "panic'd" is discouraged.
 */
```

```
123 #include <sys/types.h>
124 #include <sys/varargs.h>
125 #include <sys/sysmacros.h>
126 #include <sys/cmn_err.h>
127 #include <sys/cpuvar.h>
128 #include <sys/thread.h>
129 #include <sys/t_lock.h>
130 #include <sys/cred.h>
131 #include <sys/systm.h>
132 #include <sys/archsystm.h>
133 #include <sys/uadmin.h>
134 #include <sys/callb.h>
135 #include <sys/vfs.h>
136 #include <sys/log.h>
137 #include <sys/disp.h>
138 #include <sys/param.h>
139 #include <sys/dumphdr.h>
140 #include <sys/ftrace.h>
141 #include <sys/reboot.h>
142 #include <sys/debug.h>
143 #include <sys/stack.h>
144 #include <sys/spl.h>
145 #include <sys/errorq.h>
146 #include <sys/panic.h>
147 #include <sys/fm/util.h>
148 #include <sys/clock_impl.h>

150 /*
151  * Panic variables which are set once during the QUIESCE state by the
152  * first thread to initiate a panic.  These are examined by post-mortem
153  * debugging tools; the inconsistent use of 'panic' versus 'panic_' in
154  * the variable naming is historical and allows legacy tools to work.
155  */
156 #pragma align STACK_ALIGN(panic_stack)
157 char panic_stack[PANICSTKSIZE];         /* reserved stack for panic_thread */
158 kthread_t *panic_thread;                /* first thread to call panicsys() */
159 cpu_t panic_cpu;                        /* cpu from first call to panicsys() */
160 label_t panic_regs;                     /* setjmp label from panic_thread */
161 label_t panic_pcb;                      /* t_pcb at time of panic */
162 struct regs *panic_reg;                 /* regs struct from first panicsys() */
163 char *volatile panicstr;                /* format string to first panicsys() */
164 va_list panicargs;                      /* arguments to first panicsys() */
165 clock_t panic_lbolt;                    /* lbolt at time of panic */
166 int64_t panic_lbolt64;                  /* lbolt64 at time of panic */
167 hrtime_t panic_hrtime;                  /* hrtime at time of panic */
168 timespec_t panic_hrestime;              /* hrestime at time of panic */
169 int panic_ipl;                          /* ipl on panic_cpu at time of panic */
170 ushort_t panic_schedflag;               /* t_schedflag for panic_thread */
171 cpu_t *panic_bound_cpu;                 /* t_bound_cpu for panic_thread */
172 char panic_preempt;                     /* t_preempt for panic_thread */

174 /*
175  * Panic variables which can be set via /etc/system or patched while
176  * the system is in operation.  Again, the stupid names are historic.
177  */
178 char *panic_bootstr = NULL;             /* mdboot string to use after panic */
179 int panic_bootfcn = AD_BOOT;            /* mdboot function to use after panic */
180 int halt_on_panic = 0;                  /* halt after dump instead of reboot? */
181 int nopanicdebug = 0;                   /* reboot instead of call debugger? */
182 int in_sync = 0;                        /* skip vfs_syncall() and just dump? */

184 /*
185  * The do_polled_io flag is set by the panic code to inform the SCSI subsystem
186  * to use polled mode instead of interrupt-driven i/o.
187  */
```



```
188 int do_polled_io = 0;

190 /*
191  * The panic_forced flag is set by the uadmin A_DUMP code to inform the
192  * panic subsystem that it should not attempt an initial debug_enter.
193  */
194 int panic_forced = 0;

196 /*
197  * Triggers for panic state transitions:
198  */
199 int panic_quiesce;                      /* trigger for CALM    -> QUIESCE */
200 int panic_sync;                         /* trigger for QUIESCE -> SYNC */
201 int panic_dump;                         /* trigger for SYNC    -> DUMP */

203 /*
204  * Variable signifying quiesce(9E) is in progress.
205  */
206 volatile int quiesce_active = 0;

208 void
209 panicsys(const char *format, va_list alist, struct regs *rp, int on_panic_stack)
210 {
211         int s = spl8();
212         kthread_t *t = curthread;
213         cpu_t *cp = CPU;

215         caddr_t intr_stack = NULL;
216         uint_t intr_actv;

218         ushort_t schedflag = t->t_schedflag;
219         cpu_t *bound_cpu = t->t_bound_cpu;
220         char preempt = t->t_preempt;
221         label_t pcb = t->t_pcb;

223         (void) setjmp(&t->t_pcb);
224         t->t_flag |= T_PANIC;

226         t->t_schedflag |= TS_DONT_SWAP;
226         t->t_bound_cpu = cp;
227         t->t_preempt++;

229         panic_enter_hw(s);

231         /*
232          * If we're on the interrupt stack and an interrupt thread is available
233          * in this CPU's pool, preserve the interrupt stack by detaching an
234          * interrupt thread and making its stack the intr_stack.
235          */
236         if (CPU_ON_INTR(cp) && cp->cpu_intr_thread != NULL) {
237                 kthread_t *it = cp->cpu_intr_thread;

239                 intr_stack = cp->cpu_intr_stack;
240                 intr_actv = cp->cpu_intr_actv;

242                 cp->cpu_intr_stack = thread_stk_init(it->t_stk);
243                 cp->cpu_intr_thread = it->t_link;

245                 /*
246                  * Clear only the high level bits of cpu_intr_actv.
247                  * We want to indicate that high-level interrupts are
248                  * not active without destroying the low-level interrupt
249                  * information stored there.
250                  */
251                 cp->cpu_intr_actv &= ((1 << (LOCK_LEVEL + 1)) - 1);
252         }
```

```
254         /*
255          * Record one-time panic information and quiesce the other CPUs.
256          * Then print out the panic message and stack trace.
257          */
258         if (on_panic_stack) {
259                 panic_data_t *pdp = (panic_data_t *)panicbuf;

261                 pdp->pd_version = PANICBUFVERS;
262                 pdp->pd_msgoff = sizeof (panic_data_t) - sizeof (panic_nv_t);

264                 (void) strncpy(pdp->pd_uuid, dump_get_uuid(),
265                     sizeof (pdp->pd_uuid));

267                 if (t->t_panic_trap != NULL)
268                         panic_savetrap(pdp, t->t_panic_trap);
269                 else
270                         panic_saveregs(pdp, rp);

272                 (void) vsnprintf(&panicbuf[pdp->pd_msgoff],
273                     PANICBUFSIZE - pdp->pd_msgoff, format, alist);

275                 /*
276                  * Call into the platform code to stop the other CPUs.
277                  * We currently have all interrupts blocked, and expect that
278                  * the platform code will lower ipl only as far as needed to
279                  * perform cross-calls, and will acquire as *few* locks as is
280                  * possible -- panicstr is not set so we can still deadlock.
281                  */
282                 panic_stopcpus(cp, t, s);

284                 panicstr = (char *)format;
285                 va_copy(panicargs, alist);
286                 panic_lbolt = LBOLT_NO_ACCOUNT;
287                 panic_lbolt64 = LBOLT_NO_ACCOUNT64;
288                 panic_hrestime = hrestime;
289                 panic_hrtime = gethrtime_waitfree();
290                 panic_thread = t;
291                 panic_regs = t->t_pcb;
292                 panic_reg = rp;
293                 panic_cpu = *cp;
294                 panic_ipl = spltoipl(s);
295                 panic_schedflag = schedflag;
296                 panic_bound_cpu = bound_cpu;
297                 panic_preempt = preempt;
298                 panic_pcb = pcb;

300                 if (intr_stack != NULL) {
301                         panic_cpu.cpu_intr_stack = intr_stack;
302                         panic_cpu.cpu_intr_actv = intr_actv;
303                 }

305                 /*
306                  * Lower ipl to 10 to keep clock() from running, but allow
307                  * keyboard interrupts to enter the debugger.  These callbacks
308                  * are executed with panicstr set so they can bypass locks.
309                  */
310                 splx(ipltospl(CLOCK_LEVEL));
311                 panic_quiesce_hw(pdp);
312                 (void) FTRACE_STOP();
313                 (void) callb_execute_class(CB_CL_PANIC, NULL);

315                 if (log_intrq != NULL)
316                         log_flushq(log_intrq);

318                 /*
```



```
319                  * If log_consq has been initialized and syslogd has started,
320                  * print any messages in log_consq that haven't been consumed.
321                  */
322                 if (log_consq != NULL && log_consq != log_backlogq)
323                         log_printq(log_consq);

325                 fm_banner();

327 #if defined(__x86)
328                 /*
329                  * A hypervisor panic originates outside of Solaris, so we
330                  * don't want to prepend the panic message with misleading
331                  * pointers from within Solaris.
332                  */
333                 if (!IN_XPV_PANIC())
334 #endif
335                         printf("\n\rpanic[cpu%d]/thread=%p: ", cp->cpu_id,
336                             (void *)t);
337                 vprintf(format, alist);
338                 printf("\n\n");

340                 if (t->t_panic_trap != NULL) {
341                         panic_showtrap(t->t_panic_trap);
342                         printf("\n");
343                 }

345                 traceregs(rp);
346                 printf("\n");

348                 if (((boothowto & RB_DEBUG) || obpdebug) &&
349                     !nopanicdebug && !panic_forced) {
350                         if (dumpvp != NULL) {
351                                 debug_enter("panic: entering debugger "
352                                     "(continue to save dump)");
353                         } else {
354                                 debug_enter("panic: entering debugger "
355                                     "(no dump device, continue to reboot)");
356                         }
357                 }

359         } else if (panic_dump != 0 || panic_sync != 0 || panicstr != NULL) {
360                 printf("\n\rpanic[cpu%d]/thread=%p: ", cp->cpu_id, (void *)t);
361                 vprintf(format, alist);
362                 printf("\n");
363         } else
364                 goto spin;

366         /*
367          * Prior to performing sync or dump, we make sure that do_polled_io is
368          * set, but we'll leave ipl at 10; deadman(), a CY_HIGH_LEVEL cyclic,
369          * will re-enter panic if we are not making progress with sync or dump.
370          */

372         /*
373          * Sync the filesystems.  Reset t_cred if not set because much of
374          * the filesystem code depends on CRED() being valid.
375          */
376         if (!in_sync && panic_trigger(&panic_sync)) {
377                 if (t->t_cred == NULL)
378                         t->t_cred = kcred;
379                 splx(ipltospl(CLOCK_LEVEL));
380                 do_polled_io = 1;
381                 vfs_syncall();
382         }

384         /*
```

```
385          * Take the crash dump.  If the dump trigger is already set, try to
386          * enter the debugger again before rebooting the system.
387          */
388         if (panic_trigger(&panic_dump)) {
389                 panic_dump_hw(s);
390                 splx(ipltospl(CLOCK_LEVEL));
391                 errorq_panic();
392                 do_polled_io = 1;
393                 dumpsys();
394         } else if (((boothowto & RB_DEBUG) || obpdebug) && !nopanicdebug) {
395                 debug_enter("panic: entering debugger (continue to reboot)");
396         } else
397                 printf("dump aborted: please record the above information!\n");

399         if (halt_on_panic)
400                 mdboot(A_REBOOT, AD_HALT, NULL, B_FALSE);
401         else
402                 mdboot(A_REBOOT, panic_bootfcn, panic_bootstr, B_FALSE);
403 spin:
404         /*
405          * Restore ipl to at most CLOCK_LEVEL so we don't end up spinning
406          * and unable to jump into the debugger.
407          */
408         splx(MIN(s, ipltospl(CLOCK_LEVEL)));
409         for (;;)
410                 ;
411 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    2754 Fri May  8 18:10:30 2015
new/usr/src/uts/common/os/sched.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
```

```
/*
 * 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.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

#include <sys/param.h>
#include <sys/types.h>
#include <sys/sysmacros.h>
#include <sys/systm.h>
#include <sys/proc.h>
#include <sys/cpuvar.h>
#include <sys/var.h>
#include <sys/tuneable.h>
#include <sys/cmn_err.h>
#include <sys/buf.h>
#include <sys/disp.h>
#include <sys/vmsystm.h>
#include <sys/vmparam.h>
#include <sys/class.h>
#include <sys/vtrace.h>
#include <sys/modctl.h>
#include <sys/debug.h>
#include <sys/tnf_probe.h>
#include <sys/procfs.h>

#include <vm/seg.h>
#include <vm/seg_kp.h>
#include <vm/as.h>
#include <vm/rm.h>
#include <vm/seg_kmem.h>
#include <sys/callb.h>
```



```
/*
 * The swapper sleeps on runout when there is no one to swap in.
 * It sleeps on runin when it could not find space to swap someone
 * in or after swapping someone in.
 */
char    runout;
char    runin;
char    wake_sched;     /* flag tells clock to wake swapper on next tick */
char    wake_sched_sec; /* flag tells clock to wake swapper after a second */

/*
 * The swapper swaps processes to reduce memory demand and runs
 * when avefree < desfree.  The swapper resorts to SOFTSWAP when
 * avefree < desfree which results in swapping out all processes
 * sleeping for more than maxslp seconds.  HARDSWAP occurs when the
 * system is on the verge of thrashing and this results in swapping
 * out runnable threads or threads sleeping for less than maxslp secs.
 *
 * The swapper runs through all the active processes in the system
 * and invokes the scheduling class specific swapin/swapout routine
 * for every thread in the process to obtain an effective priority
 * for the process.  A priority of -1 implies that the thread isn't
 * swappable.  This effective priority is used to find the most
 * eligible process to swapout or swapin.
 *
 * NOTE:  Threads which have been swapped are not linked on any
 *        queue and their dispatcher lock points at the "swapped_lock".
 *
 * Processes containing threads with the TS_DONT_SWAP flag set cannot be
 * swapped out immediately by the swapper.  This is due to the fact that
 * such threads may be holding locks which may be needed by the swapper
 * to push its pages out.  The TS_SWAPENQ flag is set on such threads
 * to prevent them running in user mode.  When such threads reach a
 * safe point (i.e., are not holding any locks - CL_TRAPRET), they
 * queue themseleves onto the swap queue which is processed by the
 * swapper.  This results in reducing memory demand when the system
 * is desparate for memory as the thread can't run in user mode.
 *
 * The swap queue consists of threads, linked via t_link, which are
 * haven't been swapped, are runnable but not on the run queue.  The
 * swap queue is protected by the "swapped_lock".  The dispatcher
 * lock (t_lockp) of all threads on the swap queue points at the
 * "swapped_lock".  Thus, the entire queue and/or threads on the
 * queue can be locked by acquiring "swapped_lock".
 */
static kthread_t *tswap_queue;
extern disp_lock_t swapped_lock; /* protects swap queue and threads on it */

int     maxslp = 0;
pgcnt_t avefree;        /* 5 sec moving average of free memory */
pgcnt_t avefree30;      /* 30 sec moving average of free memory */

/*
 * Minimum size used to decide if sufficient memory is available
 * before a process is swapped in.  This is necessary since in most
 * cases the actual size of a process (p_swrss) being swapped in
 * is usually 2 pages (kernel stack pages).  This is due to the fact
 * almost all user pages of a process are stolen by pageout before
 * the swapper decides to swapout it out.
 */
int     min_procsize = 12;

static int      swapin(proc_t *);
static int      swapout(proc_t *, uint_t *, int);
static void     process_swap_queue();
```

```
123 #ifdef __sparc
124 extern void lwp_swapin(kthread_t *);
125 #endif /* __sparc */

127 /*
128  * Counters to keep track of the number of swapins or swapouts.
129  */
130 uint_t tot_swapped_in, tot_swapped_out;
131 uint_t softswap, hardswap, swapqswap;

133 /*
134  * Macro to determine if a process is eligble to be swapped.
135  */
136 #define not_swappable(p)                                        \
137         (((p)->p_flag & SSYS) || (p)->p_stat == SIDL ||         \
138             (p)->p_stat == SZOMB || (p)->p_as == NULL ||        \
139             (p)->p_as == &kas)

141 /*
 61  * Memory scheduler.
 62  */
 63 void
 64 sched()
 65 {
 66         kthread_id_t    t;
148         pri_t           proc_pri;
149         pri_t           thread_pri;
150         pri_t           swapin_pri;
151         int             desperate;
152         pgcnt_t         needs;
153         int             divisor;
154         proc_t          *prp;
155         proc_t          *swapout_prp;
156         proc_t          *swapin_prp;
157         spgcnt_t        avail;
158         int             chosen_pri;
159         time_t          swapout_time;
160         time_t          swapin_proc_time;
 67         callb_cpr_t     cprinfo;
 68         kmutex_t        swap_cpr_lock;

 70         mutex_init(&swap_cpr_lock, NULL, MUTEX_DEFAULT, NULL);
 71         CALLB_CPR_INIT(&cprinfo, &swap_cpr_lock, callb_generic_cpr, "sched");
166         if (maxslp == 0)
167                 maxslp = MAXSLP;
168 loop:
169         needs = 0;
170         desperate = 0;

 73         for (;;) {
172         swapin_pri = v.v_nglobpris;
173         swapin_prp = NULL;
174         chosen_pri = -1;

176         process_swap_queue();

178         /*
179          * Set desperate if
180          *      1.  At least 2 runnable processes (on average).
181          *      2.  Short (5 sec) and longer (30 sec) average is less
182          *          than minfree and desfree respectively.
183          *      3.  Pagein + pageout rate is excessive.
184          */
 74                 if (avenrun[0] >= 2 * FSCALE &&
 75                     (MAX(avefree, avefree30) < desfree) &&
```



```
 76                     (pginrate + pgoutrate > maxpgio || avefree < minfree)) {
188                 TRACE_4(TR_FAC_SCHED, TR_DESPERATE,
189                     "desp:avefree: %d, avefree30: %d, freemem: %d"
190                     " pginrate: %d\n", avefree, avefree30, freemem, pginrate);
191                 desperate = 1;
192                 goto unload;
193         }

195         /*
196          * Search list of processes to swapin and swapout deadwood.
197          */
198         swapin_proc_time = 0;
199 top:
200         mutex_enter(&pidlock);
201         for (prp = practive; prp != NULL; prp = prp->p_next) {
202                 if (not_swappable(prp))
203                         continue;

205                 /*
206                  * Look at processes with at least one swapped lwp.
207                  */
208                 if (prp->p_swapcnt) {
209                         time_t proc_time;

211                         /*
212                          * Higher priority processes are good candidates
213                          * to swapin.
214                          */
215                         mutex_enter(&prp->p_lock);
216                         proc_pri = -1;
217                         t = prp->p_tlist;
218                         proc_time = 0;
219                         do {
220                                 if (t->t_schedflag & TS_LOAD)
221                                         continue;

223                                 thread_lock(t);
224                                 thread_pri = CL_SWAPIN(t, 0);
225                                 thread_unlock(t);

227                                 if (t->t_stime - proc_time > 0)
228                                         proc_time = t->t_stime;
229                                 if (thread_pri > proc_pri)
230                                         proc_pri = thread_pri;
231                         } while ((t = t->t_forw) != prp->p_tlist);
232                         mutex_exit(&prp->p_lock);

234                         if (proc_pri == -1)
235                                 continue;

237                         TRACE_3(TR_FAC_SCHED, TR_CHOOSE_SWAPIN,
238                             "prp %p epri %d proc_time %d",
239                             prp, proc_pri, proc_time);

241                         /*
242                          * Swapin processes with a high effective priority.
243                          */
244                         if (swapin_prp == NULL || proc_pri > chosen_pri) {
245                                 swapin_prp = prp;
246                                 chosen_pri = proc_pri;
247                                 swapin_pri = proc_pri;
248                                 swapin_proc_time = proc_time;
249                         }
250                 } else {
251                         /*
252                          * No need to soft swap if we have sufficient
```

```
253                          * memory.
254                          */
255                         if (avefree > desfree ||
256                             avefree < desfree && freemem > desfree)
257                                 continue;
258
259                         /*
260                          * Skip processes that are exiting
261                          * or whose address spaces are locked.
262                          */
263                         mutex_enter(&prp->p_lock);
264                         if ((prp->p_flag & SEXITING) ||
265                             (prp->p_as != NULL && AS_ISPGLCK(prp->p_as))) {
266                                 mutex_exit(&prp->p_lock);
267                                 continue;
268                         }
269
270                         /*
271                          * Softswapping to kick out deadwood.
272                          */
273                         proc_pri = -1;
274                         t = prp->p_tlist;
275                         do {
276                                 if ((t->t_schedflag & (TS_SWAPENQ |
277                                     TS_ON_SWAPQ | TS_LOAD)) != TS_LOAD)
278                                         continue;
279
280                                 thread_lock(t);
281                                 thread_pri = CL_SWAPOUT(t, SOFTSWAP);
282                                 thread_unlock(t);
283                                 if (thread_pri > proc_pri)
284                                         proc_pri = thread_pri;
285                         } while ((t = t->t_forw) != prp->p_tlist);
286
287                         if (proc_pri != -1) {
288                                 uint_t swrss;
289
290                                 mutex_exit(&pidlock);
291
292                                 TRACE_1(TR_FAC_SCHED, TR_SOFTSWAP,
293                                     "softswap:prp %p", prp);
294
295                                 (void) swapout(prp, &swrss, SOFTSWAP);
296                                 softswap++;
297                                 prp->p_swrss += swrss;
298                                 mutex_exit(&prp->p_lock);
299                                 goto top;
300                         }
301                         mutex_exit(&prp->p_lock);
302                 }
303         }
304         if (swapin_prp != NULL)
305                 mutex_enter(&swapin_prp->p_lock);
306         mutex_exit(&pidlock);
307
308         if (swapin_prp == NULL) {
309                 TRACE_3(TR_FAC_SCHED, TR_RUNOUT,
310                 "schedrunout:runout nswapped: %d, avefree: %ld freemem: %ld",
311                     nswapped, avefree, freemem);
312
313                 t = curthread;
314                 thread_lock(t);
315                 runout++;
316                 t->t_schedflag |= (TS_ALLSTART & ~TS_CSTART);
317                 t->t_whystop = PR_SUSPENDED;
318                 t->t_whatstop = SUSPEND_NORMAL;
```



```
319                 (void) new_mstate(t, LMS_SLEEP);
320                 mutex_enter(&swap_cpr_lock);
321                 CALLB_CPR_SAFE_BEGIN(&cprinfo);
322                 mutex_exit(&swap_cpr_lock);
323                 thread_stop(t);         /* change state and drop lock */
324                 swtch();
325                 mutex_enter(&swap_cpr_lock);
326                 CALLB_CPR_SAFE_END(&cprinfo, &swap_cpr_lock);
327                 mutex_exit(&swap_cpr_lock);
328                 goto loop;
329         }
330
331         /*
332          * Decide how deserving this process is to be brought in.
333          * Needs is an estimate of how much core the process will
334          * need.  If the process has been out for a while, then we
335          * will bring it in with 1/2 the core needed, otherwise
336          * we are conservative.
337          */
338         divisor = 1;
339         swapout_time = (ddi_get_lbolt() - swapin_proc_time) / hz;
340         if (swapout_time > maxslp / 2)
341                 divisor = 2;
342
343         needs = MIN(swapin_prp->p_swrss, lotsfree);
344         needs = MAX(needs, min_procsize);
345         needs = needs / divisor;
346
347         /*
348          * Use freemem, since we want processes to be swapped
349          * in quickly.
350          */
351         avail = freemem - deficit;
352         if (avail > (spgcnt_t)needs) {
353                 deficit += needs;
354
355                 TRACE_2(TR_FAC_SCHED, TR_SWAPIN_VALUES,
356                     "swapin_values: prp %p needs %lu", swapin_prp, needs);
357
358                 if (swapin(swapin_prp)) {
359                         mutex_exit(&swapin_prp->p_lock);
360                         goto loop;
361                 }
362                 deficit -= MIN(needs, deficit);
363                 mutex_exit(&swapin_prp->p_lock);
364         } else {
365                 mutex_exit(&swapin_prp->p_lock);
366                 /*
367                  * If deficit is high, too many processes have been
368                  * swapped in so wait a sec before attempting to
369                  * swapin more.
370                  */
371                 if (freemem > needs) {
372                         TRACE_2(TR_FAC_SCHED, TR_HIGH_DEFICIT,
373                             "deficit: prp %p needs %lu", swapin_prp, needs);
374                         goto block;
375                 }
376         }
377
378         TRACE_2(TR_FAC_SCHED, TR_UNLOAD,
379             "unload: prp %p needs %lu", swapin_prp, needs);
380
381 unload:
 77                         /*
 78                          * Unload all unloadable modules, free all other memory
 79                          * resources we can find, then look for a thread to
```

```
 80                          * hardswap.
384          * resources we can find, then look for a thread to hardswap.
 81                          */
 82                         modreap();
 83                         segkp_cache_free();

389         swapout_prp = NULL;
390         mutex_enter(&pidlock);
391         for (prp = practive; prp != NULL; prp = prp->p_next) {

393                 /*
394                  * No need to soft swap if we have sufficient
395                  * memory.
396                  */
397                 if (not_swappable(prp))
398                         continue;

400                 if (avefree > minfree ||
401                     avefree < minfree && freemem > desfree) {
402                         swapout_prp = NULL;
403                         break;
 84                 }

406                 /*
407                  * Skip processes that are exiting
408                  * or whose address spaces are locked.
409                  */
410                 mutex_enter(&prp->p_lock);
411                 if ((prp->p_flag & SEXITING) ||
412                     (prp->p_as != NULL && AS_ISPGLCK(prp->p_as))) {
413                         mutex_exit(&prp->p_lock);
414                         continue;
415                 }

417                 proc_pri = -1;
418                 t = prp->p_tlist;
419                 do {
420                         if ((t->t_schedflag & (TS_SWAPENQ |
421                             TS_ON_SWAPQ | TS_LOAD)) != TS_LOAD)
422                                 continue;

424                         thread_lock(t);
425                         thread_pri = CL_SWAPOUT(t, HARDSWAP);
426                         thread_unlock(t);
427                         if (thread_pri > proc_pri)
428                                 proc_pri = thread_pri;
429                 } while ((t = t->t_forw) != prp->p_tlist);

431                 mutex_exit(&prp->p_lock);
432                 if (proc_pri == -1)
433                         continue;

435                 /*
436                  * Swapout processes sleeping with a lower priority
437                  * than the one currently being swapped in, if any.
438                  */
439                 if (swapin_prp == NULL || swapin_pri > proc_pri) {
440                         TRACE_2(TR_FAC_SCHED, TR_CHOOSE_SWAPOUT,
441                             "hardswap: prp %p needs %lu", prp, needs);

443                         if (swapout_prp == NULL || proc_pri < chosen_pri) {
444                                 swapout_prp = prp;
445                                 chosen_pri = proc_pri;
446                         }
447                 }
448         }
```



```
450         /*
451          * Acquire the "p_lock" before dropping "pidlock"
452          * to prevent the proc structure from being freed
453          * if the process exits before swapout completes.
454          */
455         if (swapout_prp != NULL)
456                 mutex_enter(&swapout_prp->p_lock);
457         mutex_exit(&pidlock);

459         if ((prp = swapout_prp) != NULL) {
460                 uint_t swrss = 0;
461                 int swapped;

463                 swapped = swapout(prp, &swrss, HARDSWAP);
464                 if (swapped) {
465                         /*
466                          * If desperate, we want to give the space obtained
467                          * by swapping this process out to processes in core,
468                          * so we give them a chance by increasing deficit.
469                          */
470                         prp->p_swrss += swrss;
471                         if (desperate)
472                                 deficit += MIN(prp->p_swrss, lotsfree);
473                         hardswap++;
474                 }
475                 mutex_exit(&swapout_prp->p_lock);

477                 if (swapped)
478                         goto loop;
479         }

481         /*
482          * Delay for 1 second and look again later.
483          */
484         TRACE_3(TR_FAC_SCHED, TR_RUNIN,
485             "schedrunin:runin nswapped: %d, avefree: %ld freemem: %ld",
486             nswapped, avefree, freemem);

488 block:
 86                 t = curthread;
 87                 thread_lock(t);
491         runin++;
 88                 t->t_schedflag |= (TS_ALLSTART & ~TS_CSTART);
 89                 t->t_whystop = PR_SUSPENDED;
 90                 t->t_whatstop = SUSPEND_NORMAL;
 91                 (void) new_mstate(t, LMS_SLEEP);
 92                 mutex_enter(&swap_cpr_lock);
 93                 CALLB_CPR_SAFE_BEGIN(&cprinfo);
 94                 mutex_exit(&swap_cpr_lock);
 95                 thread_stop(t);         /* change to stop state and drop lock */
 96                 swtch();
 97                 mutex_enter(&swap_cpr_lock);
 98                 CALLB_CPR_SAFE_END(&cprinfo, &swap_cpr_lock);
 99                 mutex_exit(&swap_cpr_lock);
504         goto loop;
505 }

507 /*
508  * Remove the specified thread from the swap queue.
509  */
510 static void
511 swapdeq(kthread_id_t tp)
512 {
513         kthread_id_t *tpp;
```

```
515         ASSERT(THREAD_LOCK_HELD(tp));
516         ASSERT(tp->t_schedflag & TS_ON_SWAPQ);

518         tpp = &tswap_queue;
519         for (;;) {
520                 ASSERT(*tpp != NULL);
521                 if (*tpp == tp)
522                         break;
523                 tpp = &(*tpp)->t_link;
524         }
525         *tpp = tp->t_link;
526         tp->t_schedflag &= ~TS_ON_SWAPQ;
527 }

529 /*
530  * Swap in lwps.  Returns nonzero on success (i.e., if at least one lwp is
531  * swapped in) and 0 on failure.
532  */
533 static int
534 swapin(proc_t *pp)
535 {
536         kthread_id_t tp;
537         int err;
538         int num_swapped_in = 0;
539         struct cpu *cpup = CPU;
540         pri_t thread_pri;

542         ASSERT(MUTEX_HELD(&pp->p_lock));
543         ASSERT(pp->p_swapcnt);

545 top:
546         tp = pp->p_tlist;
547         do {
548                 /*
549                  * Only swapin eligible lwps (specified by the scheduling
550                  * class) which are unloaded and ready to run.
551                  */
552                 thread_lock(tp);
553                 thread_pri = CL_SWAPIN(tp, 0);
554                 if (thread_pri != -1 && tp->t_state == TS_RUN &&
555                     (tp->t_schedflag & TS_LOAD) == 0) {
556                         size_t stack_size;
557                         pgcnt_t stack_pages;

559                         ASSERT((tp->t_schedflag & TS_ON_SWAPQ) == 0);

561                         thread_unlock(tp);
562                         /*
563                          * Now drop the p_lock since the stack needs
564                          * to brought in.
565                          */
566                         mutex_exit(&pp->p_lock);

568                         stack_size = swapsize(tp->t_swap);
569                         stack_pages = btopr(stack_size);
570                         /* Kernel probe */
571                         TNF_PROBE_4(swapin_lwp, "vm swap swapin", /* CSTYLED */,
572                             tnf_pid,            pid,            pp->p_pid,
573                             tnf_lwpid,          lwpid,          tp->t_tid,
574                             tnf_kthread_id,     tid,            tp,
575                             tnf_ulong,          page_count,     stack_pages);

577                         rw_enter(&kas.a_lock, RW_READER);
578                         err = segkp_fault(segkp->s_as->a_hat, segkp,
579                             tp->t_swap, stack_size, F_SOFTLOCK, S_OTHER);
580                         rw_exit(&kas.a_lock);
```



```
582                         /*
583                          * Re-acquire the p_lock.
584                          */
585                         mutex_enter(&pp->p_lock);
586                         if (err) {
587                                 num_swapped_in = 0;
588                                 break;
589                         } else {
590 #ifdef __sparc
591                                 lwp_swapin(tp);
592 #endif /* __sparc */
593                                 CPU_STATS_ADDQ(cpup, vm, swapin, 1);
594                                 CPU_STATS_ADDQ(cpup, vm, pgswapin,
595                                     stack_pages);

597                                 pp->p_swapcnt--;
598                                 pp->p_swrss -= stack_pages;

600                                 thread_lock(tp);
601                                 tp->t_schedflag |= TS_LOAD;
602                                 dq_sruninc(tp);

604                                 /* set swapin time */
605                                 tp->t_stime = ddi_get_lbolt();
606                                 thread_unlock(tp);

608                                 nswapped--;
609                                 tot_swapped_in++;
610                                 num_swapped_in++;

612                                 TRACE_2(TR_FAC_SCHED, TR_SWAPIN,
613                                     "swapin: pp %p stack_pages %lu",
614                                     pp, stack_pages);
615                                 goto top;
616                         }
617                 }
618                 thread_unlock(tp);
619         } while ((tp = tp->t_forw) != pp->p_tlist);
620         return (num_swapped_in);
621 }

623 /*
624  * Swap out lwps.  Returns nonzero on success (i.e., if at least one lwp is
625  * swapped out) and 0 on failure.
626  */
627 static int
628 swapout(proc_t *pp, uint_t *swrss, int swapflags)
629 {
630         kthread_id_t tp;
631         pgcnt_t ws_pages = 0;
632         int err;
633         int swapped_lwps = 0;
634         struct as *as = pp->p_as;
635         struct cpu *cpup = CPU;
636         pri_t thread_pri;

638         ASSERT(MUTEX_HELD(&pp->p_lock));

640         if (pp->p_flag & SEXITING)
641                 return (0);

643 top:
644         tp = pp->p_tlist;
645         do {
646                 klwp_t *lwp = ttolwp(tp);
```

```
648                 /*
649                  * Swapout eligible lwps (specified by the scheduling
650                  * class) which don't have TS_DONT_SWAP set.  Set the
651                  * "intent to swap" flag (TS_SWAPENQ) on threads
652                  * which have TS_DONT_SWAP set so that they can be
653                  * swapped if and when they reach a safe point.
654                  */
655                 thread_lock(tp);
656                 thread_pri = CL_SWAPOUT(tp, swapflags);
657                 if (thread_pri != -1) {
658                         if (tp->t_schedflag & TS_DONT_SWAP) {
659                                 tp->t_schedflag |= TS_SWAPENQ;
660                                 tp->t_trapret = 1;
661                                 aston(tp);
662                         } else {
663                                 pgcnt_t stack_pages;
664                                 size_t stack_size;
665
666                                 ASSERT((tp->t_schedflag &
667                                     (TS_DONT_SWAP | TS_LOAD)) == TS_LOAD);
668
669                                 if (lock_try(&tp->t_lock)) {
670                                         /*
671                                          * Remove thread from the swap_queue.
672                                          */
673                                         if (tp->t_schedflag & TS_ON_SWAPQ) {
674                                                 ASSERT(!(tp->t_schedflag &
675                                                     TS_SWAPENQ));
676                                                 swapdeq(tp);
677                                         } else if (tp->t_state == TS_RUN)
678                                                 dq_srundec(tp);
679
680                                         tp->t_schedflag &=
681                                             ~(TS_LOAD | TS_SWAPENQ);
682                                         lock_clear(&tp->t_lock);
683
684                                         /*
685                                          * Set swapout time if the thread isn't
686                                          * sleeping.
687                                          */
688                                         if (tp->t_state != TS_SLEEP)
689                                                 tp->t_stime = ddi_get_lbolt();
690                                         thread_unlock(tp);
691
692                                         nswapped++;
693                                         tot_swapped_out++;
694
695                                         lwp->lwp_ru.nswap++;
696
697                                         /*
698                                          * Now drop the p_lock since the
699                                          * stack needs to pushed out.
700                                          */
701                                         mutex_exit(&pp->p_lock);
702
703                                         stack_size = swapsize(tp->t_swap);
704                                         stack_pages = btopr(stack_size);
705                                         ws_pages += stack_pages;
706                                         /* Kernel probe */
707                                         TNF_PROBE_4(swapout_lwp,
708                                             "vm swap swapout",
709                                             /* CSTYLED */,
710                                             tnf_pid, pid, pp->p_pid,
711                                             tnf_lwpid, lwpid, tp->t_tid,
712                                             tnf_kthread_id, tid, tp,
```



```
713                                             tnf_ulong, page_count,
714                                             stack_pages);
715
716                                         rw_enter(&kas.a_lock, RW_READER);
717                                         err = segkp_fault(segkp->s_as->a_hat,
718                                             segkp, tp->t_swap, stack_size,
719                                             F_SOFTUNLOCK, S_WRITE);
720                                         rw_exit(&kas.a_lock);
721
722                                         if (err) {
723                                                 cmn_err(CE_PANIC,
724                                                     "swapout: segkp_fault "
725                                                     "failed err: %d", err);
726                                         }
727                                         CPU_STATS_ADDQ(cpup,
728                                             vm, pgswapout, stack_pages);
729
730                                         mutex_enter(&pp->p_lock);
731                                         pp->p_swapcnt++;
732                                         swapped_lwps++;
733                                         goto top;
734                                 }
735                         }
736                 }
737                 thread_unlock(tp);
738         } while ((tp = tp->t_forw) != pp->p_tlist);
739
740         /*
741          * Unload address space when all lwps are swapped out.
742          */
743         if (pp->p_swapcnt == pp->p_lwpcnt) {
744                 size_t as_size = 0;
745
746                 /*
747                  * Avoid invoking as_swapout() if the process has
748                  * no MMU resources since pageout will eventually
749                  * steal pages belonging to this address space.  This
750                  * saves CPU cycles as the number of pages that are
751                  * potentially freed or pushed out by the segment
752                  * swapout operation is very small.
753                  */
754                 if (rm_asrss(pp->p_as) != 0)
755                         as_size = as_swapout(as);
756
757                 CPU_STATS_ADDQ(cpup, vm, pgswapout, btop(as_size));
758                 CPU_STATS_ADDQ(cpup, vm, swapout, 1);
759                 ws_pages += btop(as_size);
760
761                 TRACE_2(TR_FAC_SCHED, TR_SWAPOUT,
762                     "swapout: pp %p pages_pushed %lu", pp, ws_pages);
763                 /* Kernel probe */
764                 TNF_PROBE_2(swapout_process, "vm swap swapout", /* CSTYLED */,
765                     tnf_pid,    pid,            pp->p_pid,
766                     tnf_ulong,  page_count,     ws_pages);
767         }
768         *swrss = ws_pages;
769         return (swapped_lwps);
770 }
771
772 void
773 swapout_lwp(klwp_t *lwp)
774 {
775         kthread_id_t tp = curthread;
776
777         ASSERT(curthread == lwptot(lwp));
```

```
779         /*
780          * Don't insert the thread onto the swap queue if
781          * sufficient memory is available.
782          */
783         if (avefree > desfree || avefree < desfree && freemem > desfree) {
784                 thread_lock(tp);
785                 tp->t_schedflag &= ~TS_SWAPENQ;
786                 thread_unlock(tp);
787                 return;
788         }
789
790         /*
791          * Lock the thread, then move it to the swapped queue from the
792          * onproc queue and set its state to be TS_RUN.
793          */
794         thread_lock(tp);
795         ASSERT(tp->t_state == TS_ONPROC);
796         if (tp->t_schedflag & TS_SWAPENQ) {
797                 tp->t_schedflag &= ~TS_SWAPENQ;
798
799                 /*
800                  * Set the state of this thread to be runnable
801                  * and move it from the onproc queue to the swap queue.
802                  */
803                 disp_swapped_enq(tp);
804
805                 /*
806                  * Insert the thread onto the swap queue.
807                  */
808                 tp->t_link = tswap_queue;
809                 tswap_queue = tp;
810                 tp->t_schedflag |= TS_ON_SWAPQ;
811
812                 thread_unlock_nopreempt(tp);
813
814                 TRACE_1(TR_FAC_SCHED, TR_SWAPOUT_LWP, "swapout_lwp:%x", lwp);
815
816                 swtch();
817         } else {
818                 thread_unlock(tp);
819         }
820 }
821
822 /*
823  * Swap all threads on the swap queue.
824  */
825 static void
826 process_swap_queue(void)
827 {
828         kthread_id_t tp;
829         uint_t ws_pages;
830         proc_t *pp;
831         struct cpu *cpup = CPU;
832         klwp_t *lwp;
833         int err;
834
835         if (tswap_queue == NULL)
836                 return;
837
838         /*
839          * Acquire the "swapped_lock" which locks the swap queue,
840          * and unload the stacks of all threads on it.
841          */
842         disp_lock_enter(&swapped_lock);
843         while ((tp = tswap_queue) != NULL) {
844                 pgcnt_t stack_pages;
```



```
845                 size_t stack_size;
846
847                 tswap_queue = tp->t_link;
848                 tp->t_link = NULL;
849
850                 /*
851                  * Drop the "dispatcher lock" before acquiring "t_lock"
852                  * to avoid spinning on it since the thread at the front
853                  * of the swap queue could be pinned before giving up
854                  * its "t_lock" in resume.
855                  */
856                 disp_lock_exit(&swapped_lock);
857                 lock_set(&tp->t_lock);
858
859                 /*
860                  * Now, re-acquire the "swapped_lock".  Acquiring this lock
861                  * results in locking the thread since its dispatcher lock
862                  * (t_lockp) is the "swapped_lock".
863                  */
864                 disp_lock_enter(&swapped_lock);
865                 ASSERT(tp->t_state == TS_RUN);
866                 ASSERT(tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ));
867
868                 tp->t_schedflag &= ~(TS_LOAD | TS_ON_SWAPQ);
869                 tp->t_stime = ddi_get_lbolt();          /* swapout time */
870                 disp_lock_exit(&swapped_lock);
871                 lock_clear(&tp->t_lock);
872
873                 lwp = ttolwp(tp);
874                 lwp->lwp_ru.nswap++;
875
876                 pp = ttoproc(tp);
877                 stack_size = swapsize(tp->t_swap);
878                 stack_pages = btopr(stack_size);
879
880                 /* Kernel probe */
881                 TNF_PROBE_4(swapout_lwp, "vm swap swapout", /* CSTYLED */,
882                     tnf_pid,            pid,            pp->p_pid,
883                     tnf_lwpid,          lwpid,          tp->t_tid,
884                     tnf_kthread_id,     tid,            tp,
885                     tnf_ulong,          page_count,     stack_pages);
886
887                 rw_enter(&kas.a_lock, RW_READER);
888                 err = segkp_fault(segkp->s_as->a_hat, segkp, tp->t_swap,
889                     stack_size, F_SOFTUNLOCK, S_WRITE);
890                 rw_exit(&kas.a_lock);
891
892                 if (err) {
893                         cmn_err(CE_PANIC,
894                         "process_swap_list: segkp_fault failed err: %d", err);
895                 }
896                 CPU_STATS_ADDQ(cpup, vm, pgswapout, stack_pages);
897
898                 nswapped++;
899                 tot_swapped_out++;
900                 swapqswap++;
901
902                 /*
903                  * Don't need p_lock since the swapper is the only
904                  * thread which increments/decrements p_swapcnt and p_swrss.
905                  */
906                 ws_pages = stack_pages;
907                 pp->p_swapcnt++;
908
909                 TRACE_1(TR_FAC_SCHED, TR_SWAPQ_LWP, "swaplist: pp %p", pp);
```

```
911                 /*
912                  * Unload address space when all lwps are swapped out.
913                  */
914                 if (pp->p_swapcnt == pp->p_lwpcnt) {
915                         size_t as_size = 0;

917                         if (rm_asrss(pp->p_as) != 0)
918                                 as_size = as_swapout(pp->p_as);

920                         CPU_STATS_ADDQ(cpup, vm, pgswapout,
921                             btop(as_size));
922                         CPU_STATS_ADDQ(cpup, vm, swapout, 1);

924                         ws_pages += btop(as_size);

926                         TRACE_2(TR_FAC_SCHED, TR_SWAPQ_PROC,
927                             "swaplist_proc: pp %p pages_pushed: %lu",
928                             pp, ws_pages);
929                         /* Kernel probe */
930                         TNF_PROBE_2(swapout_process, "vm swap swapout",
931                             /* CSTYLED */,
932                             tnf_pid,    pid,            pp->p_pid,
933                             tnf_ulong,  page_count,     ws_pages);
934                 }
935                 pp->p_swrss += ws_pages;
936                 disp_lock_enter(&swapped_lock);
100         }
938         disp_lock_exit(&swapped_lock);
101 }
```

_____unchanged_portion_omitted_

```
**********************************************************
  248852 Fri May  8 18:10:30 2015
new/usr/src/uts/common/os/sunddi.c
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

8219 /*
8220  * A consolidation private function which is essentially equivalent to
8221  * ddi_umem_lock but with the addition of arguments ops_vector and procp.
8222  * A call to as_add_callback is done if DDI_UMEMLOCK_LONGTERM is set, and
8223  * the ops_vector is valid.
8224  *
8225  * Lock the virtual address range in the current process and create a
8226  * ddi_umem_cookie (of type UMEM_LOCKED). This can be used to pass to
8227  * ddi_umem_iosetup to create a buf or do devmap_umem_setup/remap to export
8228  * to user space.
8229  *
8230  * Note: The resource control accounting currently uses a full charge model
8231  * in other words attempts to lock the same/overlapping areas of memory
8232  * will deduct the full size of the buffer from the projects running
8233  * counter for the device locked memory.
8234  *
8235  * addr, size should be PAGESIZE aligned
8236  *
8237  * flags - DDI_UMEMLOCK_READ, DDI_UMEMLOCK_WRITE or both
8238  *      identifies whether the locked memory will be read or written or both
8239  *      DDI_UMEMLOCK_LONGTERM  must be set when the locking will
8240  * be maintained for an indefinitely long period (essentially permanent),
8241  * rather than for what would be required for a typical I/O completion.
8242  * When DDI_UMEMLOCK_LONGTERM is set, umem_lockmemory will return EFAULT
8243  * if the memory pertains to a regular file which is mapped MAP_SHARED.
8244  * This is to prevent a deadlock if a file truncation is attempted after
8245  * after the locking is done.
8246  *
8247  * Returns 0 on success
8248  *      EINVAL - for invalid parameters
8249  *      EPERM, ENOMEM and other error codes returned by as_pagelock
8250  *      ENOMEM - is returned if the current request to lock memory exceeds
8251  *               *.max-locked-memory resource control value.
8252  *      EFAULT - memory pertains to a regular file mapped shared and
8253  *               and DDI_UMEMLOCK_LONGTERM flag is set
8254  *      EAGAIN - could not start the ddi_umem_unlock list processing thread
8255  */
8256 int
8257 umem_lockmemory(caddr_t addr, size_t len, int flags, ddi_umem_cookie_t *cookie,
8258                 struct umem_callback_ops *ops_vector,
8259                 proc_t *procp)
8260 {
8261         int     error;
8262         struct ddi_umem_cookie *p;
8263         void    (*driver_callback)() = NULL;
8264         struct as *as;
8265         struct seg              *seg;
8266         vnode_t                 *vp;

8268         /* Allow device drivers to not have to reference "curproc" */
8269         if (procp == NULL)
8270                 procp = curproc;
8271         as = procp->p_as;
8272         *cookie = NULL;         /* in case of any error return */

8274         /* These are the only three valid flags */
8275         if ((flags & ~(DDI_UMEMLOCK_READ | DDI_UMEMLOCK_WRITE |
```



```
8276             DDI_UMEMLOCK_LONGTERM)) != 0)
8277                 return (EINVAL);

8279         /* At least one (can be both) of the two access flags must be set */
8280         if ((flags & (DDI_UMEMLOCK_READ | DDI_UMEMLOCK_WRITE)) == 0)
8281                 return (EINVAL);

8283         /* addr and len must be page-aligned */
8284         if (((uintptr_t)addr & PAGEOFFSET) != 0)
8285                 return (EINVAL);

8287         if ((len & PAGEOFFSET) != 0)
8288                 return (EINVAL);

8290         /*
8291          * For longterm locking a driver callback must be specified; if
8292          * not longterm then a callback is optional.
8293          */
8294         if (ops_vector != NULL) {
8295                 if (ops_vector->cbo_umem_callback_version !=
8296                     UMEM_CALLBACK_VERSION)
8297                         return (EINVAL);
8298                 else
8299                         driver_callback = ops_vector->cbo_umem_lock_cleanup;
8300         }
8301         if ((driver_callback == NULL) && (flags & DDI_UMEMLOCK_LONGTERM))
8302                 return (EINVAL);

8304         /*
8305          * Call i_ddi_umem_unlock_thread_start if necessary.  It will
8306          * be called on first ddi_umem_lock or umem_lockmemory call.
8307          */
8308         if (ddi_umem_unlock_thread == NULL)
8309                 i_ddi_umem_unlock_thread_start();

8311         /* Allocate memory for the cookie */
8312         p = kmem_zalloc(sizeof (struct ddi_umem_cookie), KM_SLEEP);

8314         /* Convert the flags to seg_rw type */
8315         if (flags & DDI_UMEMLOCK_WRITE) {
8316                 p->s_flags = S_WRITE;
8317         } else {
8318                 p->s_flags = S_READ;
8319         }

8321         /* Store procp in cookie for later iosetup/unlock */
8322         p->procp = (void *)procp;

8324         /*
8325          * Store the struct as pointer in cookie for later use by
8326          * ddi_umem_unlock.  The proc->p_as will be stale if ddi_umem_unlock
8327          * is called after relvm is called.
8328          */
8329         p->asp = as;

8331         /*
8332          * The size field is needed for lockmem accounting.
8333          */
8334         p->size = len;
8335         init_lockedmem_rctl_flag(p);

8337         if (umem_incr_devlockmem(p) != 0) {
8338                 /*
8339                  * The requested memory cannot be locked
8340                  */
8341                 kmem_free(p, sizeof (struct ddi_umem_cookie));
```

```
8342                *cookie = (ddi_umem_cookie_t)NULL;
8343                return (ENOMEM);
8344        }

8346        /* Lock the pages corresponding to addr, len in memory */
8347        error = as_pagelock(as, &(p->pparray), addr, len, p->s_flags);
8348        if (error != 0) {
8349                umem_decr_devlockmem(p);
8350                kmem_free(p, sizeof (struct ddi_umem_cookie));
8351                *cookie = (ddi_umem_cookie_t)NULL;
8352                return (error);
8353        }

8355        /*
8356         * For longterm locking the addr must pertain to a seg_vn segment or
8357         * or a seg_spt segment.
8358         * If the segment pertains to a regular file, it cannot be
8359         * mapped MAP_SHARED.
8360         * This is to prevent a deadlock if a file truncation is attempted
8361         * after the locking is done.
8362         * Doing this after as_pagelock guarantees persistence of the as; if
8363         * an unacceptable segment is found, the cleanup includes calling
8364         * as_pageunlock before returning EFAULT.
8365         *
8366         * segdev is allowed here as it is already locked.  This allows
8367         * for memory exported by drivers through mmap() (which is already
8368         * locked) to be allowed for LONGTERM.
8369         */
8370        if (flags & DDI_UMEMLOCK_LONGTERM) {
8371                extern const struct seg_ops segspt_shmops;
8372                extern const struct seg_ops segdev_ops;
8371                extern  struct seg_ops segspt_shmops;
8372                extern  struct seg_ops segdev_ops;
8373                AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
8374                for (seg = as_segat(as, addr); ; seg = AS_SEGNEXT(as, seg)) {
8375                        if (seg == NULL || seg->s_base > addr + len)
8376                                break;
8377                        if (seg->s_ops == &segdev_ops)
8378                                continue;
8379                        if (((seg->s_ops != &segvn_ops) &&
8380                            (seg->s_ops != &segspt_shmops)) ||
8381                            ((segop_getvp(seg, addr, &vp) == 0 &&
8381                            ((SEGOP_GETVP(seg, addr, &vp) == 0 &&
8382                            vp != NULL && vp->v_type == VREG) &&
8383                            (segop_gettype(seg, addr) & MAP_SHARED))) {
8383                            (SEGOP_GETTYPE(seg, addr) & MAP_SHARED))) {
8384                                as_pageunlock(as, p->pparray,
8385                                    addr, len, p->s_flags);
8386                                AS_LOCK_EXIT(as, &as->a_lock);
8387                                umem_decr_devlockmem(p);
8388                                kmem_free(p, sizeof (struct ddi_umem_cookie));
8389                                *cookie = (ddi_umem_cookie_t)NULL;
8390                                return (EFAULT);
8391                        }
8392                }
8393                AS_LOCK_EXIT(as, &as->a_lock);
8394        }


8397        /* Initialize the fields in the ddi_umem_cookie */
8398        p->cvaddr = addr;
8399        p->type = UMEM_LOCKED;
8400        if (driver_callback != NULL) {
8401                /* i_ddi_umem_unlock and umem_lock_undo may need the cookie */
8402                p->cook_refcnt = 2;
8403                p->callbacks = *ops_vector;
```



```
8404        } else {
8405                /* only i_ddi_umme_unlock needs the cookie */
8406                p->cook_refcnt = 1;
8407        }

8409        *cookie = (ddi_umem_cookie_t)p;

8411        /*
8412         * If a driver callback was specified, add an entry to the
8413         * as struct callback list. The as_pagelock above guarantees
8414         * the persistence of as.
8415         */
8416        if (driver_callback) {
8417                error = as_add_callback(as, umem_lock_undo, p, AS_ALL_EVENT,
8418                    addr, len, KM_SLEEP);
8419                if (error != 0) {
8420                        as_pageunlock(as, p->pparray,
8421                            addr, len, p->s_flags);
8422                        umem_decr_devlockmem(p);
8423                        kmem_free(p, sizeof (struct ddi_umem_cookie));
8424                        *cookie = (ddi_umem_cookie_t)NULL;
8425                }
8426        }
8427        return (error);
8428 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   39428 Fri May  8 18:10:30 2015
new/usr/src/uts/common/os/timers.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

 622 /*
 623  * Real time profiling interval timer expired:
 624  * Increment microstate counters for each lwp in the process
 625  * and ensure that running lwps are kicked into the kernel.
 626  * If time is not set up to reload, then just return.
 627  * Else compute next time timer should go off which is > current time,
 628  * as above.
 629  */
 630 static void
 631 realprofexpire(void *arg)
 632 {
 633         struct proc *p = arg;
 634         kthread_t *t;

 636         mutex_enter(&p->p_lock);
 637         if (p->p_rprof_cyclic == CYCLIC_NONE ||
 638             (t = p->p_tlist) == NULL) {
 639                 mutex_exit(&p->p_lock);
 640                 return;
 641         }
 642         do {
 643                 int mstate;

 645                 /*
 646                  * Attempt to allocate the SIGPROF buffer, but don't sleep.
 647                  */
 648                 if (t->t_rprof == NULL)
 649                         t->t_rprof = kmem_zalloc(sizeof (struct rprof),
 650                             KM_NOSLEEP);
 651                 if (t->t_rprof == NULL)
 652                         continue;

 654                 thread_lock(t);
 655                 switch (t->t_state) {
 656                 case TS_SLEEP:
 657                         /*
 658                          * Don't touch the lwp is it is swapped out.
 659                          */
 660                         if (!(t->t_schedflag & TS_LOAD)) {
 661                                 mstate = LMS_SLEEP;
 662                                 break;
 663                         }
 657                         switch (mstate = ttolwp(t)->lwp_mstate.ms_prev) {
 658                         case LMS_TFAULT:
 659                         case LMS_DFAULT:
 660                         case LMS_KFAULT:
 661                         case LMS_USER_LOCK:
 662                                 break;
 663                         default:
 664                                 mstate = LMS_SLEEP;
 665                                 break;
 666                         }
 667                         break;
```



```
 668                 case TS_RUN:
 669                 case TS_WAIT:
 670                         mstate = LMS_WAIT_CPU;
 671                         break;
 672                 case TS_ONPROC:
 673                         switch (mstate = t->t_mstate) {
 674                         case LMS_USER:
 675                         case LMS_SYSTEM:
 676                         case LMS_TRAP:
 677                                 break;
 678                         default:
 679                                 mstate = LMS_SYSTEM;
 680                                 break;
 681                         }
 682                         break;
 683                 default:
 684                         mstate = t->t_mstate;
 685                         break;
 686                 }
 687                 t->t_rprof->rp_anystate = 1;
 688                 t->t_rprof->rp_state[mstate]++;
 689                 aston(t);
 690                 /*
 691                  * force the thread into the kernel
 692                  * if it is not already there.
 693                  */
 694                 if (t->t_state == TS_ONPROC && t->t_cpu != CPU)
 695                         poke_cpu(t->t_cpu->cpu_id);
 696                 thread_unlock(t);
 697         } while ((t = t->t_forw) != p->p_tlist);

 699         mutex_exit(&p->p_lock);
 700 }
______unchanged_portion_omitted_
```

```
**********************************************************
    8581 Fri May  8 18:10:31 2015
new/usr/src/uts/common/os/urw.c
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
patch lower-case-segops
**********************************************************
```

```
/*
 * 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.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

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

#include <sys/atomic.h>
#include <sys/errno.h>
#include <sys/stat.h>
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/systm.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/cpuvar.h>
#include <sys/kmem.h>
#include <sys/strsubr.h>
#include <sys/sysmacros.h>
#include <sys/frame.h>
#include <sys/stack.h>
#include <sys/proc.h>
#include <sys/priv.h>
#include <sys/policy.h>
#include <sys/ontrap.h>
#include <sys/vmsystm.h>
#include <sys/prsystm.h>

#include <vm/as.h>
#include <vm/seg.h>
#include <vm/seg_dev.h>
#include <vm/seg_vn.h>
#include <vm/seg_spt.h>
#include <vm/seg_kmem.h>

extern const struct seg_ops segdev_ops; /* needs a header file */
```



```
extern const struct seg_ops segspt_shmops;      /* needs a header file */
extern struct seg_ops segdev_ops;       /* needs a header file */
extern struct seg_ops segspt_shmops;    /* needs a header file */

static int
page_valid(struct seg *seg, caddr_t addr)
{
        struct segvn_data *svd;
        vnode_t *vp;
        vattr_t vattr;

        /*
         * Fail if the page doesn't map to a page in the underlying
         * mapped file, if an underlying mapped file exists.
         */
        vattr.va_mask = AT_SIZE;
        if (seg->s_ops == &segvn_ops &&
            segop_getvp(seg, addr, &vp) == 0 &&
            SEGOP_GETVP(seg, addr, &vp) == 0 &&
            vp != NULL && vp->v_type == VREG &&
            VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
                u_offset_t size = roundup(vattr.va_size, (u_offset_t)PAGESIZE);
                u_offset_t offset = segop_getoffset(seg, addr);
                u_offset_t offset = SEGOP_GETOFFSET(seg, addr);

                if (offset >= size)
                        return (0);
        }

        /*
         * Fail if this is an ISM shared segment and the address is
         * not within the real size of the spt segment that backs it.
         */
        if (seg->s_ops == &segspt_shmops &&
            addr >= seg->s_base + spt_realsize(seg))
                return (0);

        /*
         * Fail if the segment is mapped from /dev/null.
         * The key is that the mapping comes from segdev and the
         * type is neither MAP_SHARED nor MAP_PRIVATE.
         */
        if (seg->s_ops == &segdev_ops &&
            ((segop_gettype(seg, addr) & (MAP_SHARED | MAP_PRIVATE)) == 0))
            ((SEGOP_GETTYPE(seg, addr) & (MAP_SHARED | MAP_PRIVATE)) == 0))
                return (0);

        /*
         * Fail if the page is a MAP_NORESERVE page that has
         * not actually materialized.
         * We cheat by knowing that segvn is the only segment
         * driver that supports MAP_NORESERVE.
         */
        if (seg->s_ops == &segvn_ops &&
            (svd = (struct segvn_data *)seg->s_data) != NULL &&
            (svd->vp == NULL || svd->vp->v_type != VREG) &&
            (svd->flags & MAP_NORESERVE)) {
                /*
                 * Guilty knowledge here.  We know that
                 * segvn_incore returns more than just the
                 * low-order bit that indicates the page is
                 * actually in memory.  If any bits are set,
                 * then there is backing store for the page.
                 */
                char incore = 0;
                (void) segop_incore(seg, addr, PAGESIZE, &incore);
```

```
120                 (void) SEGOP_INCORE(seg, addr, PAGESIZE, &incore);
121                 if (incore == 0)
122                         return (0);
123         }
124         return (1);
125 }
_____unchanged_portion_omitted_

178 /*
179  * Perform I/O to a given process. This will return EIO if we detect
180  * corrupt memory and ENXIO if there is no such mapped address in the
181  * user process's address space.
182  */
183 static int
184 urw(proc_t *p, int writing, void *buf, size_t len, uintptr_t a)
185 {
186         caddr_t addr = (caddr_t)a;
187         caddr_t page;
188         caddr_t vaddr;
189         struct seg *seg;
190         int error = 0;
191         int err = 0;
192         uint_t prot;
193         uint_t prot_rw = writing ? PROT_WRITE : PROT_READ;
194         int protchanged;
195         on_trap_data_t otd;
196         int retrycnt;
197         struct as *as = p->p_as;
198         enum seg_rw rw;

200         /*
201          * Locate segment containing address of interest.
202          */
203         page = (caddr_t)(uintptr_t)((uintptr_t)addr & PAGEMASK);
204         retrycnt = 0;
205         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
206 retry:
207         if ((seg = as_segat(as, page)) == NULL ||
208             !page_valid(seg, page)) {
209                 AS_LOCK_EXIT(as, &as->a_lock);
210                 return (ENXIO);
211         }
212         segop_getprot(seg, page, 0, &prot);
212         SEGOP_GETPROT(seg, page, 0, &prot);

214         protchanged = 0;
215         if ((prot & prot_rw) == 0) {
216                 protchanged = 1;
217                 err = segop_setprot(seg, page, PAGESIZE, prot | prot_rw);
217                 err = SEGOP_SETPROT(seg, page, PAGESIZE, prot | prot_rw);

219                 if (err == IE_RETRY) {
220                         protchanged = 0;
221                         ASSERT(retrycnt == 0);
222                         retrycnt++;
223                         goto retry;
224                 }

226                 if (err != 0) {
227                         AS_LOCK_EXIT(as, &as->a_lock);
228                         return (ENXIO);
229                 }
230         }

232         /*
233          * segvn may do a copy-on-write for F_SOFTLOCK/S_READ case to break
```



```
234          * sharing to avoid a copy on write of a softlocked page by another
235          * thread. But since we locked the address space as a writer no other
236          * thread can cause a copy on write. S_READ_NOCOW is passed as the
237          * access type to tell segvn that it's ok not to do a copy-on-write
238          * for this SOFTLOCK fault.
239          */
240         if (writing)
241                 rw = S_WRITE;
242         else if (seg->s_ops == &segvn_ops)
243                 rw = S_READ_NOCOW;
244         else
245                 rw = S_READ;

247         if (segop_fault(as->a_hat, seg, page, PAGESIZE, F_SOFTLOCK, rw)) {
247         if (SEGOP_FAULT(as->a_hat, seg, page, PAGESIZE, F_SOFTLOCK, rw)) {
248                 if (protchanged)
249                         (void) segop_setprot(seg, page, PAGESIZE, prot);
249                         (void) SEGOP_SETPROT(seg, page, PAGESIZE, prot);
250                 AS_LOCK_EXIT(as, &as->a_lock);
251                 return (ENXIO);
252         }
253         CPU_STATS_ADD_K(vm, softlock, 1);

255         /*
256          * Make sure we're not trying to read or write off the end of the page.
257          */
258         ASSERT(len <= page + PAGESIZE - addr);

260         /*
261          * Map in the locked page, copy to our local buffer,
262          * then map the page out and unlock it.
263          */
264         vaddr = mapin(as, addr, writing);

266         /*
267          * Since we are copying memory on behalf of the user process,
268          * protect against memory error correction faults.
269          */
270         if (!on_trap(&otd, OT_DATA_EC)) {
271                 if (seg->s_ops == &segdev_ops) {
272                         /*
273                          * Device memory can behave strangely; invoke
274                          * a segdev-specific copy operation instead.
275                          */
276                         if (writing) {
277                                 if (segdev_copyto(seg, addr, buf, vaddr, len))
278                                         error = ENXIO;
279                         } else {
280                                 if (segdev_copyfrom(seg, addr, vaddr, buf, len))
281                                         error = ENXIO;
282                         }
283                 } else {
284                         if (writing)
285                                 bcopy(buf, vaddr, len);
286                         else
287                                 bcopy(vaddr, buf, len);
288                 }
289         } else {
290                 error = EIO;
291         }
292         no_trap();

294         /*
295          * If we're writing to an executable page, we may need to sychronize
296          * the I$ with the modifications we made through the D$.
297          */
```

```
298         if (writing && (prot & PROT_EXEC))
299                 sync_icache(vaddr, (uint_t)len);

301         mapout(as, addr, vaddr, writing);

303         if (rw == S_READ_NOCOW)
304                 rw = S_READ;

306         (void) segop_fault(as->a_hat, seg, page, PAGESIZE, F_SOFTUNLOCK, rw);
306         (void) SEGOP_FAULT(as->a_hat, seg, page, PAGESIZE, F_SOFTUNLOCK, rw);

308         if (protchanged)
309                 (void) segop_setprot(seg, page, PAGESIZE, prot);
309                 (void) SEGOP_SETPROT(seg, page, PAGESIZE, prot);

311         AS_LOCK_EXIT(as, &as->a_lock);

313         return (error);
314 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   14034 Fri May  8 18:10:31 2015
new/usr/src/uts/common/os/vm_subr.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

#define MAX_MAPIN_PAGES 8

/*
 * This function temporarily "borrows" user pages for kernel use. If
 * "cow" is on, it also sets up copy-on-write protection (only feasible
 * on MAP_PRIVATE segment) on the user mappings, to protect the borrowed
 * pages from any changes by the user. The caller is responsible for
 * unlocking and tearing down cow settings when it's done with the pages.
 * For an example, see kcfree().
 *
 * Pages behind [uaddr..uaddr+*lenp] under address space "as" are locked
 * (shared), and mapped into kernel address range [kaddr..kaddr+*lenp] if
 * kaddr != -1. On entering this function, cached_ppp contains a list
 * of pages that are mapped into [kaddr..kaddr+*lenp] already (from a
 * previous call). Thus if same pages remain behind [uaddr..uaddr+*lenp],
 * the kernel map won't need to be reloaded again.
 *
 * For cow == 1, if the pages are anonymous pages, it also bumps the anon
 * reference count, and change the user-mapping to read-only. This
 * scheme should work on all types of segment drivers. But to be safe,
 * we check against segvn here.
 *
 * Since this function is used to emulate copyin() semantic, it checks
 * to make sure the user-mappings allow "user-read".
 *
 * On exit "lenp" contains the number of bytes successfully locked and
 * mapped in. For the unsuccessful ones, the caller can fall back to
 * copyin().
 *
 * Error return:
 * ENOTSUP - operation like this is not supported either on this segment
 * type, or on this platform type.
 */
int
cow_mapin(struct as *as, caddr_t uaddr, caddr_t kaddr, struct page **cached_ppp,
    struct anon **app, size_t *lenp, int cow)
{
        struct          hat *hat;
        struct seg      *seg;
        caddr_t         base;
        page_t          *pp, *ppp[MAX_MAPIN_PAGES];
        long            i;
        int             flags;
        size_t          size, total = *lenp;
        char            first = 1;
        faultcode_t     res;

        *lenp = 0;
        if (cow) {
                AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
                seg = as_findseg(as, uaddr, 0);
                if ((seg == NULL) || ((base = seg->s_base) > uaddr) ||
                    (uaddr + total) > base + seg->s_size) {
                        AS_LOCK_EXIT(as, &as->a_lock);
                        return (EINVAL);
                }
                /*
                 * The COW scheme should work for all segment types.
                 * But to be safe, we check against segvn.
```



```
                 */
                if (seg->s_ops != &segvn_ops) {
                        AS_LOCK_EXIT(as, &as->a_lock);
                        return (ENOTSUP);
                } else if ((segop_gettype(seg, uaddr) & MAP_PRIVATE) == 0) {
                } else if ((SEGOP_GETTYPE(seg, uaddr) & MAP_PRIVATE) == 0) {
                        AS_LOCK_EXIT(as, &as->a_lock);
                        return (ENOTSUP);
                }
        }
        hat = as->a_hat;
        size = total;
tryagain:
        /*
         * If (cow), hat_softlock will also change the usr protection to RO.
         * This is the first step toward setting up cow. Before we
         * bump up an_refcnt, we can't allow any cow-fault on this
         * address. Otherwise segvn_fault will change the protection back
         * to RW upon seeing an_refcnt == 1.
         * The solution is to hold the writer lock on "as".
         */
        res = hat_softlock(hat, uaddr, &size, &ppp[0], cow ? HAT_COW : 0);
        size = total - size;
        *lenp += size;
        size = size >> PAGESHIFT;
        i = 0;
        while (i < size) {
                pp = ppp[i];
                if (cow) {
                        kmutex_t *ahm;
                        /*
                         * Another solution is to hold SE_EXCL on pp, and
                         * disable PROT_WRITE. This also works for MAP_SHARED
                         * segment. The disadvantage is that it locks the
                         * page from being used by anybody else.
                         */
                        ahm = AH_MUTEX(pp->p_vnode, pp->p_offset);
                        mutex_enter(ahm);
                        *app = swap_anon(pp->p_vnode, pp->p_offset);
                        /*
                         * Since we are holding the as lock, this avoids a
                         * potential race with anon_decref. (segvn_unmap and
                         * segvn_free needs the as writer lock to do anon_free.)
                         */
                        if (*app != NULL) {
#if 0
                                if ((*app)->an_refcnt == 0)
                                /*
                                 * Consider the following senario (unlikey
                                 * though):
                                 * 1. an_refcnt == 2
                                 * 2. we solftlock the page.
                                 * 3. cow ocurrs on this addr. So a new ap,
                                 * page and mapping is established on addr.
                                 * 4. an_refcnt drops to 1 (segvn_faultpage
                                 * -> anon_decref(oldap))
                                 * 5. the last ref to ap also drops (from
                                 * another as). It ends up blocked inside
                                 * anon_decref trying to get page's excl lock.
                                 * 6. Later kcfree unlocks the page, call
                                 * anon_decref -> oops, ap is gone already.
                                 *
                                 * Holding as writer lock solves all problems.
                                 */
                                        *app = NULL;
                                else
```

```
#endif
                                        (*app)->an_refcnt++;
                        }
                        mutex_exit(ahm);
                } else {
                        *app = NULL;
                }
                if (kaddr != (caddr_t)-1) {
                        if (pp != *cached_ppp) {
                                if (*cached_ppp == NULL)
                                        flags = HAT_LOAD_LOCK | HAT_NOSYNC |
                                            HAT_LOAD_NOCONSIST;
                                else
                                        flags = HAT_LOAD_REMAP |
                                            HAT_LOAD_NOCONSIST;
                                /*
                                 * In order to cache the kernel mapping after
                                 * the user page is unlocked, we call
                                 * hat_devload instead of hat_memload so
                                 * that the kernel mapping we set up here is
                                 * "invisible" to the rest of the world. This
                                 * is not very pretty. But as long as the
                                 * caller bears the responsibility of keeping
                                 * cache consistency, we should be ok -
                                 * HAT_NOCONSIST will get us a uncached
                                 * mapping on VAC. hat_softlock will flush
                                 * a VAC_WRITEBACK cache. Therefore the kaddr
                                 * doesn't have to be of the same vcolor as
                                 * uaddr.
                                 * The alternative is - change hat_devload
                                 * to get a cached mapping. Allocate a kaddr
                                 * with the same vcolor as uaddr. Then
                                 * hat_softlock won't need to flush the VAC.
                                 */
                                hat_devload(kas.a_hat, kaddr, PAGESIZE,
                                    page_pptonum(pp), PROT_READ, flags);
                                *cached_ppp = pp;
                        }
                        kaddr += PAGESIZE;
                }
                cached_ppp++;
                app++;
                ++i;
        }
        if (cow) {
                AS_LOCK_EXIT(as, &as->a_lock);
        }
        if (first && res == FC_NOMAP) {
                /*
                 * If the address is not mapped yet, we call as_fault to
                 * fault the pages in. We could've fallen back to copy and
                 * let it fault in the pages. But for a mapped file, we
                 * normally reference each page only once. For zero-copy to
                 * be of any use, we'd better fall in the page now and try
                 * again.
                 */
                first = 0;
                size = size << PAGESHIFT;
                uaddr += size;
                total -= size;
                size = total;
                res = as_fault(as->a_hat, as, uaddr, size, F_INVAL, S_READ);
                if (cow)
                        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
                goto tryagain;
        }
```



```
        switch (res) {
        case FC_NOSUPPORT:
                return (ENOTSUP);
        case FC_PROT:   /* Pretend we don't know about it. This will be */
                        /* caught by the caller when uiomove fails. */
        case FC_NOMAP:
        case FC_OBJERR:
        default:
                return (0);
        }
}
```

_____***unchanged_portion_omitted_***

```
*********************************************************
   10025 Fri May  8 18:10:31 2015
new/usr/src/uts/common/os/waitq.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
*********************************************************
______unchanged_portion_omitted_

 197 /*
 198  * Put specified thread to specified wait queue without dropping thread's lock.
 199  * Returns 1 if thread was successfully placed on project's wait queue, or
 200  * 0 if wait queue is blocked.
 201  */
 202 int
 203 waitq_enqueue(waitq_t *wq, kthread_t *t)
 204 {
 205         ASSERT(THREAD_LOCK_HELD(t));
 206         ASSERT(t->t_sleepq == NULL);
 207         ASSERT(t->t_waitq == NULL);
 208         ASSERT(t->t_link == NULL);

 210         disp_lock_enter_high(&wq->wq_lock);

 212         /*
 213          * Can't enqueue anything on a blocked wait queue
 214          */
 215         if (wq->wq_blocked) {
 216                 disp_lock_exit_high(&wq->wq_lock);
 217                 return (0);
 218         }

 220         /*
 221          * Mark the time when thread is placed on wait queue. The microstate
 222          * accounting code uses this timestamp to determine wait times.
 223          */
 224         t->t_waitrq = gethrtime_unscaled();

 226         /*
 227          * Mark thread as not swappable.  If necessary, it will get
 228          * swapped out when it returns to the userland.
 229          */
 230         t->t_schedflag |= TS_DONT_SWAP;
 226         DTRACE_SCHED1(cpucaps__sleep, kthread_t *, t);
 227         waitq_link(wq, t);

 229         THREAD_WAIT(t, &wq->wq_lock);
 230         return (1);
 231 }
______unchanged_portion_omitted_
```

```
**********************************************************
   38530 Fri May  8 18:10:31 2015
new/usr/src/uts/common/os/watchpoint.c
patch lower-case-segops
remove xhat
The xhat infrastructure was added to support hardware such as the zulu
graphics card - hardware which had on-board MMUs.  The VM used the xhat code
to keep the CPU's and Zulu's page tables in-sync.  Since the only xhat user
was zulu (which is gone), we can safely remove it simplifying the whole VM
subsystem.
Assorted notes:
- AS_BUSY flag was used solely by xhat
**********************************************************
______unchanged_portion_omitted_
```

```
153 #define X       0
154 #define W       1
155 #define R       2
156 #define sum(a)  (a[X] + a[W] + a[R])

158 /*
159  * Common code for pr_mappage() and pr_unmappage().
160  */
161 static int
162 pr_do_mappage(caddr_t addr, size_t size, int mapin, enum seg_rw rw, int kernel)
163 {
164         proc_t *p = curproc;
165         struct as *as = p->p_as;
166         char *eaddr = addr + size;
167         int prot_rw = rw_to_prot(rw);
168         int xrw = rw_to_index(rw);
169         int rv = 0;
170         struct watched_page *pwp;
171         struct watched_page tpw;
172         avl_index_t where;
173         uint_t prot;

175         ASSERT(as != &kas);

177 startover:
178         ASSERT(rv == 0);
179         if (avl_numnodes(&as->a_wpage) == 0)
180                 return (0);

182         /*
183          * as->a_wpage can only be changed while the process is totally stopped.
184          * Don't grab p_lock here.  Holding p_lock while grabbing the address
185          * space lock leads to deadlocks with the clock thread.
185          * space lock leads to deadlocks with the clock thread.  Note that if an
186          * as_fault() is servicing a fault to a watched page on behalf of an
187          * XHAT provider, watchpoint will be temporarily cleared (and wp_prot
188          * will be set to wp_oprot).  Since this is done while holding as writer
189          * lock, we need to grab as lock (reader lock is good enough).
186          *
187          * p_maplock prevents simultaneous execution of this function.  Under
188          * normal circumstances, holdwatch() will stop all other threads, so the
189          * lock isn't really needed.  But there may be multiple threads within
190          * stop() when SWATCHOK is set, so we need to handle multiple threads
191          * at once.  See holdwatch() for the details of this dance.
192          */

194         mutex_enter(&p->p_maplock);
195         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

197         tpw.wp_vaddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
198         if ((pwp = avl_find(&as->a_wpage, &tpw, &where)) == NULL)
```



```
199                 pwp = avl_nearest(&as->a_wpage, where, AVL_AFTER);

201         for (; pwp != NULL && pwp->wp_vaddr < eaddr;
202                 pwp = AVL_NEXT(&as->a_wpage, pwp)) {

204                 /*
205                  * If the requested protection has not been
206                  * removed, we need not remap this page.
207                  */
208                 prot = pwp->wp_prot;
209                 if (kernel || (prot & PROT_USER))
210                         if (prot & prot_rw)
211                                 continue;
212                 /*
213                  * If the requested access does not exist in the page's
214                  * original protections, we need not remap this page.
215                  * If the page does not exist yet, we can't test it.
216                  */
217                 if ((prot = pwp->wp_oprot) != 0) {
218                         if (!(kernel || (prot & PROT_USER)))
219                                 continue;
220                         if (!(prot & prot_rw))
221                                 continue;
222                 }

224                 if (mapin) {
225                         /*
226                          * Before mapping the page in, ensure that
227                          * all other lwps are held in the kernel.
228                          */
229                         if (p->p_mapcnt == 0) {
230                                 /*
231                                  * Release as lock while in holdwatch()
232                                  * in case other threads need to grab it.
233                                  */
234                                 AS_LOCK_EXIT(as, &as->a_lock);
235                                 mutex_exit(&p->p_maplock);
236                                 if (holdwatch() != 0) {
237                                         /*
238                                          * We stopped in holdwatch().
239                                          * Start all over again because the
240                                          * watched page list may have changed.
241                                          */
242                                         goto startover;
243                                 }
244                                 mutex_enter(&p->p_maplock);
245                                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
246                         }
247                         p->p_mapcnt++;
248                 }

250                 addr = pwp->wp_vaddr;
251                 rv++;

253                 prot = pwp->wp_prot;
254                 if (mapin) {
255                         if (kernel)
256                                 pwp->wp_kmap[xrw]++;
257                         else
258                                 pwp->wp_umap[xrw]++;
259                         pwp->wp_flags |= WP_NOWATCH;
260                         if (pwp->wp_kmap[X] + pwp->wp_umap[X])
261                                 /* cannot have exec-only protection */
262                                 prot |= PROT_READ|PROT_EXEC;
263                         if (pwp->wp_kmap[R] + pwp->wp_umap[R])
264                                 prot |= PROT_READ;
```

```
                        if (pwp->wp_kmap[W] + pwp->wp_umap[W])
                                /* cannot have write-only protection */
                                prot |= PROT_READ|PROT_WRITE;
#if 0   /* damned broken mmu feature! */
                        if (sum(pwp->wp_umap) == 0)
                                prot &= ~PROT_USER;
#endif
                } else {
                        ASSERT(pwp->wp_flags & WP_NOWATCH);
                        if (kernel) {
                                ASSERT(pwp->wp_kmap[xrw] != 0);
                                --pwp->wp_kmap[xrw];
                        } else {
                                ASSERT(pwp->wp_umap[xrw] != 0);
                                --pwp->wp_umap[xrw];
                        }
                        if (sum(pwp->wp_kmap) + sum(pwp->wp_umap) == 0)
                                pwp->wp_flags &= ~WP_NOWATCH;
                        else {
                                if (pwp->wp_kmap[X] + pwp->wp_umap[X])
                                        /* cannot have exec-only protection */
                                        prot |= PROT_READ|PROT_EXEC;
                                if (pwp->wp_kmap[R] + pwp->wp_umap[R])
                                        prot |= PROT_READ;
                                if (pwp->wp_kmap[W] + pwp->wp_umap[W])
                                        /* cannot have write-only protection */
                                        prot |= PROT_READ|PROT_WRITE;
#if 0   /* damned broken mmu feature! */
                                if (sum(pwp->wp_umap) == 0)
                                        prot &= ~PROT_USER;
#endif
                        }
                }


                if (pwp->wp_oprot != 0) {       /* if page exists */
                        struct seg *seg;
                        uint_t oprot;
                        int err, retrycnt = 0;

                        AS_LOCK_EXIT(as, &as->a_lock);
                        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
                retry:
                        seg = as_segat(as, addr);
                        ASSERT(seg != NULL);
                        segop_getprot(seg, addr, 0, &oprot);
                        SEGOP_GETPROT(seg, addr, 0, &oprot);
                        if (prot != oprot) {
                                err = segop_setprot(seg, addr, PAGESIZE, prot);
                                err = SEGOP_SETPROT(seg, addr, PAGESIZE, prot);
                                if (err == IE_RETRY) {
                                        ASSERT(retrycnt == 0);
                                        retrycnt++;
                                        goto retry;
                                }
                        }
                        AS_LOCK_EXIT(as, &as->a_lock);
                } else
                        AS_LOCK_EXIT(as, &as->a_lock);

                /*
                 * When all pages are mapped back to their normal state,
                 * continue the other lwps.
                 */
                if (!mapin) {
                        ASSERT(p->p_mapcnt > 0);
```



```
                        p->p_mapcnt--;
                        if (p->p_mapcnt == 0) {
                                mutex_exit(&p->p_maplock);
                                mutex_enter(&p->p_lock);
                                continuelwps(p);
                                mutex_exit(&p->p_lock);
                                mutex_enter(&p->p_maplock);
                        }
                }

                AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
        }

        AS_LOCK_EXIT(as, &as->a_lock);
        mutex_exit(&p->p_maplock);

        return (rv);
}
```

unchanged_portion_omitted_

```
/*
 * Function called by an lwp after it resumes from stop().
 */
void
setallwatch(void)
{
        proc_t *p = curproc;
        struct as *as = curproc->p_as;
        struct watched_page *pwp, *next;
        struct seg *seg;
        caddr_t vaddr;
        uint_t prot;
        int err, retrycnt;

        if (p->p_wprot == NULL)
                return;

        ASSERT(MUTEX_NOT_HELD(&curproc->p_lock));

        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

        pwp = p->p_wprot;
        while (pwp != NULL) {

                vaddr = pwp->wp_vaddr;
                retrycnt = 0;
        retry:
                ASSERT(pwp->wp_flags & WP_SETPROT);
                if ((seg = as_segat(as, vaddr)) != NULL &&
                    !(pwp->wp_flags & WP_NOWATCH)) {
                        prot = pwp->wp_prot;
                        err = segop_setprot(seg, vaddr, PAGESIZE, prot);
                        err = SEGOP_SETPROT(seg, vaddr, PAGESIZE, prot);
                        if (err == IE_RETRY) {
                                ASSERT(retrycnt == 0);
                                retrycnt++;
                                goto retry;
                        }
                }

                next = pwp->wp_list;

                if (pwp->wp_read + pwp->wp_write + pwp->wp_exec == 0) {
                        /*
                         * No watched areas remain in this page.
                         * Free the watched_page structure.
```

```
 419                          */
 420                         avl_remove(&as->a_wpage, pwp);
 421                         kmem_free(pwp, sizeof (struct watched_page));
 422                 } else {
 423                         pwp->wp_flags &= ~WP_SETPROT;
 424                 }

 426                 pwp = next;
 427         }
 428         p->p_wprot = NULL;

 430         AS_LOCK_EXIT(as, &as->a_lock);
 431 }
______unchanged_portion_omitted_


 490 /*
 491  * trap() calls here to determine if a fault is in a watched page.
 492  * We return nonzero if this is true and the load/store would fail.
 493  */
 494 int
 495 pr_is_watchpage(caddr_t addr, enum seg_rw rw)
 496 {
 497         struct as *as = curproc->p_as;
 498         int rv;

 500         if ((as == &kas) || avl_numnodes(&as->a_wpage) == 0)
 501                 return (0);

 507         /* Grab the lock because of XHAT (see comment in pr_mappage()) */
 503         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
 504         rv = pr_is_watchpage_as(addr, rw, as);
 505         AS_LOCK_EXIT(as, &as->a_lock);

 507         return (rv);
 508 }
______unchanged_portion_omitted_
```

****
193225 Fri May 8 18:10:31 2015
new/usr/src/uts/common/os/zone.c
patch lower-case-segops
****
_____unchanged_portion_omitted_

```
5593 /*
5594  * Return zero if the process has at least one vnode mapped in to its
5595  * address space which shouldn't be allowed to change zones.
5596  *
5597  * Also return zero if the process has any shared mappings which reserve
5598  * swap.  This is because the counting for zone.max-swap does not allow swap
5599  * reservation to be shared between zones.  zone swap reservation is counted
5600  * on zone->zone_max_swap.
5601  */
5602 static int
5603 as_can_change_zones(void)
5604 {
5605         proc_t *pp = curproc;
5606         struct seg *seg;
5607         struct as *as = pp->p_as;
5608         vnode_t *vp;
5609         int allow = 1;

5611         ASSERT(pp->p_as != &kas);
5612         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
5613         for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {

5615                 /*
5616                  * Cannot enter zone with shared anon memory which
5617                  * reserves swap.  See comment above.
5618                  */
5619                 if (seg_can_change_zones(seg) == B_FALSE) {
5620                         allow = 0;
5621                         break;
5622                 }
5623                 /*
5624                  * if we can't get a backing vnode for this segment then skip
5625                  * it.
5626                  */
5627                 vp = NULL;
5628                 if (segop_getvp(seg, seg->s_base, &vp) != 0 || vp == NULL)
5628                 if (SEGOP_GETVP(seg, seg->s_base, &vp) != 0 || vp == NULL)
5629                         continue;
5630                 if (!vn_can_change_zones(vp)) { /* bail on first match */
5631                         allow = 0;
5632                         break;
5633                 }
5634         }
5635         AS_LOCK_EXIT(as, &as->a_lock);
5636         return (allow);
5637 }
```

_____unchanged_portion_omitted_

```
**********************************************************
    7397 Fri May  8 18:10:32 2015
new/usr/src/uts/common/sys/class.h
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

  72 typedef struct thread_ops {
  73         int     (*cl_enterclass)(kthread_t *, id_t, void *, cred_t *, void *);
  74         void    (*cl_exitclass)(void *);
  75         int     (*cl_canexit)(kthread_t *, cred_t *);
  76         int     (*cl_fork)(kthread_t *, kthread_t *, void *);
  77         void    (*cl_forkret)(kthread_t *, kthread_t *);
  78         void    (*cl_parmsget)(kthread_t *, void *);
  79         int     (*cl_parmsset)(kthread_t *, void *, id_t, cred_t *);
  80         void    (*cl_stop)(kthread_t *, int, int);
  81         void    (*cl_exit)(kthread_t *);
  82         void    (*cl_active)(kthread_t *);
  83         void    (*cl_inactive)(kthread_t *);
  84         pri_t   (*cl_swapin)(kthread_t *, int);
  85         pri_t   (*cl_swapout)(kthread_t *, int);
  84         void    (*cl_trapret)(kthread_t *);
  85         void    (*cl_preempt)(kthread_t *);
  86         void    (*cl_setrun)(kthread_t *);
  87         void    (*cl_sleep)(kthread_t *);
  88         void    (*cl_tick)(kthread_t *);
  89         void    (*cl_wakeup)(kthread_t *);
  90         int     (*cl_donice)(kthread_t *, cred_t *, int, int *);
  91         pri_t   (*cl_globpri)(kthread_t *);
  92         void    (*cl_set_process_group)(pid_t, pid_t, pid_t);
  93         void    (*cl_yield)(kthread_t *);
  94         int     (*cl_doprio)(kthread_t *, cred_t *, int, int *);
  95 } thread_ops_t;
______unchanged_portion_omitted_

 111 #define STATIC_SCHED            (krwlock_t *)0xffffffff
 112 #define LOADABLE_SCHED(s)       ((s)->cl_lock != STATIC_SCHED)
 113 #define SCHED_INSTALLED(s)      ((s)->cl_funcs != NULL)
 114 #define ALLOCATED_SCHED(s)      ((s)->cl_lock != NULL)

 116 #ifdef  _KERNEL

 118 #define CLASS_KERNEL(cid)       ((cid) == syscid || (cid) == sysdccid)

 120 extern int      nclass;         /* number of configured scheduling classes */
 121 extern char     *defaultclass;  /* default class for newproc'd processes */
 122 extern struct sclass sclass[];  /* the class table */
 123 extern kmutex_t class_lock;     /* lock protecting class table */
 124 extern int      loaded_classes; /* number of classes loaded */

 126 extern pri_t    minclsyspri;
 127 extern id_t     syscid;         /* system scheduling class ID */
 128 extern id_t     sysdccid;       /* system duty-cycle scheduling class ID */
 129 extern id_t     defaultcid;     /* "default" class id; see dispadmin(1M) */

 131 extern int      alloc_cid(char *, id_t *);
 132 extern int      scheduler_load(char *, sclass_t *);
 133 extern int      getcid(char *, id_t *);
 134 extern int      getcidbyname(char *, id_t *);
 135 extern int      parmsin(pcparms_t *, pc_vaparms_t *);
```



```
 136 extern int      parmsout(pcparms_t *, pc_vaparms_t *);
 137 extern int      parmsset(pcparms_t *, kthread_t *);
 138 extern void     parmsget(kthread_t *, pcparms_t *);
 139 extern int      vaparmsout(char *, pcparms_t *, pc_vaparms_t *, uio_seg_t);

 141 #endif

 143 #define CL_ADMIN(clp, uaddr, reqpcredp) \
 144         (*(clp)->cl_funcs->sclass.cl_admin)(uaddr, reqpcredp)

 146 #define CL_ENTERCLASS(t, cid, clparmsp, credp, bufp) \
 147         (sclass[cid].cl_funcs->thread.cl_enterclass) (t, cid, \
 148             (void *)clparmsp, credp, bufp)

 150 #define CL_EXITCLASS(cid, clprocp)\
 151         (sclass[cid].cl_funcs->thread.cl_exitclass) ((void *)clprocp)

 153 #define CL_CANEXIT(t, cr)       (*(t)->t_clfuncs->cl_canexit)(t, cr)

 155 #define CL_FORK(tp, ct, bufp)   (*(tp)->t_clfuncs->cl_fork)(tp, ct, bufp)

 157 #define CL_FORKRET(t, ct)       (*(t)->t_clfuncs->cl_forkret)(t, ct)

 159 #define CL_GETCLINFO(clp, clinfop) \
 160         (*(clp)->cl_funcs->sclass.cl_getclinfo)((void *)clinfop)

 162 #define CL_GETCLPRI(clp, clprip) \
 163         (*(clp)->cl_funcs->sclass.cl_getclpri)(clprip)

 165 #define CL_PARMSGET(t, clparmsp) \
 166         (*(t)->t_clfuncs->cl_parmsget)(t, (void *)clparmsp)

 168 #define CL_PARMSIN(clp, clparmsp) \
 169         (clp)->cl_funcs->sclass.cl_parmsin((void *)clparmsp)

 171 #define CL_PARMSOUT(clp, clparmsp, vaparmsp) \
 172         (clp)->cl_funcs->sclass.cl_parmsout((void *)clparmsp, vaparmsp)

 174 #define CL_VAPARMSIN(clp, clparmsp, vaparmsp) \
 175         (clp)->cl_funcs->sclass.cl_vaparmsin((void *)clparmsp, vaparmsp)

 177 #define CL_VAPARMSOUT(clp, clparmsp, vaparmsp) \
 178         (clp)->cl_funcs->sclass.cl_vaparmsout((void *)clparmsp, vaparmsp)

 180 #define CL_PARMSSET(t, clparmsp, cid, curpcredp) \
 181         (*(t)->t_clfuncs->cl_parmsset)(t, (void *)clparmsp, cid, curpcredp)

 183 #define CL_PREEMPT(tp)          (*(tp)->t_clfuncs->cl_preempt)(tp)

 185 #define CL_SETRUN(tp)           (*(tp)->t_clfuncs->cl_setrun)(tp)

 187 #define CL_SLEEP(tp)            (*(tp)->t_clfuncs->cl_sleep)(tp)

 189 #define CL_STOP(t, why, what)   (*(t)->t_clfuncs->cl_stop)(t, why, what)

 191 #define CL_EXIT(t)              (*(t)->t_clfuncs->cl_exit)(t)

 193 #define CL_ACTIVE(t)            (*(t)->t_clfuncs->cl_active)(t)

 195 #define CL_INACTIVE(t)          (*(t)->t_clfuncs->cl_inactive)(t)

 199 #define CL_SWAPIN(t, flags)     (*(t)->t_clfuncs->cl_swapin)(t, flags)

 201 #define CL_SWAPOUT(t, flags)    (*(t)->t_clfuncs->cl_swapout)(t, flags)

 197 #define CL_TICK(t)              (*(t)->t_clfuncs->cl_tick)(t)
```

```
199 #define CL_TRAPRET(t)           (*(t)->t_clfuncs->cl_trapret)(t)

201 #define CL_WAKEUP(t)            (*(t)->t_clfuncs->cl_wakeup)(t)

203 #define CL_DONICE(t, cr, inc, ret) \
204         (*(t)->t_clfuncs->cl_donice)(t, cr, inc, ret)

206 #define CL_DOPRIO(t, cr, inc, ret) \
207         (*(t)->t_clfuncs->cl_doprio)(t, cr, inc, ret)

209 #define CL_GLOBPRI(t)           (*(t)->t_clfuncs->cl_globpri)(t)

211 #define CL_SET_PROCESS_GROUP(t, s, b, f) \
212         (*(t)->t_clfuncs->cl_set_process_group)(s, b, f)

214 #define CL_YIELD(tp)            (*(tp)->t_clfuncs->cl_yield)(tp)

216 #define CL_ALLOC(pp, cid, flag) \
217         (sclass[cid].cl_funcs->sclass.cl_alloc) (pp, flag)

219 #define CL_FREE(cid, bufp)      (sclass[cid].cl_funcs->sclass.cl_free) (bufp)

221 #ifdef  __cplusplus
222 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    5723 Fri May  8 18:10:32 2015
new/usr/src/uts/common/sys/disp.h
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
```

```
   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 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
  27 /*        All Rights Reserved   */


  30 #ifndef _SYS_DISP_H
  31 #define _SYS_DISP_H

  33 #pragma ident   "%Z%%M% %I%     %E% SMI"       /* SVr4.0 1.11  */

  33 #include <sys/priocntl.h>
  34 #include <sys/thread.h>
  35 #include <sys/class.h>

  37 #ifdef  __cplusplus
  38 extern "C" {
  39 #endif

  41 /*
  42  * The following is the format of a dispatcher queue entry.
  43  */
  44 typedef struct dispq {
  45         kthread_t       *dq_first;      /* first thread on queue or NULL */
  46         kthread_t       *dq_last;       /* last thread on queue or NULL */
  47         int             dq_sruncnt;     /* number of loaded, runnable */
  48                                         /*    threads on queue */
  49 } dispq_t;
______unchanged_portion_omitted_

  80 #if defined(_KERNEL)
```



```
  82 #define MAXCLSYSPRI     99
  83 #define MINCLSYSPRI     60


  86 /*
  87  * Global scheduling variables.
  88  *      - See sys/cpuvar.h for CPU-local variables.
  89  */
  90 extern int      nswapped;       /* number of swapped threads */
  91                                 /* nswapped protected by swap_lock */

  93 extern  pri_t   minclsyspri;    /* minimum level of any system class */
  94 extern  pri_t   maxclsyspri;    /* maximum level of any system class */
  95 extern  pri_t   intr_pri;       /* interrupt thread priority base level */

  97 /*
  98  * Minimum amount of time that a thread can remain runnable before it can
  99  * be stolen by another CPU (in nanoseconds).
 100  */
 101 extern hrtime_t nosteal_nsec;

 103 /*
 104  * Kernel preemption occurs if a higher-priority thread is runnable with
 105  * a priority at or above kpreemptpri.
 106  *
 107  * So that other processors can watch for such threads, a separate
 108  * dispatch queue with unbound work above kpreemptpri is maintained.
 109  * This is part of the CPU partition structure (cpupart_t).
 110  */
 111 extern  pri_t   kpreemptpri;    /* level above which preemption takes place */

 113 extern void             disp_kp_alloc(disp_t *, pri_t); /* allocate kp queue */
 114 extern void             disp_kp_free(disp_t *);         /* free kp queue */

 116 /*
 117  * Macro for use by scheduling classes to decide whether the thread is about
 118  * to be scheduled or not.  This returns the maximum run priority.
 119  */
 120 #define DISP_MAXRUNPRI(t)       ((t)->t_disp_queue->disp_maxrunpri)

 122 /*
 123  * Platform callbacks for various dispatcher operations
 124  *
 125  * idle_cpu() is invoked when a cpu goes idle, and has nothing to do.
 126  * disp_enq_thread() is invoked when a thread is placed on a run queue.
 127  */
 128 extern void     (*idle_cpu)();
 129 extern void     (*disp_enq_thread)(struct cpu *, int);


 132 extern int              dispdeq(kthread_t *);
 133 extern void             dispinit(void);
 134 extern void             disp_add(sclass_t *);
 135 extern int              intr_active(struct cpu *, int);
 136 extern int              servicing_interrupt(void);
 137 extern void             preempt(void);
 138 extern void             setbackdq(kthread_t *);
 139 extern void             setfrontdq(kthread_t *);
 140 extern void             swtch(void);
 141 extern void             swtch_to(kthread_t *);
 142 extern void             swtch_from_zombie(void)
 143                                 __NORETURN;
 146 extern void             dq_sruninc(kthread_t *);
 147 extern void             dq_srundec(kthread_t *);
 144 extern void             cpu_rechoose(kthread_t *);
 145 extern void             cpu_surrender(kthread_t *);
```

```
146 extern void             kpreempt(int);
147 extern struct cpu       *disp_lowpri_cpu(struct cpu *, struct lgrp_ld *, pri_t,
148                             struct cpu *);
149 extern int              disp_bound_threads(struct cpu *, int);
150 extern int              disp_bound_anythreads(struct cpu *, int);
151 extern int              disp_bound_partition(struct cpu *, int);
152 extern void             disp_cpu_init(struct cpu *);
153 extern void             disp_cpu_fini(struct cpu *);
154 extern void             disp_cpu_inactive(struct cpu *);
155 extern void             disp_adjust_unbound_pri(kthread_t *);
156 extern void             resume(kthread_t *);
157 extern void             resume_from_intr(kthread_t *);
158 extern void             resume_from_zombie(kthread_t *)
159                                 __NORETURN;
160 extern void             disp_swapped_enq(kthread_t *);
161 extern int              disp_anywork(void);

163 #define KPREEMPT_SYNC           (-1)
164 #define kpreempt_disable()                              \
165         {                                               \
166                 curthread->t_preempt++;                 \
167                 ASSERT(curthread->t_preempt >= 1);      \
168         }
169 #define kpreempt_enable()                               \
170         {                                               \
171                 ASSERT(curthread->t_preempt >= 1);      \
172                 if (--curthread->t_preempt == 0 &&      \
173                     CPU->cpu_kprunrun)                  \
174                         kpreempt(KPREEMPT_SYNC);        \
175         }

177 #endif  /* _KERNEL */

179 #ifdef  __cplusplus
180 }
```

unchanged_portion_omitted_

```
**********************************************************
   29295 Fri May  8 18:10:32 2015
new/usr/src/uts/common/sys/proc.h
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

 124 struct pool;
 125 struct task;
 126 struct zone;
 127 struct brand;
 128 struct corectl_path;
 129 struct corectl_content;

 131 /*
 132  * One structure allocated per active process.  Per-process data (user.h) is
 133  * also inside the proc structure.
 132  * One structure allocated per active process.  It contains all
 133  * data needed about the process while the process may be swapped
 134  * out.  Other per-process data (user.h) is also inside the proc structure.
 135  * Lightweight-process data (lwp.h) and the kernel stack may be swapped out.
 134  */
 135 typedef struct  proc {
 136         /*
 137          * Fields requiring no explicit locking
 138          */
 139         struct  vnode *p_exec;          /* pointer to a.out vnode */
 140         struct  as *p_as;               /* process address space pointer */
 141         struct  plock *p_lockp;         /* ptr to proc struct's mutex lock */
 142         kmutex_t p_crlock;              /* lock for p_cred */
 143         struct  cred    *p_cred;        /* process credentials */
 144         /*
 145          * Fields protected by pidlock
 146          */
 149         int     p_swapcnt;              /* number of swapped out lwps */
 147         char    p_stat;                 /* status of process */
 148         char    p_wcode;                /* current wait code */
 149         ushort_t p_pidflag;             /* flags protected only by pidlock */
 150         int     p_wdata;                /* current wait return value */
 151         pid_t   p_ppid;                 /* process id of parent */
 152         struct  proc    *p_link;        /* forward link */
 153         struct  proc    *p_parent;      /* ptr to parent process */
 154         struct  proc    *p_child;       /* ptr to first child process */
 155         struct  proc    *p_sibling;     /* ptr to next sibling proc on chain */
 156         struct  proc    *p_psibling;    /* ptr to prev sibling proc on chain */
 157         struct  proc    *p_sibling_ns;  /* prt to siblings with new state */
 158         struct  proc    *p_child_ns;    /* prt to children with new state */
 159         struct  proc    *p_next;        /* active chain link next */
 160         struct  proc    *p_prev;        /* active chain link prev */
 161         struct  proc    *p_nextofkin;   /* gets accounting info at exit */
 162         struct  proc    *p_orphan;
 163         struct  proc    *p_nextorph;
 164         struct  proc    *p_pglink;      /* process group hash chain link next */
 165         struct  proc    *p_ppglink;     /* process group hash chain link prev */
 166         struct  sess    *p_sessp;       /* session information */
 167         struct  pid     *p_pidp;        /* process ID info */
 168         struct  pid     *p_pgidp;       /* process group ID info */
 169         /*
 170          * Fields protected by p_lock
 171          */
```



```
 172         kcondvar_t p_cv;                /* proc struct's condition variable */
 173         kcondvar_t p_flag_cv;
 174         kcondvar_t p_lwpexit;           /* waiting for some lwp to exit */
 175         kcondvar_t p_holdlwps;          /* process is waiting for its lwps */
 176                                         /* to to be held.  */
 177         uint_t  p_proc_flag;            /* /proc-related flags */
 178         uint_t  p_flag;                 /* protected while set. */
 179                                         /* flags defined below */
 180         clock_t p_utime;                /* user time, this process */
 181         clock_t p_stime;                /* system time, this process */
 182         clock_t p_cutime;               /* sum of children's user time */
 183         clock_t p_cstime;               /* sum of children's system time */
 184         avl_tree_t *p_segacct;          /* System V shared segment list */
 185         avl_tree_t *p_semacct;          /* System V semaphore undo list */
 186         caddr_t p_bssbase;              /* base addr of last bss below heap */
 187         caddr_t p_brkbase;              /* base addr of heap */
 188         size_t  p_brksize;              /* heap size in bytes */
 189         uint_t  p_brkpageszc;           /* preferred heap max page size code */
 190         /*
 191          * Per process signal stuff.
 192          */
 193         k_sigset_t p_sig;               /* signals pending to this process */
 194         k_sigset_t p_extsig;            /* signals sent from another contract */
 195         k_sigset_t p_ignore;            /* ignore when generated */
 196         k_sigset_t p_siginf0;           /* gets signal info with signal */
 197         struct sigqueue *p_sigqueue;    /* queued siginfo structures */
 198         struct sigqhdr *p_sigqhdr;      /* hdr to sigqueue structure pool */
 199         struct sigqhdr *p_signhdr;      /* hdr to signotify structure pool */
 200         uchar_t p_stopsig;              /* jobcontrol stop signal */

 202         /*
 203          * Special per-process flag when set will fix misaligned memory
 204          * references.
 205          */
 206         char    p_fixalignment;

 208         /*
 209          * Per process lwp and kernel thread stuff
 210          */
 211         id_t    p_lwpid;                /* most recently allocated lwpid */
 212         int     p_lwpcnt;               /* number of lwps in this process */
 213         int     p_lwprcnt;              /* number of not stopped lwps */
 214         int     p_lwpdaemon;            /* number of TP_DAEMON lwps */
 215         int     p_lwpwait;              /* number of lwps in lwp_wait() */
 216         int     p_lwpdwait;             /* number of daemons in lwp_wait() */
 217         int     p_zombcnt;              /* number of zombie lwps */
 218         kthread_t *p_tlist;             /* circular list of threads */
 219         lwpdir_t *p_lwpdir;             /* thread (lwp) directory */
 220         lwpdir_t *p_lwpfree;            /* p_lwpdir free list */
 221         tidhash_t *p_tidhash;           /* tid (lwpid) lookup hash table */
 222         uint_t  p_lwpdir_sz;            /* number of p_lwpdir[] entries */
 223         uint_t  p_tidhash_sz;           /* number of p_tidhash[] entries */
 224         ret_tidhash_t *p_ret_tidhash;   /* retired tidhash hash tables */
 225         uint64_t p_lgrpset;             /* unprotected hint of set of lgrps */
 226                                         /* on which process has threads */
 227         volatile lgrp_id_t  p_t1_lgrpid; /* main's thread lgroup id */
 228         volatile lgrp_id_t  p_tr_lgrpid; /* text replica's lgroup id */
 229 #if defined(_LP64)
 230         uintptr_t  p_lgrpres2;          /* reserved for lgrp migration */
 231 #endif
 232         /*
 233          * /proc (process filesystem) debugger interface stuff.
 234          */
 235         k_sigset_t p_sigmask;           /* mask of traced signals (/proc) */
 236         k_fltset_t p_fltmask;           /* mask of traced faults (/proc) */
 237         struct vnode *p_trace;          /* pointer to primary /proc vnode */
```

```
238         struct vnode *p_plist;          /* list of /proc vnodes for process */
239         kthread_t *p_agenttp;           /* thread ptr for /proc agent lwp */
240         avl_tree_t p_warea;             /* list of watched areas */
241         avl_tree_t p_wpage;             /* remembered watched pages (vfork) */
242         watched_page_t *p_wprot;        /* pages that need to have prot set */
243         int     p_mapcnt;               /* number of active pr_mappage()s */
244         kmutex_t p_maplock;             /* lock for pr_mappage() */
245         struct  proc  *p_rlink;         /* linked list for server */
246         kcondvar_t p_srwchan_cv;
247         size_t  p_stksize;              /* process stack size in bytes */
248         uint_t  p_stkpageszc;           /* preferred stack max page size code */

250         /*
251          * Microstate accounting, resource usage, and real-time profiling
252          */
253         hrtime_t p_mstart;              /* hi-res process start time */
254         hrtime_t p_mterm;               /* hi-res process termination time */
255         hrtime_t p_mlreal;              /* elapsed time sum over defunct lwps */
256         hrtime_t p_acct[NMSTATES];      /* microstate sum over defunct lwps */
257         hrtime_t p_cacct[NMSTATES];     /* microstate sum over child procs */
258         struct lrusage p_ru;            /* lrusage sum over defunct lwps */
259         struct lrusage p_cru;           /* lrusage sum over child procs */
260         struct itimerval p_rprof_timer; /* ITIMER_REALPROF interval timer */
261         uintptr_t p_rprof_cyclic;       /* ITIMER_REALPROF cyclic */
262         uint_t  p_defunct;              /* number of defunct lwps */
263         /*
264          * profiling. A lock is used in the event of multiple lwp's
265          * using the same profiling base/size.
266          */
267         kmutex_t p_pflock;              /* protects user profile arguments */
268         struct prof p_prof;             /* profile arguments */

270         /*
271          * Doors.
272          */
273         door_pool_t             p_server_threads; /* common thread pool */
274         struct door_node        *p_door_list;   /* active doors */
275         struct door_node        *p_unref_list;
276         kcondvar_t              p_unref_cv;
277         char                    p_unref_thread; /* unref thread created */

279         /*
280          * Kernel probes
281          */
282         uchar_t                 p_tnf_flags;

284         /*
285          * Solaris Audit
286          */
287         struct p_audit_data     *p_audit_data; /* per process audit structure */

289         pctxop_t        *p_pctx;

291 #if defined(__x86)
292         /*
293          * LDT support.
294          */
295         kmutex_t        p_ldtlock;      /* protects the following fields */
296         user_desc_t     *p_ldt;         /* Pointer to private LDT */
297         system_desc_t   p_ldt_desc;     /* segment descriptor for private LDT */
298         ushort_t        p_ldtlimit;     /* highest selector used */
299 #endif
300         size_t p_swrss;                 /* resident set size before last swap */
301         struct aio      *p_aio;         /* pointer to async I/O struct */
302         struct itimer   **p_itimer;     /* interval timers */
303         timeout_id_t    p_alarmid;      /* alarm's timeout id */
```



```
304         caddr_t         p_usrstack;     /* top of the process stack */
305         uint_t          p_stkprot;      /* stack memory protection */
306         uint_t          p_datprot;      /* data memory protection */
307         model_t         p_model;        /* data model determined at exec time */
308         struct lwpchan_data *p_lcp;     /* lwpchan cache */
309         kmutex_t        p_lcp_lock;     /* protects assignments to p_lcp */
310         utrap_handler_t *p_utraps;      /* pointer to user trap handlers */
311         struct corectl_path     *p_corefile;    /* pattern for core file */
312         struct task     *p_task;        /* our containing task */
313         struct proc     *p_taskprev;    /* ptr to previous process in task */
314         struct proc     *p_tasknext;    /* ptr to next process in task */
315         kmutex_t        p_sc_lock;      /* protects p_pagep */
316         struct sc_page_ctl *p_pagep;    /* list of process's shared pages */
317         struct rctl_set *p_rctls;       /* resource controls for this process */
318         rlim64_t        p_stk_ctl;      /* currently enforced stack size */
319         rlim64_t        p_fsz_ctl;      /* currently enforced file size */
320         rlim64_t        p_vmem_ctl;     /* currently enforced addr-space size */
321         rlim64_t        p_fno_ctl;      /* currently enforced file-desc limit */
322         pid_t           p_ancpid;       /* ancestor pid, used by exacct */
323         struct itimerval p_realitimer;  /* real interval timer */
324         timeout_id_t    p_itimerid;     /* real interval timer's timeout id */
325         struct corectl_content *p_content;      /* content of core file */

327         avl_tree_t      p_ct_held;      /* held contracts */
328         struct ct_equeue **p_ct_equeue; /* process-type event queues */

330         struct cont_process *p_ct_process; /* process contract */
331         list_node_t     p_ct_member;    /* process contract membership */
332         sigqueue_t      *p_killsqp;     /* sigqueue pointer for SIGKILL */

334         int             p_dtrace_probes; /* are there probes for this proc? */
335         uint64_t        p_dtrace_count; /* number of DTrace tracepoints */
336                                         /* (protected by P_PR_LOCK) */
337         void            *p_dtrace_helpers; /* DTrace helpers, if any */
338         struct pool     *p_pool;        /* pointer to containing pool */
339         kcondvar_t      p_poolcv;       /* synchronization with pools */
340         uint_t          p_poolcnt;      /* # threads inside pool barrier */
341         uint_t          p_poolflag;     /* pool-related flags (see below) */
342         uintptr_t       p_portcnt;      /* event ports counter */
343         struct zone     *p_zone;        /* zone in which process lives */
344         struct vnode    *p_execdir;     /* directory that p_exec came from */
345         struct brand    *p_brand;       /* process's brand  */
346         void            *p_brand_data;  /* per-process brand state */

348         /* additional lock to protect p_sessp (but not its contents) */
349         kmutex_t p_splock;
350         rctl_qty_t      p_locked_mem;   /* locked memory charged to proc */
351                                         /* protected by p_lock */
352         rctl_qty_t      p_crypto_mem;   /* /dev/crypto memory charged to proc */
353                                         /* protected by p_lock */
354         clock_t p_ttime;                /* buffered task time */

356         /*
357          * The user structure
358          */
359         struct user p_user;             /* (see sys/user.h) */
360 } proc_t;
```

_____***unchanged_portion_omitted_***

****
11325 Fri May 8 18:10:32 2015
new/usr/src/uts/common/sys/sysinfo.h
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim memory. The code is there and in theory it runs when we get *extremely* low on memory. In practice, it never runs since the definition of low-on-memory is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
****

_____***unchanged_portion_omitted_***

```
 248 typedef struct cpu_vm_stats {
 249         uint64_t pgrec;                 /* page reclaims (includes pageout) */
 250         uint64_t pgfrec;                /* page reclaims from free list */
 251         uint64_t pgin;                  /* pageins */
 252         uint64_t pgpgin;                /* pages paged in */
 253         uint64_t pgout;                 /* pageouts */
 254         uint64_t pgpgout;               /* pages paged out */
 255         uint64_t swapin;                /* swapins */
 256         uint64_t pgswapin;              /* pages swapped in */
 257         uint64_t swapout;               /* swapouts */
 258         uint64_t pgswapout;             /* pages swapped out */
 255         uint64_t zfod;                  /* pages zero filled on demand */
 256         uint64_t dfree;                 /* pages freed by daemon or auto */
 257         uint64_t scan;                  /* pages examined by pageout daemon */
 258         uint64_t rev;                   /* revolutions of page daemon hand */
 259         uint64_t hat_fault;             /* minor page faults via hat_fault() */
 260         uint64_t as_fault;              /* minor page faults via as_fault() */
 261         uint64_t maj_fault;             /* major page faults */
 262         uint64_t cow_fault;             /* copy-on-write faults */
 263         uint64_t prot_fault;            /* protection faults */
 264         uint64_t softlock;              /* faults due to software locking req */
 265         uint64_t kernel_asflt;          /* as_fault()s in kernel addr space */
 266         uint64_t pgrrun;                /* times pager scheduled */
 267         uint64_t execpgin;              /* executable pages paged in */
 268         uint64_t execpgout;             /* executable pages paged out */
 269         uint64_t execfree;              /* executable pages freed */
 270         uint64_t anonpgin;              /* anon pages paged in */
 271         uint64_t anonpgout;             /* anon pages paged out */
 272         uint64_t anonfree;              /* anon pages freed */
 273         uint64_t fspgin;                /* fs pages paged in */
 274         uint64_t fspgout;               /* fs pages paged out */
 275         uint64_t fsfree;                /* fs pages free */
 276 } cpu_vm_stats_t;
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   15085 Fri May  8 18:10:33 2015
new/usr/src/uts/common/sys/systm.h
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
```

```
   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 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
  22 /*        All Rights Reserved   */


  25 /*
  26  * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
  27  * Use is subject to license terms.
  28  */

  30 #ifndef _SYS_SYSTM_H
  31 #define _SYS_SYSTM_H

  33 #include <sys/types.h>
  34 #include <sys/t_lock.h>
  35 #include <sys/proc.h>
  36 #include <sys/dditypes.h>

  38 #ifdef  __cplusplus
  39 extern "C" {
  40 #endif

  42 /*
  43  * The pc_t is the type of the kernel's program counter.  In general, a
  44  * pc_t is a uintptr_t -- except for a sparcv9 kernel, in which case all
  45  * instruction text is below 4G, and a pc_t is thus a uint32_t.
  46  */
  47 #ifdef __sparcv9
  48 typedef uint32_t pc_t;
  49 #else
  50 typedef uintptr_t pc_t;
  51 #endif

  53 /*
  54  * Random set of variables used by more than one routine.
  55  */
```



```
  57 #ifdef _KERNEL
  58 #include <sys/varargs.h>
  59 #include <sys/uadmin.h>

  61 extern int hz;                  /* system clock rate */
  62 extern struct vnode *rootdir;   /* pointer to vnode of root directory */
  63 extern struct vnode *devicesdir;        /* pointer to /devices vnode */
  64 extern int interrupts_unleashed;        /* set after the spl0() in main() */

  66 extern char runin;              /* scheduling flag */
  67 extern char runout;             /* scheduling flag */
  68 extern char wake_sched;         /* causes clock to wake swapper on next tick */
  69 extern char wake_sched_sec;     /* causes clock to wake swapper after a sec */

  66 extern pgcnt_t  maxmem;         /* max available memory (pages) */
  67 extern pgcnt_t  physmem;        /* physical memory (pages) on this CPU */
  68 extern pfn_t    physmax;        /* highest numbered physical page present */
  69 extern pgcnt_t  physinstalled;  /* physical pages including PROM/boot use */

  71 extern caddr_t  s_text;         /* start of kernel text segment */
  72 extern caddr_t  e_text;         /* end of kernel text segment */
  73 extern caddr_t  s_data;         /* start of kernel text segment */
  74 extern caddr_t  e_data;         /* end of kernel text segment */

  76 extern pgcnt_t  availrmem;      /* Available resident (not swapable)    */
  77                                 /* memory in pages.                     */
  78 extern pgcnt_t  availrmem_initial;      /* initial value of availrmem   */
  79 extern pgcnt_t  segspt_minfree; /* low water mark for availrmem in seg_spt */
  80 extern pgcnt_t  freemem;        /* Current free memory.                 */

  82 extern dev_t    rootdev;        /* device of the root */
  83 extern struct vnode *rootvp;    /* vnode of root device */
  84 extern boolean_t root_is_svm;           /* root is a mirrored device flag */
  85 extern boolean_t root_is_ramdisk;       /* root is boot_archive ramdisk */
  86 extern uint32_t  ramdisk_size;          /* (KB) set only for sparc netboots */
  87 extern char *volatile panicstr; /* panic string pointer */
  88 extern va_list  panicargs;      /* panic arguments */
  89 extern volatile int quiesce_active;     /* quiesce(9E) is in progress */

  91 extern int      rstchown;       /* 1 ==> restrictive chown(2) semantics */
  92 extern int      klustsize;

  94 extern int      abort_enable;   /* Platform input-device abort policy */

  96 extern int      audit_active;   /* Solaris Auditing module state */

  98 extern int      avenrun[];      /* array of load averages */

 100 extern char *isa_list;          /* For sysinfo's isalist option */

 102 extern int noexec_user_stack;           /* patchable via /etc/system */
 103 extern int noexec_user_stack_log;       /* patchable via /etc/system */

 105 /*
 106  * Use NFS client operations in the global zone only.  Under contract with
 107  * admin/install; do not change without coordinating with that consolidation.
 108  */
 109 extern int nfs_global_client_only;

 111 extern void report_stack_exec(proc_t *, caddr_t);

 113 extern void startup(void);
 114 extern void clkstart(void);
 115 extern void post_startup(void);
 116 extern void kern_setup1(void);
```

```
 117 extern void ka_init(void);
 118 extern void nodename_set(void);

 120 /*
 121  * for tod fault detection
 122  */
 123 enum tod_fault_type {
 124         TOD_REVERSED = 0,
 125         TOD_STALLED,
 126         TOD_JUMPED,
 127         TOD_RATECHANGED,
 128         TOD_RDONLY,
 129         TOD_NOFAULT
 130 };
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   26128 Fri May  8 18:10:33 2015
new/usr/src/uts/common/sys/thread.h
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

  96 typedef struct _kthread *kthread_id_t;

  98 struct turnstile;
  99 struct panic_trap_info;
 100 struct upimutex;
 101 struct kproject;
 102 struct on_trap_data;
 103 struct waitq;
 104 struct _kcpc_ctx;
 105 struct _kcpc_set;

 107 /* Definition for kernel thread identifier type */
 108 typedef uint64_t kt_did_t;

 110 typedef struct _kthread {
 111         struct _kthread *t_link; /* dispq, sleepq, and free queue link */

 113         caddr_t t_stk;          /* base of stack (kernel sp value to use) */
 114         void    (*t_startpc)(void);     /* PC where thread started */
 115         struct cpu *t_bound_cpu; /* cpu bound to, or NULL if not bound */
 116         short   t_affinitycnt;  /* nesting level of kernel affinity-setting */
 117         short   t_bind_cpu;     /* user-specified CPU binding (-1 if none) */
 118         ushort_t t_flag;                /* modified only by current thread */
 119         ushort_t t_proc_flag;   /* modified holding ttproc(t)->p_lock */
 120         ushort_t t_schedflag;   /* modified holding thread_lock(t) */
 121         volatile char t_preempt;        /* don't preempt thread if set */
 122         volatile char t_preempt_lk;
 123         uint_t  t_state;        /* thread state (protected by thread_lock) */
 124         pri_t   t_pri;          /* assigned thread priority */
 125         pri_t   t_epri;         /* inherited thread priority */
 126         pri_t   t_cpri;         /* thread scheduling class priority */
 127         char    t_writer;       /* sleeping in lwp_rwlock_lock(RW_WRITE_LOCK) */
 128         uchar_t t_bindflag;     /* CPU and pset binding type */
 129         label_t t_pcb;          /* pcb, save area when switching */
 130         lwpchan_t t_lwpchan;    /* reason for blocking */
 131 #define t_wchan0        t_lwpchan.lc_wchan0
 132 #define t_wchan         t_lwpchan.lc_wchan
 133         struct _sobj_ops *t_sobj_ops;
 134         id_t    t_cid;          /* scheduling class id */
 135         struct thread_ops *t_clfuncs;   /* scheduling class ops vector */
 136         void    *t_cldata;      /* per scheduling class specific data */
 137         ctxop_t *t_ctx;         /* thread context */
 138         uintptr_t t_lofault;    /* ret pc for failed page faults */
 139         label_t *t_onfault;     /* on_fault() setjmp buf */
 140         struct on_trap_data *t_ontrap;  /* on_trap() protection data */
 141         caddr_t t_swap;         /* the bottom of the stack, if from segkp */
 142         lock_t  t_lock;         /* used to resume() a thread */
 143         uint8_t t_lockstat;     /* set while thread is in lockstat code */
 144         uint8_t t_pil;          /* interrupt thread PIL */
 145         disp_lock_t     t_pi_lock;      /* lock protecting t_prioinv list */
 146         char    t_nomigrate;    /* do not migrate if set */
 147         struct cpu      *t_cpu; /* CPU that thread last ran on */
 148         struct cpu      *t_weakbound_cpu;       /* cpu weakly bound to */
```



```
 149         struct lgrp_ld  *t_lpl; /* load average for home lgroup */
 150         void            *t_lgrp_reserv[2];      /* reserved for future */
 151         struct _kthread *t_intr; /* interrupted (pinned) thread */
 152         uint64_t        t_intr_start;   /* timestamp when time slice began */
 153         kt_did_t        t_did;  /* thread id for kernel debuggers */
 154         caddr_t t_tnf_tpdp;     /* Trace facility data pointer */
 155         struct _kcpc_ctx *t_cpc_ctx;    /* performance counter context */
 156         struct _kcpc_set *t_cpc_set;    /* set this thread has bound */

 158         /*
 159          * non swappable part of the lwp state.
 160          */
 158         id_t            t_tid;          /* lwp's id */
 159         id_t            t_waitfor;      /* target lwp id in lwp_wait() */
 160         struct sigqueue *t_sigqueue;    /* queue of siginfo structs */
 161         k_sigset_t      t_sig;          /* signals pending to this process */
 162         k_sigset_t      t_extsig;       /* signals sent from another contract */
 163         k_sigset_t      t_hold;         /* hold signal bit mask */
 164         k_sigset_t      t_sigwait;      /* sigtimedwait() is accepting these */
 165         struct  _kthread *t_forw;       /* process's forward thread link */
 166         struct  _kthread *t_back;       /* process's backward thread link */
 167         struct  _kthread *t_thlink;     /* tid (lwpid) lookup hash link */
 168         klwp_t  *t_lwp;                 /* thread's lwp pointer */
 169         struct  proc    *t_procp;       /* proc pointer */
 170         struct  t_audit_data *t_audit_data;     /* per thread audit data */
 171         struct  _kthread *t_next;       /* doubly linked list of all threads */
 172         struct  _kthread *t_prev;
 173         ushort_t t_whystop;             /* reason for stopping */
 174         ushort_t t_whatstop;            /* more detailed reason */
 175         int     t_dslot;                /* index in proc's thread directory */
 176         struct  pollstate *t_pollstate; /* state used during poll(2) */
 177         struct  pollcache *t_pollcache; /* to pass a pcache ptr by /dev/poll */
 178         struct  cred    *t_cred;        /* pointer to current cred */
 179         time_t  t_start;                /* start time, seconds since epoch */
 180         clock_t t_lbolt;                /* lbolt at last clock_tick() */
 181         hrtime_t t_stoptime;            /* timestamp at stop() */
 182         uint_t  t_pctcpu;               /* %cpu at last clock_tick(), binary */
 183                                         /* point at right of high-order bit */
 184         short   t_sysnum;               /* system call number */
 185         kcondvar_t      t_delay_cv;
 186         kmutex_t        t_delay_lock;

 188         /*
 189          * Pointer to the dispatcher lock protecting t_state and state-related
 190          * flags.  This pointer can change during waits on the lock, so
 191          * it should be grabbed only by thread_lock().
 192          */
 193         disp_lock_t     *t_lockp;       /* pointer to the dispatcher lock */
 194         ushort_t        t_oldspl;       /* spl level before dispatcher locked */
 195         volatile char   t_pre_sys;      /* pre-syscall work needed */
 196         lock_t          t_lock_flush;   /* for lock_mutex_flush() impl */
 197         struct _disp    *t_disp_queue;  /* run queue for chosen CPU */
 198         clock_t         t_disp_time;    /* last time this thread was running */
 199         uint_t          t_kpri_req;     /* kernel priority required */

 201         /*
 202          * Post-syscall / post-trap flags.
 203          *      No lock is required to set these.
 204          *      These must be cleared only by the thread itself.
 205          *
 206          *      t_astflag indicates that some post-trap processing is required,
 207          *              possibly a signal or a preemption.  The thread will not
 208          *              return to user with this set.
 209          *      t_post_sys indicates that some unusualy post-system call
 210          *              handling is required, such as an error or tracing.
 211          *      t_sig_check indicates that some condition in ISSIG() must be
```

```
212          *              checked, but doesn't prevent returning to user.
213          *      t_post_sys_ast is a way of checking whether any of these three
214          *              flags are set.
215          */
216         union __tu {
217                 struct __ts {
218                         volatile char   _t_astflag;     /* AST requested */
219                         volatile char   _t_sig_check;   /* ISSIG required */
220                         volatile char   _t_post_sys;    /* post_syscall req */
221                         volatile char   _t_trapret;     /* call CL_TRAPRET */
222                 } _ts;
223                 volatile int    _t_post_sys_ast;        /* OR of these flags */
224         } _tu;
225 #define t_astflag       _tu._ts._t_astflag
226 #define t_sig_check     _tu._ts._t_sig_check
227 #define t_post_sys      _tu._ts._t_post_sys
228 #define t_trapret       _tu._ts._t_trapret
229 #define t_post_sys_ast  _tu._t_post_sys_ast

231         /*
232          * Real time microstate profiling.
233          */
234                                         /* possible 4-byte filler */
235         hrtime_t t_waitrq;              /* timestamp for run queue wait time */
236         int     t_mstate;               /* current microstate */
237         struct rprof {
238                 int     rp_anystate;            /* set if any state non-zero */
239                 uint_t  rp_state[NMSTATES];     /* mstate profiling counts */
240         } *t_rprof;

242         /*
243          * There is a turnstile inserted into the list below for
244          * every priority inverted synchronization object that
245          * this thread holds.
246          */

248         struct turnstile *t_prioinv;

250         /*
251          * Pointer to the turnstile attached to the synchronization
252          * object where this thread is blocked.
253          */

255         struct turnstile *t_ts;

257         /*
258          * kernel thread specific data
259          *      Borrowed from userland implementation of POSIX tsd
260          */
261         struct tsd_thread {
262                 struct tsd_thread *ts_next;     /* threads with TSD */
263                 struct tsd_thread *ts_prev;     /* threads with TSD */
264                 uint_t            ts_nkeys;     /* entries in value array */
265                 void              **ts_value;   /* array of value/key */
266         } *t_tsd;

271         clock_t         t_stime;        /* time stamp used by the swapper */
268         struct door_data *t_door;       /* door invocation data */
269         kmutex_t        *t_plockp;      /* pointer to process's p_lock */

271         struct sc_shared *t_schedctl;   /* scheduler activations shared data */
272         uintptr_t       t_sc_uaddr;     /* user-level address of shared data */

274         struct cpupart  *t_cpupart;     /* partition containing thread */
275         int             t_bind_pset;    /* processor set binding */
```



```
277         struct copyops  *t_copyops;     /* copy in/out ops vector */

279         caddr_t         t_stkbase;      /* base of the the stack */
280         struct page     *t_red_pp;      /* if non-NULL, redzone is mapped */

282         afd_t           t_activefd;     /* active file descriptor table */

284         struct _kthread *t_priforw;     /* sleepq per-priority sublist */
285         struct _kthread *t_priback;

287         struct sleepq   *t_sleepq;      /* sleep queue thread is waiting on */
288         struct panic_trap_info *t_panic_trap;   /* saved data from fatal trap */
289         int             *t_lgrp_affinity;       /* lgroup affinity */
290         struct upimutex *t_upimutex;    /* list of upimutexes owned by thread */
291         uint32_t        t_nupinest;     /* number of nested held upi mutexes */
292         struct kproject *t_proj;        /* project containing this thread */
293         uint8_t         t_unpark;       /* modified holding t_delay_lock */
294         uint8_t         t_release;      /* lwp_release() waked up the thread */
295         uint8_t         t_hatdepth;     /* depth of recursive hat_memloads */
296         uint8_t         t_xpvcntr;      /* see xen_block_migrate() */
297         kcondvar_t      t_joincv;       /* cv used to wait for thread exit */
298         void            *t_taskq;       /* for threads belonging to taskq */
299         hrtime_t        t_anttime;      /* most recent time anticipatory load */
300                                         /*      was added to an lgroup's load */
301                                         /*      on this thread's behalf */
302         char            *t_pdmsg;       /* privilege debugging message */

304         uint_t          t_predcache;    /* DTrace predicate cache */
305         hrtime_t        t_dtrace_vtime; /* DTrace virtual time */
306         hrtime_t        t_dtrace_start; /* DTrace slice start time */

308         uint8_t         t_dtrace_stop;  /* indicates a DTrace-desired stop */
309         uint8_t         t_dtrace_sig;   /* signal sent via DTrace's raise() */

311         union __tdu {
312                 struct __tds {
313                         uint8_t _t_dtrace_on;   /* hit a fasttrap tracepoint */
314                         uint8_t _t_dtrace_step; /* about to return to kernel */
315                         uint8_t _t_dtrace_ret;  /* handling a return probe */
316                         uint8_t _t_dtrace_ast;  /* saved ast flag */
317 #ifdef __amd64
318                         uint8_t _t_dtrace_reg;  /* modified register */
319 #endif
320                 } _tds;
321                 ulong_t _t_dtrace_ft;           /* bitwise or of these flags */
322         } _tdu;
323 #define t_dtrace_ft     _tdu._t_dtrace_ft
324 #define t_dtrace_on     _tdu._tds._t_dtrace_on
325 #define t_dtrace_step   _tdu._tds._t_dtrace_step
326 #define t_dtrace_ret    _tdu._tds._t_dtrace_ret
327 #define t_dtrace_ast    _tdu._tds._t_dtrace_ast
328 #ifdef __amd64
329 #define t_dtrace_reg    _tdu._tds._t_dtrace_reg
330 #endif

332         uintptr_t       t_dtrace_pc;    /* DTrace saved pc from fasttrap */
333         uintptr_t       t_dtrace_npc;   /* DTrace next pc from fasttrap */
334         uintptr_t       t_dtrace_scrpc; /* DTrace per-thread scratch location */
335         uintptr_t       t_dtrace_astpc; /* DTrace return sequence location */
336 #ifdef __amd64
337         uint64_t        t_dtrace_regv;  /* DTrace saved reg from fasttrap */
338 #endif
339         hrtime_t        t_hrtime;       /* high-res last time on cpu */
340         kmutex_t        t_ctx_lock;     /* protects t_ctx in removectx() */
341         struct waitq    *t_waitq;       /* wait queue */
342         kmutex_t        t_wait_mutex;   /* used in CV wait functions */
```

```
343 } kthread_t;

345 /*
346  * Thread flag (t_flag) definitions.
347  *      These flags must be changed only for the current thread,
348  *      and not during preemption code, since the code being
349  *      preempted could be modifying the flags.
350  *
351  *      For the most part these flags do not need locking.
352  *      The following flags will only be changed while the thread_lock is held,
353  *      to give assurrance that they are consistent with t_state:
354  *              T_WAKEABLE
355  */
356 #define T_INTR_THREAD   0x0001  /* thread is an interrupt thread */
357 #define T_WAKEABLE      0x0002  /* thread is blocked, signals enabled */
358 #define T_TOMASK        0x0004  /* use lwp_sigoldmask on return from signal */
359 #define T_TALLOCSTK     0x0008  /* thread structure allocated from stk */
360 #define T_FORKALL       0x0010  /* thread was cloned by forkall() */
361 #define T_WOULDBLOCK    0x0020  /* for lockfs */
362 #define T_DONTBLOCK     0x0040  /* for lockfs */
363 #define T_DONTPEND      0x0080  /* for lockfs */
364 #define T_SYS_PROF      0x0100  /* profiling on for duration of system call */
365 #define T_WAITCVSEM     0x0200  /* waiting for a lwp_cv or lwp_sema on sleepq */
366 #define T_WATCHPT       0x0400  /* thread undergoing a watchpoint emulation */
367 #define T_PANIC         0x0800  /* thread initiated a system panic */
368 #define T_LWPREUSE      0x1000  /* stack and LWP can be reused */
369 #define T_CAPTURING     0x2000  /* thread is in page capture logic */
370 #define T_VFPARENT      0x4000  /* thread is vfork parent, must call vfwait */
371 #define T_DONTDTRACE    0x8000  /* disable DTrace probes */

373 /*
374  * Flags in t_proc_flag.
375  *      These flags must be modified only when holding the p_lock
376  *      for the associated process.
377  */
378 #define TP_DAEMON       0x0001  /* this is an LWP_DAEMON lwp */
379 #define TP_HOLDLWP      0x0002  /* hold thread's lwp */
380 #define TP_TWAIT        0x0004  /* wait to be freed by lwp_wait() */
381 #define TP_LWPEXIT      0x0008  /* lwp has exited */
382 #define TP_PRSTOP       0x0010  /* thread is being stopped via /proc */
383 #define TP_CHKPT        0x0020  /* thread is being stopped via CPR checkpoint */
384 #define TP_EXITLWP      0x0040  /* terminate this lwp */
385 #define TP_PRVSTOP      0x0080  /* thread is virtually stopped via /proc */
386 #define TP_MSACCT       0x0100  /* collect micro-state accounting information */
387 #define TP_STOPPING     0x0200  /* thread is executing stop() */
388 #define TP_WATCHPT      0x0400  /* process has watchpoints in effect */
389 #define TP_PAUSE        0x0800  /* process is being stopped via pauselwps() */
390 #define TP_CHANGEBIND   0x1000  /* thread has a new cpu/cpupart binding */
391 #define TP_ZTHREAD      0x2000  /* this is a kernel thread for a zone */
392 #define TP_WATCHSTOP    0x4000  /* thread is stopping via holdwatch() */

394 /*
395  * Thread scheduler flag (t_schedflag) definitions.
396  *      The thread must be locked via thread_lock() or equiv. to change these.
397  */
402 #define TS_LOAD         0x0001  /* thread is in memory */
403 #define TS_DONT_SWAP    0x0002  /* thread/lwp should not be swapped */
404 #define TS_SWAPENQ      0x0004  /* swap thread when it reaches a safe point */
405 #define TS_ON_SWAPQ     0x0008  /* thread is on the swap queue */
398 #define TS_SIGNALLED    0x0010  /* thread was awakened by cv_signal() */
399 #define TS_PROJWAITQ    0x0020  /* thread is on its project's waitq */
400 #define TS_ZONEWAITQ    0x0040  /* thread is on its zone's waitq */
401 #define TS_CSTART       0x0100  /* setrun() by continuelwps() */
402 #define TS_UNPAUSE      0x0200  /* setrun() by unpauselwps() */
403 #define TS_XSTART       0x0400  /* setrun() by SIGCONT */
404 #define TS_PSTART       0x0800  /* setrun() by /proc */
```



```
405 #define TS_RESUME       0x1000  /* setrun() by CPR resume process */
406 #define TS_CREATE       0x2000  /* setrun() by syslwp_create() */
407 #define TS_RUNQMATCH    0x4000  /* exact run queue balancing by setbackdq() */
408 #define TS_ALLSTART     \
409         (TS_CSTART|TS_UNPAUSE|TS_XSTART|TS_PSTART|TS_RESUME|TS_CREATE)
410 #define TS_ANYWAITQ     (TS_PROJWAITQ|TS_ZONEWAITQ)

412 /*
413  * Thread binding types
414  */
415 #define TB_ALLHARD      0
416 #define TB_CPU_SOFT     0x01            /* soft binding to CPU */
417 #define TB_PSET_SOFT    0x02            /* soft binding to pset */

419 #define TB_CPU_SOFT_SET(t)              ((t)->t_bindflag |= TB_CPU_SOFT)
420 #define TB_CPU_HARD_SET(t)              ((t)->t_bindflag &= ~TB_CPU_SOFT)
421 #define TB_PSET_SOFT_SET(t)             ((t)->t_bindflag |= TB_PSET_SOFT)
422 #define TB_PSET_HARD_SET(t)             ((t)->t_bindflag &= ~TB_PSET_SOFT)
423 #define TB_CPU_IS_SOFT(t)               ((t)->t_bindflag & TB_CPU_SOFT)
424 #define TB_CPU_IS_HARD(t)               (!TB_CPU_IS_SOFT(t))
425 #define TB_PSET_IS_SOFT(t)              ((t)->t_bindflag & TB_PSET_SOFT)

427 /*
428  * No locking needed for AST field.
429  */
430 #define aston(t)                ((t)->t_astflag = 1)
431 #define astoff(t)               ((t)->t_astflag = 0)

433 /* True if thread is stopped on an event of interest */
434 #define ISTOPPED(t) ((t)->t_state == TS_STOPPED && \
435                         !((t)->t_schedflag & TS_PSTART))

437 /* True if thread is asleep and wakeable */
438 #define ISWAKEABLE(t) (((t)->t_state == TS_SLEEP && \
439                         ((t)->t_flag & T_WAKEABLE)))

441 /* True if thread is on the wait queue */
442 #define ISWAITING(t) ((t)->t_state == TS_WAIT)

444 /* similar to ISTOPPED except the event of interest is CPR */
445 #define CPR_ISTOPPED(t) ((t)->t_state == TS_STOPPED && \
446                         !((t)->t_schedflag & TS_RESUME))

448 /*
449  * True if thread is virtually stopped (is or was asleep in
450  * one of the lwp_*() system calls and marked to stop by /proc.)
451  */
452 #define VSTOPPED(t)     ((t)->t_proc_flag & TP_PRVSTOP)

454 /* similar to VSTOPPED except the point of interest is CPR */
455 #define CPR_VSTOPPED(t)                         \
456         ((t)->t_state == TS_SLEEP &&            \
457         (t)->t_wchan0 != NULL &&                \
458         ((t)->t_flag & T_WAKEABLE) &&           \
459         ((t)->t_proc_flag & TP_CHKPT))

461 /* True if thread has been stopped by hold*() or was created stopped */
462 #define SUSPENDED(t) ((t)->t_state == TS_STOPPED && \
463         ((t)->t_schedflag & (TS_CSTART|TS_UNPAUSE)) != (TS_CSTART|TS_UNPAUSE))

465 /* True if thread possesses an inherited priority */
466 #define INHERITED(t)    ((t)->t_epri != 0)

468 /* The dispatch priority of a thread */
469 #define DISP_PRIO(t) ((t)->t_epri > (t)->t_pri ? (t)->t_epri : (t)->t_pri)
```

```
471 /* The assigned priority of a thread */
472 #define ASSIGNED_PRIO(t)        ((t)->t_pri)

474 /*
483  * Macros to determine whether a thread can be swapped.
484  * If t_lock is held, the thread is either on a processor or being swapped.
485  */
486 #define SWAP_OK(t)      (!LOCK_HELD(&(t)->t_lock))

488 /*
475  * proctot(x)
476  *      convert a proc pointer to a thread pointer. this only works with
477  *      procs that have only one lwp.
478  *
479  * proctolwp(x)
480  *      convert a proc pointer to a lwp pointer. this only works with
481  *      procs that have only one lwp.
482  *
483  * ttolwp(x)
484  *      convert a thread pointer to its lwp pointer.
485  *
486  * ttoproc(x)
487  *      convert a thread pointer to its proc pointer.
488  *
489  * ttoproj(x)
490  *      convert a thread pointer to its project pointer.
491  *
492  * ttozone(x)
493  *      convert a thread pointer to its zone pointer.
494  *
495  * lwptot(x)
496  *      convert a lwp pointer to its thread pointer.
497  *
498  * lwptoproc(x)
499  *      convert a lwp to its proc pointer.
500  */
501 #define proctot(x)      ((x)->p_tlist)
502 #define proctolwp(x)    ((x)->p_tlist->t_lwp)
503 #define ttolwp(x)       ((x)->t_lwp)
504 #define ttoproc(x)      ((x)->t_procp)
505 #define ttoproj(x)      ((x)->t_proj)
506 #define ttozone(x)      ((x)->t_procp->p_zone)
507 #define lwptot(x)       ((x)->lwp_thread)
508 #define lwptoproc(x)    ((x)->lwp_procp)

510 #define t_pc            t_pcb.val[0]
511 #define t_sp            t_pcb.val[1]

513 #ifdef  _KERNEL

515 extern  kthread_t       *threadp(void); /* inline, returns thread pointer */
516 #define curthread       (threadp())             /* current thread pointer */
517 #define curproc         (ttoproc(curthread))    /* current process pointer */
518 #define curproj         (ttoproj(curthread))    /* current project pointer */
519 #define curzone         (curproc->p_zone)       /* current zone pointer */

521 extern  struct _kthread t0;             /* the scheduler thread */
522 extern  kmutex_t        pidlock;        /* global process lock */

524 /*
525  * thread_free_lock is used by the tick accounting thread to keep a thread
526  * from being freed while it is being examined.
527  *
528  * Thread structures are 32-byte aligned structures. That is why we use the
529  * following formula.
530  */
```



```
531 #define THREAD_FREE_BITS        10
532 #define THREAD_FREE_NUM         (1 << THREAD_FREE_BITS)
533 #define THREAD_FREE_MASK        (THREAD_FREE_NUM - 1)
534 #define THREAD_FREE_1           PTR24_LSB
535 #define THREAD_FREE_2           (PTR24_LSB + THREAD_FREE_BITS)
536 #define THREAD_FREE_SHIFT(t)    \
537         (((ulong_t)(t) >> THREAD_FREE_1) ^ ((ulong_t)(t) >> THREAD_FREE_2))
538 #define THREAD_FREE_HASH(t)     (THREAD_FREE_SHIFT(t) & THREAD_FREE_MASK)

540 typedef struct thread_free_lock {
541         kmutex_t        tf_lock;
542         uchar_t         tf_pad[64 - sizeof (kmutex_t)];
543 } thread_free_lock_t;
_____unchanged_portion_omitted_

612 /*
613  * Macros to change thread state and the associated lock.
614  */
615 #define THREAD_SET_STATE(tp, state, lp) \
616                 ((tp)->t_state = state, (tp)->t_lockp = lp)

618 /*
619  * Point it at the transition lock, which is always held.
620  * The previosly held lock is dropped.
621  */
622 #define THREAD_TRANSITION(tp)   thread_transition(tp);
623 /*
624  * Set the thread's lock to be the transition lock, without dropping
625  * previosly held lock.
626  */
627 #define THREAD_TRANSITION_NOLOCK(tp)    ((tp)->t_lockp = &transition_lock)

629 /*
630  * Put thread in run state, and set the lock pointer to the dispatcher queue
631  * lock pointer provided.  This lock should be held.
632  */
633 #define THREAD_RUN(tp, lp)      THREAD_SET_STATE(tp, TS_RUN, lp)

635 /*
636  * Put thread in wait state, and set the lock pointer to the wait queue
637  * lock pointer provided.  This lock should be held.
638  */
639 #define THREAD_WAIT(tp, lp)     THREAD_SET_STATE(tp, TS_WAIT, lp)

655 /*
656  * Put thread in run state, and set the lock pointer to the dispatcher queue
657  * lock pointer provided (i.e., the "swapped_lock").  This lock should be held.
658  */
659 #define THREAD_SWAP(tp, lp)     THREAD_SET_STATE(tp, TS_RUN, lp)

641 /*
642  * Put the thread in zombie state and set the lock pointer to NULL.
643  * The NULL will catch anything that tries to lock a zombie.
644  */
645 #define THREAD_ZOMB(tp)         THREAD_SET_STATE(tp, TS_ZOMB, NULL)

647 /*
648  * Set the thread into ONPROC state, and point the lock at the CPUs
649  * lock for the onproc thread(s).  This lock should be held, so the
650  * thread deoes not become unlocked, since these stores can be reordered.
651  */
652 #define THREAD_ONPROC(tp, cpu)  \
653                 THREAD_SET_STATE(tp, TS_ONPROC, &(cpu)->cpu_thread_lock)

655 /*
656  * Set the thread into the TS_SLEEP state, and set the lock pointer to
```

```
 657  * to some sleep queue's lock.  The new lock should already be held.
 658  */
 659 #define THREAD_SLEEP(tp, lp)    {                                       \
 660                         disp_lock_t     *tlp;                           \
 661                         tlp = (tp)->t_lockp;                            \
 662                         THREAD_SET_STATE(tp, TS_SLEEP, lp);             \
 663                         disp_lock_exit_high(tlp);                       \
 664                         }

 666 /*
 667  * Interrupt threads are created in TS_FREE state, and their lock
 668  * points at the associated CPU's lock.
 669  */
 670 #define THREAD_FREEINTR(tp, cpu)        \
 671                 THREAD_SET_STATE(tp, TS_FREE, &(cpu)->cpu_thread_lock)

 673 /* if tunable kmem_stackinfo is set, fill kthread stack with a pattern */
 674 #define KMEM_STKINFO_PATTERN    0xbadcbadcbadcbadcULL

 676 /*
 677  * If tunable kmem_stackinfo is set, log the latest KMEM_LOG_STK_USAGE_SIZE
 678  * dead kthreads that used their kernel stack the most.
 679  */
 680 #define KMEM_STKINFO_LOG_SIZE   16

 682 /* kthread name (cmd/lwpid) string size in the stackinfo log */
 683 #define KMEM_STKINFO_STR_SIZE   64

 685 /*
 686  * stackinfo logged data.
 687  */
 688 typedef struct kmem_stkinfo {
 689         caddr_t kthread;        /* kthread pointer */
 690         caddr_t t_startpc;      /* where kthread started */
 691         caddr_t start;          /* kthread stack start address */
 692         size_t  stksz;          /* kthread stack size */
 693         size_t  percent;        /* kthread stack high water mark */
 694         id_t    t_tid;          /* kthread id */
 695         char    cmd[KMEM_STKINFO_STR_SIZE];     /* kthread name (cmd/lwpid) */
 696 } kmem_stkinfo_t;
```

*unchanged_portion_omitted_*

```
**********************************************************
    5125 Fri May  8 18:10:33 2015
new/usr/src/uts/common/sys/vmsystm.h
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
```

```
   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 /*      Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T     */
  27 /*        All Rights Reserved   */

  29 /*
  30  * University Copyright- Copyright (c) 1982, 1986, 1988
  31  * The Regents of the University of California
  32  * All Rights Reserved
  33  *
  34  * University Acknowledgment- Portions of this document are derived from
  35  * software developed by the University of California, Berkeley, and its
  36  * contributors.
  37  */

  39 #ifndef _SYS_VMSYSTM_H
  40 #define _SYS_VMSYSTM_H

  42 #include <sys/proc.h>

  44 #ifdef __cplusplus
  45 extern "C" {
  46 #endif

  48 /*
  49  * Miscellaneous virtual memory subsystem variables and structures.
  50  */
  51 #ifdef _KERNEL
  52 extern pgcnt_t  freemem;        /* remaining blocks of free memory */
  53 extern pgcnt_t  avefree;        /* 5 sec moving average of free memory */
  54 extern pgcnt_t  avefree30;      /* 30 sec moving average of free memory */
  55 extern pgcnt_t  deficit;        /* estimate of needs of new swapped in procs */
```



```
  56 extern pgcnt_t  nscan;          /* number of scans in last second */
  57 extern pgcnt_t  desscan;        /* desired pages scanned per second */
  58 extern pgcnt_t  slowscan;
  59 extern pgcnt_t  fastscan;
  60 extern pgcnt_t  pushes;         /* number of pages pushed to swap device */

  62 /* writable copies of tunables */
  63 extern pgcnt_t  maxpgio;        /* max paging i/o per sec before start swaps */
  64 extern pgcnt_t  lotsfree;       /* max free before clock freezes */
  65 extern pgcnt_t  desfree;        /* minimum free pages before swapping begins */
  66 extern pgcnt_t  minfree;        /* no of pages to try to keep free via daemon */
  67 extern pgcnt_t  needfree;       /* no of pages currently being waited for */
  68 extern pgcnt_t  throttlefree;   /* point at which we block PG_WAIT calls */
  69 extern pgcnt_t  pageout_reserve; /* point at which we deny non-PG_WAIT calls */
  70 extern pgcnt_t  pages_before_pager; /* XXX */

  72 /*
  73  * TRUE if the pageout daemon, fsflush daemon or the scheduler.  These
  74  * processes can't sleep while trying to free up memory since a deadlock
  75  * will occur if they do sleep.
  76  */
  77 #define NOMEMWAIT() (ttoproc(curthread) == proc_pageout || \
  78                      ttoproc(curthread) == proc_fsflush || \
  79                      ttoproc(curthread) == proc_sched)

  81 /* insure non-zero */
  82 #define nz(x)   ((x) != 0 ? (x) : 1)

  84 /*
  85  * Flags passed by the swapper to swapout routines of each
  86  * scheduling class.
  87  */
  88 #define HARDSWAP        1
  89 #define SOFTSWAP        2

  91 /*
  92  * Values returned by valid_usr_range()
  93  */
  94 #define RANGE_OKAY      (0)
  95 #define RANGE_BADADDR   (1)
  96 #define RANGE_BADPROT   (2)

  98 /*
  99  * map_pgsz: temporary - subject to change.
 100  */
 101 #define MAPPGSZ_VA      0x01
 102 #define MAPPGSZ_STK     0x02
 103 #define MAPPGSZ_HEAP    0x04
 104 #define MAPPGSZ_ISM     0x08

 106 /*
 107  * Flags for map_pgszcvec
 108  */
 109 #define MAPPGSZC_SHM    0x01
 110 #define MAPPGSZC_PRIVM  0x02
 111 #define MAPPGSZC_STACK  0x04
 112 #define MAPPGSZC_HEAP   0x08

 114 /*
 115  * vacalign values for choose_addr
 116  */
 117 #define ADDR_NOVACALIGN 0
 118 #define ADDR_VACALIGN   1

 120 struct as;
 121 struct page;
```

```
122 struct anon;

124 extern int maxslp;
124 extern ulong_t pginrate;
125 extern ulong_t pgoutrate;
127 extern void swapout_lwp(klwp_t *);

127 extern  int valid_va_range(caddr_t *basep, size_t *lenp, size_t minlen,
128                 int dir);
129 extern  int valid_va_range_aligned(caddr_t *basep, size_t *lenp,
130     size_t minlen, int dir, size_t align, size_t redzone, size_t off);

132 extern  int valid_usr_range(caddr_t, size_t, uint_t, struct as *, caddr_t);
133 extern  int useracc(void *, size_t, int);
134 extern  size_t map_pgsz(int maptype, struct proc *p, caddr_t addr, size_t len,
135     int memcntl);
136 extern  uint_t map_pgszcvec(caddr_t addr, size_t size, uintptr_t off, int flags,
137     int type, int memcntl);
138 extern int choose_addr(struct as *as, caddr_t *addrp, size_t len, offset_t off,
139     int vacalign, uint_t flags);
140 extern  void map_addr(caddr_t *addrp, size_t len, offset_t off, int vacalign,
141     uint_t flags);
142 extern  int map_addr_vacalign_check(caddr_t, u_offset_t);
143 extern  void map_addr_proc(caddr_t *addrp, size_t len, offset_t off,
144     int vacalign, caddr_t userlimit, struct proc *p, uint_t flags);
145 extern  void vmmeter(void);
146 extern  int cow_mapin(struct as *, caddr_t, caddr_t, struct page **,
147         struct anon **, size_t *, int);

149 extern  caddr_t ppmapin(struct page *, uint_t, caddr_t);
150 extern  void    ppmapout(caddr_t);

152 extern  int pf_is_memory(pfn_t);

154 extern  void    dcache_flushall(void);

156 extern  void    *boot_virt_alloc(void *addr, size_t size);

158 extern  size_t  exec_get_spslew(void);

160 #endif  /* _KERNEL */

162 #ifdef  __cplusplus
163 }
```

_____unchanged_portion_omitted_

```
**********************************************************
    4234 Fri May  8 18:10:33 2015
new/usr/src/uts/common/sys/watchpoint.h
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

  75 /* wp_flags */
  76 #define WP_NOWATCH      0x01    /* protections temporarily restored */
  77 #define WP_SETPROT      0x02    /* segop_setprot() needed on this page */

  79 #ifdef  _KERNEL

  81 /*
  82  * These functions handle the necessary logic to perform the copy operation
  83  * while ignoring watchpoints.
  84  */
  85 extern int copyin_nowatch(const void *, void *, size_t);
  86 extern int copyout_nowatch(const void *, void *, size_t);
  87 extern int fuword32_nowatch(const void *, uint32_t *);
  88 extern int suword32_nowatch(void *, uint32_t);
  89 #ifdef _LP64
  90 extern int suword64_nowatch(void *, uint64_t);
  91 extern int fuword64_nowatch(const void *, uint64_t *);
  92 #endif

  94 /*
  95  * Disable watchpoints for a given region of memory.  When bracketed by these
  96  * calls, functions can use copyops and ignore watchpoints.
  97  */
  98 extern int watch_disable_addr(const void *, size_t, enum seg_rw);
  99 extern void watch_enable_addr(const void *, size_t, enum seg_rw);

 101 /*
 102  * Enable/Disable watchpoints for an entire thread.
 103  */
 104 extern  void    watch_enable(kthread_id_t);
 105 extern  void    watch_disable(kthread_id_t);

 107 struct as;
 108 struct proc;
 109 struct k_siginfo;
 110 extern  void    setallwatch(void);
 111 extern  int     pr_is_watchpage(caddr_t, enum seg_rw);
 112 extern  int     pr_is_watchpage_as(caddr_t, enum seg_rw, struct as *);
 113 extern  int     pr_is_watchpoint(caddr_t *, int *, size_t, size_t *,
 114                         enum seg_rw);
 115 extern  void    do_watch_step(caddr_t, size_t, enum seg_rw, int, greg_t);
 116 extern  int     undo_watch_step(struct k_siginfo *);
 117 extern  int     wp_compare(const void *, const void *);
 118 extern  int     wa_compare(const void *, const void *);

 120 extern  struct copyops watch_copyops;

 122 extern watched_area_t *pr_find_watched_area(struct proc *, watched_area_t *,
 123     avl_index_t *);

 125 #endif

 127 #ifdef  __cplusplus
 128 }
______unchanged_portion_omitted_
```

```
**********************************************************
   23749 Fri May  8 18:10:33 2015
new/usr/src/uts/common/syscall/utssys.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

313 static fu_data_t *
314 dofusers(vnode_t *fvp, int flags)
315 {
316         fu_data_t       *fu_data;
317         proc_t          *prp;
318         vfs_t           *cvfsp;
319         pid_t           npids, pidx, *pidlist;
320         int             v_proc = v.v_proc;      /* max # of procs */
321         int             pcnt = 0;
322         int             contained = (flags & F_CONTAINED);
323         int             nbmandonly = (flags & F_NBMANDLIST);
324         int             dip_usage = (flags & F_DEVINFO);
325         int             fvp_isdev = vn_matchops(fvp, spec_getvnodeops());
326         zone_t *zone = curproc->p_zone;
327         int inglobal = INGLOBALZONE(curproc);

329         /* get a pointer to the file system containing this vnode */
330         cvfsp = fvp->v_vfsp;
331         ASSERT(cvfsp);

333         /* allocate the data structure to return our results in */
334         fu_data = kmem_alloc(fu_data_size(v_proc), KM_SLEEP);
335         fu_data->fud_user_max = v_proc;
336         fu_data->fud_user_count = 0;

338         /* get a snapshot of all the pids we're going to check out */
339         pidlist = kmem_alloc(v_proc * sizeof (pid_t), KM_SLEEP);
340         mutex_enter(&pidlock);
341         for (npids = 0, prp = practive; prp != NULL; prp = prp->p_next) {
342                 if (inglobal || prp->p_zone == zone)
343                         pidlist[npids++] = prp->p_pid;
344         }
345         mutex_exit(&pidlock);

347         /* grab each process and check its file usage */
348         for (pidx = 0; pidx < npids; pidx++) {
349                 locklist_t      *llp = NULL;
350                 uf_info_t       *fip;
351                 vnode_t         *vp;
352                 user_t          *up;
353                 sess_t          *sp;
354                 uid_t           uid;
355                 pid_t           pid = pidlist[pidx];
356                 int             i, use_flag = 0;

358                 /*
359                  * grab prp->p_lock using sprlock()
360                  * if sprlock() fails the process does not exists anymore
361                  */
362                 prp = sprlock(pid);
363                 if (prp == NULL)
364                         continue;

366                 /* get the processes credential info in case we need it */
367                 mutex_enter(&prp->p_crlock);
368                 uid = crgetruid(prp->p_cred);
369                 mutex_exit(&prp->p_crlock);

371                 /*
```



```
372                  * it's safe to drop p_lock here because we
373                  * called sprlock() before and it set the SPRLOCK
374                  * flag for the process so it won't go away.
375                  */
376                 mutex_exit(&prp->p_lock);

378                 /*
379                  * now we want to walk a processes open file descriptors
380                  * to do this we need to grab the fip->fi_lock.  (you
381                  * can't hold p_lock when grabbing the fip->fi_lock.)
382                  */
383                 fip = P_FINFO(prp);
384                 mutex_enter(&fip->fi_lock);

386                 /*
387                  * Snapshot nbmand locks for pid
388                  */
389                 llp = flk_active_nbmand_locks(prp->p_pid);
390                 for (i = 0; i < fip->fi_nfiles; i++) {
391                         uf_entry_t      *ufp;
392                         file_t          *fp;

394                         UF_ENTER(ufp, fip, i);
395                         if (((fp = ufp->uf_file) == NULL) ||
396                             ((vp = fp->f_vnode) == NULL)) {
397                                 UF_EXIT(ufp);
398                                 continue;
399                         }

401                         /*
402                          * if the target file (fvp) is not a device
403                          * and corrosponds to the root of a filesystem
404                          * (cvfsp), then check if it contains the file
405                          * is use by this process (vp).
406                          */
407                         if (contained && (vp->v_vfsp == cvfsp))
408                                 use_flag |= F_OPEN;

410                         /*
411                          * if the target file (fvp) is not a device,
412                          * then check if it matches the file in use
413                          * by this process (vp).
414                          */
415                         if (!fvp_isdev && VN_CMP(fvp, vp))
416                                 use_flag |= F_OPEN;

418                         /*
419                          * if the target file (fvp) is a device,
420                          * then check if the current file in use
421                          * by this process (vp) maps to the same device
422                          * minor node.
423                          */
424                         if (fvp_isdev &&
425                             vn_matchops(vp, spec_getvnodeops()) &&
426                             (fvp->v_rdev == vp->v_rdev))
427                                 use_flag |= F_OPEN;

429                         /*
430                          * if the target file (fvp) is a device,
431                          * and we're checking for device instance
432                          * usage, then check if the current file in use
433                          * by this process (vp) maps to the same device
434                          * instance.
435                          */
436                         if (dip_usage &&
437                             vn_matchops(vp, spec_getvnodeops()) &&
```

```
438                             (VTOCS(fvp)->s_dip == VTOCS(vp)->s_dip))
439                                 use_flag |= F_OPEN;
441                         /*
442                          * if the current file in use by this process (vp)
443                          * doesn't match what we're looking for, move on
444                          * to the next file in the process.
445                          */
446                         if ((use_flag & F_OPEN) == 0) {
447                                 UF_EXIT(ufp);
448                                 continue;
449                         }
451                         if (proc_has_nbmand_on_vp(vp, prp->p_pid, llp)) {
452                                 /* A nbmand found so we're done.  */
453                                 use_flag |= F_NBM;
454                                 UF_EXIT(ufp);
455                                 break;
456                         }
457                         UF_EXIT(ufp);
458                 }
459                 if (llp)
460                         flk_free_locklist(llp);
462                 mutex_exit(&fip->fi_lock);
464                 /*
465                  * If nbmand usage tracking is desired and no nbmand was
466                  * found for this process, then no need to do further
467                  * usage tracking for this process.
468                  */
469                 if (nbmandonly && (!(use_flag & F_NBM))) {
470                         /*
471                          * grab the process lock again, clear the SPRLOCK
472                          * flag, release the process, and continue.
473                          */
474                         mutex_enter(&prp->p_lock);
475                         sprunlock(prp);
476                         continue;
477                 }
479                 /*
480                  * All other types of usage.
481                  * For the next few checks we need to hold p_lock.
482                  */
483                 mutex_enter(&prp->p_lock);
484                 up = PTOU(prp);
485                 if (fvp_isdev) {
486                         /*
487                          * if the target file (fvp) is a device
488                          * then check if it matches the processes tty
489                          *
490                          * we grab s_lock to protect ourselves against
491                          * freectty() freeing the vnode out from under us.
492                          */
493                         sp = prp->p_sessp;
494                         mutex_enter(&sp->s_lock);
495                         vp = prp->p_sessp->s_vp;
496                         if (vp != NULL) {
497                                 if (fvp->v_rdev == vp->v_rdev)
498                                         use_flag |= F_TTY;
500                                 if (dip_usage &&
501                                     (VTOCS(fvp)->s_dip == VTOCS(vp)->s_dip))
502                                         use_flag |= F_TTY;
503                         }
```



```
504                         mutex_exit(&sp->s_lock);
505                 } else {
506                         /* check the processes current working directory */
507                         if (up->u_cdir &&
508                             (VN_CMP(fvp, up->u_cdir) ||
509                             (contained && (up->u_cdir->v_vfsp == cvfsp))))
510                                 use_flag |= F_CDIR;
512                         /* check the processes root directory */
513                         if (up->u_rdir &&
514                             (VN_CMP(fvp, up->u_rdir) ||
515                             (contained && (up->u_rdir->v_vfsp == cvfsp))))
516                                 use_flag |= F_RDIR;
518                         /* check the program text vnode */
519                         if (prp->p_exec &&
520                             (VN_CMP(fvp, prp->p_exec) ||
521                             (contained && (prp->p_exec->v_vfsp == cvfsp))))
522                                 use_flag |= F_TEXT;
523                 }
525                 /* Now we can drop p_lock again */
526                 mutex_exit(&prp->p_lock);
528                 /*
529                  * now we want to walk a processes memory mappings.
530                  * to do this we need to grab the prp->p_as lock.  (you
531                  * can't hold p_lock when grabbing the prp->p_as lock.)
532                  */
533                 if (prp->p_as != &kas) {
534                         struct seg      *seg;
535                         struct as       *as = prp->p_as;
537                         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
538                         for (seg = AS_SEGFIRST(as); seg;
539                             seg = AS_SEGNEXT(as, seg)) {
540                                 /*
541                                  * if we can't get a backing vnode for this
542                                  * segment then skip it
543                                  */
544                                 vp = NULL;
545                                 if ((segop_getvp(seg, seg->s_base, &vp)) ||
545                                 if ((SEGOP_GETVP(seg, seg->s_base, &vp)) ||
546                                     (vp == NULL))
547                                         continue;
549                                 /*
550                                  * if the target file (fvp) is not a device
551                                  * and corrosponds to the root of a filesystem
552                                  * (cvfsp), then check if it contains the
553                                  * vnode backing this segment (vp).
554                                  */
555                                 if (contained && (vp->v_vfsp == cvfsp)) {
556                                         use_flag |= F_MAP;
557                                         break;
558                                 }
560                                 /*
561                                  * if the target file (fvp) is not a device,
562                                  * check if it matches the the vnode backing
563                                  * this segment (vp).
564                                  */
565                                 if (!fvp_isdev && VN_CMP(fvp, vp)) {
566                                         use_flag |= F_MAP;
567                                         break;
568                                 }
```

```
570                                 /*
571                                  * if the target file (fvp) isn't a device,
572                                  * or the the vnode backing this segment (vp)
573                                  * isn't a device then continue.
574                                  */
575                                 if (!fvp_isdev ||
576                                     !vn_matchops(vp, spec_getvnodeops()))
577                                         continue;

579                                 /*
580                                  * check if the vnode backing this segment
581                                  * (vp) maps to the same device minor node
582                                  * as the target device (fvp)
583                                  */
584                                 if (fvp->v_rdev == vp->v_rdev) {
585                                         use_flag |= F_MAP;
586                                         break;
587                                 }

589                                 /*
590                                  * if we're checking for device instance
591                                  * usage, then check if the vnode backing
592                                  * this segment (vp) maps to the same device
593                                  * instance as the target device (fvp).
594                                  */
595                                 if (dip_usage &&
596                                     (VTOCS(fvp)->s_dip == VTOCS(vp)->s_dip)) {
597                                         use_flag |= F_MAP;
598                                         break;
599                                 }
600                         }
601                         AS_LOCK_EXIT(as, &as->a_lock);
602                 }

604                 if (use_flag) {
605                         ASSERT(pcnt < fu_data->fud_user_max);
606                         fu_data->fud_user[pcnt].fu_flags = use_flag;
607                         fu_data->fud_user[pcnt].fu_pid = pid;
608                         fu_data->fud_user[pcnt].fu_uid = uid;
609                         pcnt++;
610                 }

612                 /*
613                  * grab the process lock again, clear the SPRLOCK
614                  * flag, release the process, and continue.
615                  */
616                 mutex_enter(&prp->p_lock);
617                 sprunlock(prp);
618         }

620         kmem_free(pidlist, v_proc * sizeof (pid_t));

622         fu_data->fud_user_count = pcnt;
623         return (fu_data);
624 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   18138 Fri May  8 18:10:34 2015
new/usr/src/uts/common/vm/anon.h
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

 380 extern  struct k_anoninfo k_anoninfo;

 382 extern void     anon_init(void);
 383 extern struct   anon *anon_alloc(struct vnode *, anoff_t);
 384 extern void     anon_dup(struct anon_hdr *, ulong_t,
 385                     struct anon_hdr *, ulong_t, size_t);
 386 extern void     anon_dup_fill_holes(struct anon_hdr *, ulong_t,
 387                     struct anon_hdr *, ulong_t, size_t, uint_t, int);
 388 extern int      anon_fill_cow_holes(struct seg *, caddr_t, struct anon_hdr *,
 389                     ulong_t, struct vnode *, u_offset_t, size_t, uint_t,
 390                     uint_t, struct vpage [], struct cred *);
 391 extern void     anon_free(struct anon_hdr *, ulong_t, size_t);
 392 extern void     anon_free_pages(struct anon_hdr *, ulong_t, size_t, uint_t);
 393 extern void     anon_disclaim(struct anon_map *, ulong_t, size_t);
 394 extern int      anon_getpage(struct anon **, uint_t *, struct page **,
 395                     size_t, struct seg *, caddr_t, enum seg_rw, struct cred *);
 396 extern int      swap_getconpage(struct vnode *, u_offset_t, size_t,
 397                     uint_t *, page_t *[], size_t, page_t *, uint_t *,
 398                     spgcnt_t *, struct seg *, caddr_t,
 399                     enum seg_rw, struct cred *);
 400 extern int      anon_map_getpages(struct anon_map *, ulong_t,
 401                     uint_t, struct seg *, caddr_t, uint_t,
 402                     uint_t *, page_t *[], uint_t *,
 403                     struct vpage [], enum seg_rw, int, int, int, struct cred *);
 404 extern int      anon_map_privatepages(struct anon_map *, ulong_t,
 405                     uint_t, struct seg *, caddr_t, uint_t,
 406                     page_t *[], struct vpage [], int, int, struct cred *);
 407 extern struct   page *anon_private(struct anon **, struct seg *,
 408                     caddr_t, uint_t, struct page *,
 409                     int, struct cred *);
 410 extern struct   page *anon_zero(struct seg *, caddr_t,
 411                     struct anon **, struct cred *);
 412 extern int      anon_map_createpages(struct anon_map *, ulong_t,
 413                     size_t, struct page **,
 414                     struct seg *, caddr_t,
 415                     enum seg_rw, struct cred *);
 416 extern int      anon_map_demotepages(struct anon_map *, ulong_t,
 417                     struct seg *, caddr_t, uint_t,
 418                     struct vpage [], struct cred *);
 419 extern void     anon_shmap_free_pages(struct anon_map *, ulong_t, size_t);
 420 extern int      anon_resvmem(size_t, boolean_t, zone_t *, int);
 421 extern void     anon_unresvmem(size_t, zone_t *);
 422 extern struct   anon_map *anonmap_alloc(size_t, size_t, int);
 423 extern void     anonmap_free(struct anon_map *);
 424 extern void     anonmap_purge(struct anon_map *);
 425 extern void     anon_swap_free(struct anon *, struct page *);
 426 extern void     anon_decref(struct anon *);
 427 extern int      non_anon(struct anon_hdr *, ulong_t, u_offset_t *, size_t *);
 428 extern pgcnt_t  anon_pages(struct anon_hdr *, ulong_t, pgcnt_t);
 429 extern int      anon_swap_adjust(pgcnt_t);
 430 extern void     anon_swap_restore(pgcnt_t);
 431 extern struct   anon_hdr *anon_create(pgcnt_t, int);
 432 extern void     anon_release(struct anon_hdr *, pgcnt_t);
```



```
 433 extern struct   anon *anon_get_ptr(struct anon_hdr *, ulong_t);
 434 extern ulong_t  *anon_get_slot(struct anon_hdr *, ulong_t);
 435 extern struct   anon *anon_get_next_ptr(struct anon_hdr *, ulong_t *);
 436 extern int      anon_set_ptr(struct anon_hdr *, ulong_t, struct anon *, int);
 437 extern int      anon_copy_ptr(struct anon_hdr *, ulong_t,
 438                     struct anon_hdr *, ulong_t, pgcnt_t, int);
 439 extern pgcnt_t  anon_grow(struct anon_hdr *, ulong_t *, pgcnt_t, pgcnt_t, int);
 440 extern void     anon_array_enter(struct anon_map *, ulong_t,
 441                         anon_sync_obj_t *);
 442 extern int      anon_array_try_enter(struct anon_map *, ulong_t,
 443                         anon_sync_obj_t *);
 442 extern void     anon_array_exit(anon_sync_obj_t *);

 444 /*
 445  * anon_resv checks to see if there is enough swap space to fulfill a
 446  * request and if so, reserves the appropriate anonymous memory resources.
 447  * anon_checkspace just checks to see if there is space to fulfill the request,
 448  * without taking any resources.  Both return 1 if successful and 0 if not.
 449  *
 450  * Macros are provided as anon reservation is usually charged to the zone of
 451  * the current process.  In some cases (such as anon reserved by tmpfs), a
 452  * zone pointer is needed to charge the appropriate zone.
 453  */
 454 #define anon_unresv(size)               anon_unresvmem(size, curproc->p_zone)
 455 #define anon_unresv_zone(size, zone)    anon_unresvmem(size, zone)
 456 #define anon_resv(size)                 \
 457         anon_resvmem((size), 1, curproc->p_zone, 1)
 458 #define anon_resv_zone(size, zone)      anon_resvmem((size), 1, zone, 1)
 459 #define anon_checkspace(size, zone)     anon_resvmem((size), 0, zone, 0)
 460 #define anon_try_resv_zone(size, zone)  anon_resvmem((size), 1, zone, 0)

 462 /*
 463  * Flags to anon_private
 464  */
 465 #define STEAL_PAGE      0x1     /* page can be stolen */
 466 #define LOCK_PAGE       0x2     /* page must be ''logically'' locked */

 468 /*
 469  * SEGKP ANON pages that are locked are assumed to be LWP stack pages
 470  * and thus count towards the user pages locked count.
 471  * This value is protected by the same lock as availrmem.
 472  */
 473 extern pgcnt_t anon_segkp_pages_locked;

 475 extern int anon_debug;

 477 #ifdef ANON_DEBUG

 479 #define A_ANON  0x01
 480 #define A_RESV  0x02
 481 #define A_MRESV 0x04

 483 /* vararg-like debugging macro. */
 484 #define ANON_PRINT(f, printf_args) \
 485                 if (anon_debug & f) \
 486                         printf printf_args

 488 #else   /* ANON_DEBUG */

 490 #define ANON_PRINT(f, printf_args)

 492 #endif  /* ANON_DEBUG */

 494 #endif  /* _KERNEL */

 496 #ifdef  __cplusplus
```

```
 497 }
______unchanged_portion_omitted_
```

```
**********************************************************
   11476 Fri May  8 18:10:34 2015
new/usr/src/uts/common/vm/as.h
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
remove xhat
The xhat infrastructure was added to support hardware such as the zulu
graphics card - hardware which had on-board MMUs.  The VM used the xhat code
to keep the CPU's and Zulu's page tables in-sync.  Since the only xhat user
was zulu (which is gone), we can safely remove it simplifying the whole VM
subsystem.
Assorted notes:
- AS_BUSY flag was used solely by xhat
**********************************************************
```

```
  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 /*
 27  * Copyright (c) 2013, Joyent, Inc.  All rights reserved.
 28  */

 30 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
 31 /*       All Rights Reserved   */

 33 /*
 34  * University Copyright- Copyright (c) 1982, 1986, 1988
 35  * The Regents of the University of California
 36  * All Rights Reserved
 37  *
 38  * University Acknowledgment- Portions of this document are derived from
 39  * software developed by the University of California, Berkeley, and its
 40  * contributors.
 41  */

 43 #ifndef _VM_AS_H
 44 #define _VM_AS_H

 46 #include <sys/watchpoint.h>
 47 #include <vm/seg.h>
```



```
 48 #include <vm/faultcode.h>
 49 #include <vm/hat.h>
 50 #include <sys/avl.h>
 51 #include <sys/proc.h>

 53 #ifdef __cplusplus
 54 extern "C" {
 55 #endif

 57 /*
 58  * VM - Address spaces.
 59  */

 61 /*
 62  * Each address space consists of a sorted list of segments
 63  * and machine dependent address translation information.
 64  *
 65  * All the hard work is in the segment drivers and the
 66  * hardware address translation code.
 67  *
 68  * The segment list is represented as an AVL tree.
 69  *
 70  * The address space lock (a_lock) is a long term lock which serializes
 71  * access to certain operations (as_map, as_unmap) and protects the
 72  * underlying generic segment data (seg.h) along with some fields in the
 73  * address space structure as shown below:
 74  *
 75  *      address space structure         segment structure
 76  *
 77  *      a_segtree                       s_base
 78  *      a_size                          s_size
 79  *      a_lastgap                       s_link
 80  *      a_seglast                       s_ops
 81  *                                      s_as
 82  *                                      s_data
 83  *
 84  * The address space contents lock (a_contents) is a short term
 85  * lock that protects most of the data in the address space structure.
 86  * This lock is always acquired after the "a_lock" in all situations
 87  * except while dealing with AS_CLAIMGAP to avoid deadlocks.
 88  *
 89  * The following fields are protected by this lock:
 90  *
 91  *      a_flags (AS_PAGLCK, AS_CLAIMGAP, etc.)
 92  *      a_unmapwait
 93  *      a_seglast
 94  *
 95  * The address space lock (a_lock) is always held prior to any segment
 96  * operation.  Some segment drivers use the address space lock to protect
 97  * some or all of their segment private data, provided the version of
 98  * "a_lock" (read vs. write) is consistent with the use of the data.
 99  *
100  * The following fields are protected by the hat layer lock:
101  *
102  *      a_vbits
103  *      a_hat
104  *      a_hrm
105  */

107 struct as {
108         kmutex_t a_contents;    /* protect certain fields in the structure */
109         uchar_t  a_flags;       /* as attributes */
110         uchar_t a_vbits;        /* used for collecting statistics */
111         kcondvar_t a_cv;        /* used by as_rangelock */
112         struct  hat *a_hat;     /* hat structure */
113         struct  hrmstat *a_hrm; /* ref and mod bits */
```

```
114         caddr_t a_userlimit;    /* highest allowable address in this as */
115         struct seg *a_seglast;  /* last segment hit on the addr space */
116         krwlock_t a_lock;       /* protects segment related fields */
117         size_t  a_size;         /* total size of address space */
118         struct seg *a_lastgap;  /* last seg found by as_gap() w/ AS_HI (mmap) */
119         struct seg *a_lastgaphl; /* last seg saved in as_gap() either for */
120                                 /* AS_HI or AS_LO used in as_addseg() */
121         avl_tree_t a_segtree;   /* segments in this address space. (AVL tree) */
122         avl_tree_t a_wpage;     /* watched pages (procfs) */
123         uchar_t a_updatedir;    /* mappings changed, rebuild a_objectdir */
124         timespec_t a_updatetime;        /* time when mappings last changed */
125         vnode_t **a_objectdir;  /* object directory (procfs) */
126         size_t  a_sizedir;      /* size of object directory */
127         struct as_callback *a_callbacks; /* callback list */
128         void *a_xhat;           /* list of xhat providers */
128         proc_t  *a_proc;        /* back pointer to proc */
129         size_t  a_resvsize;     /* size of reserved part of address space */
130 };

132 #define AS_PAGLCK               0x80
133 #define AS_CLAIMGAP             0x40
134 #define AS_UNMAPWAIT            0x20
135 #define AS_NEEDSPURGE           0x10    /* mostly for seg_nf, see as_purge() */
136 #define AS_NOUNMAPWAIT          0x02
138 #define AS_BUSY                 0x01    /* needed by XHAT framework */

138 #define AS_ISPGLCK(as)          ((as)->a_flags & AS_PAGLCK)
139 #define AS_ISCLAIMGAP(as)       ((as)->a_flags & AS_CLAIMGAP)
140 #define AS_ISUNMAPWAIT(as)      ((as)->a_flags & AS_UNMAPWAIT)
143 #define AS_ISBUSY(as)           ((as)->a_flags & AS_BUSY)
141 #define AS_ISNOUNMAPWAIT(as)    ((as)->a_flags & AS_NOUNMAPWAIT)

143 #define AS_SETPGLCK(as)         ((as)->a_flags |= AS_PAGLCK)
144 #define AS_SETCLAIMGAP(as)      ((as)->a_flags |= AS_CLAIMGAP)
145 #define AS_SETUNMAPWAIT(as)     ((as)->a_flags |= AS_UNMAPWAIT)
149 #define AS_SETBUSY(as)          ((as)->a_flags |= AS_BUSY)
146 #define AS_SETNOUNMAPWAIT(as)   ((as)->a_flags |= AS_NOUNMAPWAIT)

148 #define AS_CLRPGLCK(as)         ((as)->a_flags &= ~AS_PAGLCK)
149 #define AS_CLRCLAIMGAP(as)      ((as)->a_flags &= ~AS_CLAIMGAP)
150 #define AS_CLRUNMAPWAIT(as)     ((as)->a_flags &= ~AS_UNMAPWAIT)
155 #define AS_CLRBUSY(as)          ((as)->a_flags &= ~AS_BUSY)
151 #define AS_CLRNOUNMAPWAIT(as)   ((as)->a_flags &= ~AS_NOUNMAPWAIT)

153 #define AS_TYPE_64BIT(as)       \
154             (((as)->a_userlimit > (caddr_t)UINT32_MAX) ? 1 : 0)

156 /*
157  * Flags for as_map/as_map_ansegs
158  */
159 #define AS_MAP_NO_LPOOB         ((uint_t)-1)
160 #define AS_MAP_HEAP             ((uint_t)-2)
161 #define AS_MAP_STACK            ((uint_t)-3)

163 /*
164  * The as_callback is the basic structure which supports the ability to
165  * inform clients of specific events pertaining to address space management.
166  * A user calls as_add_callback to register an address space callback
167  * for a range of pages, specifying the events that need to occur.
168  * When as_do_callbacks is called and finds a 'matching' entry, the
169  * callback is called once, and the callback function MUST call
170  * as_delete_callback when all callback activities are complete.
171  * The thread calling as_do_callbacks blocks until the as_delete_callback
172  * is called.  This allows for asynchorous events to subside before the
173  * as_do_callbacks thread continues.
174  *
```



```
175  * An example of the need for this is a driver which has done long-term
176  * locking of memory.  Address space management operations (events) such
177  * as as_free, as_umap, and as_setprot will block indefinitely until the
178  * pertinent memory is unlocked.  The callback mechanism provides the
179  * way to inform the driver of the event so that the driver may do the
180  * necessary unlocking.
181  *
182  * The contents of this structure is protected by a_contents lock
183  */
184 typedef void (*callback_func_t)(struct as *, void *, uint_t);
185 struct as_callback {
186         struct as_callback      *ascb_next;             /* list link */
187         uint_t                  ascb_events;            /* event types */
188         callback_func_t         ascb_func;              /* callback function */
189         void                    *ascb_arg;              /* callback argument */
190         caddr_t                 ascb_saddr;             /* start address */
191         size_t                  ascb_len;               /* address range */
192 };
_____unchanged_portion_omitted_

213 #ifdef _KERNEL

215 /*
216  * Flags for as_gap.
217  */
218 #define AH_DIR          0x1     /* direction flag mask */
219 #define AH_LO           0x0     /* find lowest hole */
220 #define AH_HI           0x1     /* find highest hole */
221 #define AH_CONTAIN      0x2     /* hole must contain 'addr' */

223 extern struct as kas;           /* kernel's address space */

225 /*
226  * Macros for address space locking.  Note that we use RW_READER_STARVEWRITER
227  * whenever we acquire the address space lock as reader to assure that it can
228  * be used without regard to lock order in conjunction with filesystem locks.
229  * This allows filesystems to safely induce user-level page faults with
230  * filesystem locks held while concurrently allowing filesystem entry points
231  * acquiring those same locks to be called with the address space lock held as
232  * reader.  RW_READER_STARVEWRITER thus prevents reader/reader+RW_WRITE_WANTED
233  * deadlocks in the style of fop_write()+as_fault()/as_*()+fop_putpage() and
234  * fop_read()+as_fault()/as_*()+fop_getpage().  (See the Big Theory Statement
235  * in rwlock.c for more information on the semantics of and motivation behind
236  * RW_READER_STARVEWRITER.)
237  */
238 #define AS_LOCK_ENTER(as, lock, type)           rw_enter((lock), \
239         (type) == RW_READER ? RW_READER_STARVEWRITER : (type))
240 #define AS_LOCK_EXIT(as, lock)                  rw_exit((lock))
241 #define AS_LOCK_DESTROY(as, lock)               rw_destroy((lock))
242 #define AS_LOCK_TRYENTER(as, lock, type)        rw_tryenter((lock), \
243         (type) == RW_READER ? RW_READER_STARVEWRITER : (type))

245 /*
246  * Macros to test lock states.
247  */
248 #define AS_LOCK_HELD(as, lock)          RW_LOCK_HELD((lock))
249 #define AS_READ_HELD(as, lock)          RW_READ_HELD((lock))
250 #define AS_WRITE_HELD(as, lock)         RW_WRITE_HELD((lock))

252 /*
253  * macros to walk thru segment lists
254  */
255 #define AS_SEGFIRST(as)         avl_first(&(as)->a_segtree)
256 #define AS_SEGNEXT(as, seg)     AVL_NEXT(&(as)->a_segtree, (seg))
257 #define AS_SEGPREV(as, seg)     AVL_PREV(&(as)->a_segtree, (seg))
```

```
259 void    as_init(void);
260 void    as_avlinit(struct as *);
261 struct  seg *as_segat(struct as *as, caddr_t addr);
262 void    as_rangelock(struct as *as);
263 void    as_rangeunlock(struct as *as);
264 struct  as *as_alloc();
265 void    as_free(struct as *as);
266 int     as_dup(struct as *as, struct proc *forkedproc);
267 struct  seg *as_findseg(struct as *as, caddr_t addr, int tail);
268 int     as_addseg(struct as *as, struct seg *newseg);
269 struct  seg *as_removeseg(struct as *as, struct seg *seg);
270 faultcode_t as_fault(struct hat *hat, struct as *as, caddr_t addr, size_t size,
271                 enum fault_type type, enum seg_rw rw);
272 faultcode_t as_faulta(struct as *as, caddr_t addr, size_t size);
273 int     as_setprot(struct as *as, caddr_t addr, size_t size, uint_t prot);
274 int     as_checkprot(struct as *as, caddr_t addr, size_t size, uint_t prot);
275 int     as_unmap(struct as *as, caddr_t addr, size_t size);
276 int     as_map(struct as *as, caddr_t addr, size_t size, int ((*crfp)()),
277                 void *argsp);
278 void    as_purge(struct as *as);
279 int     as_gap(struct as *as, size_t minlen, caddr_t *basep, size_t *lenp,
280                 uint_t flags, caddr_t addr);
281 int     as_gap_aligned(struct as *as, size_t minlen, caddr_t *basep,
282             size_t *lenp, uint_t flags, caddr_t addr, size_t align,
283             size_t redzone, size_t off);

285 int     as_memory(struct as *as, caddr_t *basep, size_t *lenp);
291 size_t  as_swapout(struct as *as);
286 int     as_incore(struct as *as, caddr_t addr, size_t size, char *vec,
287                 size_t *sizep);
288 int     as_ctl(struct as *as, caddr_t addr, size_t size, int func, int attr,
289                 uintptr_t arg, ulong_t *lock_map, size_t pos);
290 int     as_pagelock(struct as *as, struct page ***ppp, caddr_t addr,
291                 size_t size, enum seg_rw rw);
292 void    as_pageunlock(struct as *as, struct page **pp, caddr_t addr,
293                 size_t size, enum seg_rw rw);
294 int     as_setpagesize(struct as *as, caddr_t addr, size_t size, uint_t szc,
295                 boolean_t wait);
296 int     as_set_default_lpsize(struct as *as, caddr_t addr, size_t size);
297 void    as_setwatch(struct as *as);
298 void    as_clearwatch(struct as *as);
299 int     as_getmemid(struct as *, caddr_t, memid_t *);

301 int     as_add_callback(struct as *, void (*)(), void *, uint_t,
302                     caddr_t, size_t, int);
303 uint_t  as_delete_callback(struct as *, void *);

305 #endif  /* _KERNEL */

307 #ifdef  __cplusplus
308 }
```

unchanged_portion_omitted_

```
**********************************************************
   19654 Fri May  8 18:10:34 2015
new/usr/src/uts/common/vm/hat.h
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

  81 typedef void *hat_region_cookie_t;

  83 #ifdef  _KERNEL

  85 /*
  86  * One time hat initialization
  87  */
  88 void    hat_init(void);

  90 /*
  91  * Notify hat of a system dump
  92  */
  93 void    hat_dump(void);

  95 /*
  96  * Operations on an address space:
  97  *
  98  * struct hat *hat_alloc(as)
  99  *      allocated a hat structure for as.
 100  *
 101  * void hat_free_start(hat)
 102  *      informs hat layer process has finished executing but as has not
 103  *      been cleaned up yet.
 104  *
 105  * void hat_free_end(hat)
 106  *      informs hat layer as is being destroyed.  hat layer cannot use as
 107  *      pointer after this call.
 108  *
 109  * void hat_swapin(hat)
 110  *      allocate any hat resources required for process being swapped in.
 111  *
 112  * void hat_swapout(hat)
 113  *      deallocate hat resources for process being swapped out.
 114  *
 109  * size_t hat_get_mapped_size(hat)
 110  *      returns number of bytes that have valid mappings in hat.
 111  *
 112  * void hat_stats_enable(hat)
 113  * void hat_stats_disable(hat)
 114  *      enables/disables collection of stats for hat.
 115  *
 116  * int hat_dup(parenthat, childhat, addr, len, flags)
 117  *      Duplicate address translations of the parent to the child.  Supports
 118  *      the entire address range or a range depending on flag,
 119  *      zero returned on success, non-zero on error
 120  *
 121  * void hat_thread_exit(thread)
 122  *      Notifies the HAT that a thread is exiting, called after it has been
 123  *      reassigned to the kernel AS.
 124  */

 126 struct hat *hat_alloc(struct as *);
 127 void    hat_free_start(struct hat *);
```



```
 128 void    hat_free_end(struct hat *);
 129 int     hat_dup(struct hat *, struct hat *, caddr_t, size_t, uint_t);
 136 void    hat_swapin(struct hat *);
 137 void    hat_swapout(struct hat *);
 130 size_t  hat_get_mapped_size(struct hat *);
 131 int     hat_stats_enable(struct hat *);
 132 void    hat_stats_disable(struct hat *);
 133 void    hat_thread_exit(kthread_t *);

 135 /*
 136  * Operations on a named address within a segment:
 137  *
 138  * void hat_memload(hat, addr, pp, attr, flags)
 139  *      load/lock the given page struct
 140  *
 141  * void hat_memload_array(hat, addr, len, ppa, attr, flags)
 142  *      load/lock the given array of page structs
 143  *
 144  * void hat_devload(hat, addr, len, pf, attr, flags)
 145  *      load/lock the given page frame number
 146  *
 147  * void hat_unlock(hat, addr, len)
 148  *      unlock a given range of addresses
 149  *
 150  * void hat_unload(hat, addr, len, flags)
 151  * void hat_unload_callback(hat, addr, len, flags, callback)
 152  *      unload a given range of addresses (has optional callback)
 153  *
 154  * void hat_sync(hat, addr, len, flags)
 155  *      synchronize mapping with software data structures
 156  *
 157  * void hat_map(hat, addr, len, flags)
 158  *
 159  * void hat_setattr(hat, addr, len, attr)
 160  * void hat_clrattr(hat, addr, len, attr)
 161  * void hat_chgattr(hat, addr, len, attr)
 162  *      modify attributes for a range of addresses. skips any invalid mappings
 163  *
 164  * uint_t hat_getattr(hat, addr, *attr)
 165  *      returns attr for <hat,addr> in *attr.  returns 0 if there was a
 166  *      mapping and *attr is valid, nonzero if there was no mapping and
 167  *      *attr is not valid.
 168  *
 169  * size_t hat_getpagesize(hat, addr)
 170  *      returns pagesize in bytes for <hat, addr>. returns -1 if there is
 171  *      no mapping. This is an advisory call.
 172  *
 173  * pfn_t hat_getpfnum(hat, addr)
 174  *      returns pfn for <hat, addr> or PFN_INVALID if mapping is invalid.
 175  *
 176  * int hat_probe(hat, addr)
 177  *      return 0 if no valid mapping is present.  Faster version
 178  *      of hat_getattr in certain architectures.
 179  *
 180  * int hat_share(dhat, daddr, shat, saddr, len, szc)
 181  *
 182  * void hat_unshare(hat, addr, len, szc)
 183  *
 184  * void hat_chgprot(hat, addr, len, vprot)
 185  *      This is a deprecated call.  New segment drivers should store
 186  *      all attributes and use hat_*attr calls.
 187  *      Change the protections in the virtual address range
 188  *      given to the specified virtual protection.  If vprot is ~PROT_WRITE,
 189  *      then remove write permission, leaving the other permissions
 190  *      unchanged.  If vprot is ~PROT_USER, remove user permissions.
 191  *
```

```
 192  * void hat_flush_range(hat, addr, size)
 193  *      Invalidate a virtual address translation for the local CPU.
 194  */

 196 void    hat_memload(struct hat *, caddr_t, struct page *, uint_t, uint_t);
 197 void    hat_memload_array(struct hat *, caddr_t, size_t, struct page **,
 198                 uint_t, uint_t);
 199 void    hat_memload_region(struct hat *, caddr_t, struct page *, uint_t,
 200                 uint_t, hat_region_cookie_t);
 201 void    hat_memload_array_region(struct hat *, caddr_t, size_t, struct page **,
 202                 uint_t, uint_t, hat_region_cookie_t);

 204 void    hat_devload(struct hat *, caddr_t, size_t, pfn_t, uint_t, int);

 206 void    hat_unlock(struct hat *, caddr_t, size_t);
 207 void    hat_unlock_region(struct hat *, caddr_t, size_t, hat_region_cookie_t);

 209 void    hat_unload(struct hat *, caddr_t, size_t, uint_t);
 210 void    hat_unload_callback(struct hat *, caddr_t, size_t, uint_t,
 211                 hat_callback_t *);
 212 void    hat_flush_range(struct hat *, caddr_t, size_t);
 213 void    hat_sync(struct hat *, caddr_t, size_t, uint_t);
 214 void    hat_map(struct hat *, caddr_t, size_t, uint_t);
 215 void    hat_setattr(struct hat *, caddr_t, size_t, uint_t);
 216 void    hat_clrattr(struct hat *, caddr_t, size_t, uint_t);
 217 void    hat_chgattr(struct hat *, caddr_t, size_t, uint_t);
 218 uint_t  hat_getattr(struct hat *, caddr_t, uint_t *);
 219 ssize_t hat_getpagesize(struct hat *, caddr_t);
 220 pfn_t   hat_getpfnum(struct hat *, caddr_t);
 221 int     hat_probe(struct hat *, caddr_t);
 222 int     hat_share(struct hat *, caddr_t, struct hat *, caddr_t, size_t, uint_t);
 223 void    hat_unshare(struct hat *, caddr_t, size_t, uint_t);
 224 void    hat_chgprot(struct hat *, caddr_t, size_t, uint_t);
 225 void    hat_reserve(struct as *, caddr_t, size_t);
 226 pfn_t   va_to_pfn(void *);
 227 uint64_t va_to_pa(void *);

 229 /*
 230  * Kernel Physical Mapping (segkpm) hat interface routines.
 231  */
 232 caddr_t hat_kpm_mapin(struct page *, struct kpme *);
 233 void    hat_kpm_mapout(struct page *, struct kpme *, caddr_t);
 234 caddr_t hat_kpm_mapin_pfn(pfn_t);
 235 void    hat_kpm_mapout_pfn(pfn_t);
 236 caddr_t hat_kpm_page2va(struct page *, int);
 237 struct page *hat_kpm_vaddr2page(caddr_t);
 238 int     hat_kpm_fault(struct hat *, caddr_t);
 239 void    hat_kpm_mseghash_clear(int);
 240 void    hat_kpm_mseghash_update(pgcnt_t, struct memseg *);
 241 void    hat_kpm_addmem_mseg_update(struct memseg *, pgcnt_t, offset_t);
 242 void    hat_kpm_addmem_mseg_insert(struct memseg *);
 243 void    hat_kpm_addmem_memsegs_update(struct memseg *);
 244 caddr_t hat_kpm_mseg_reuse(struct memseg *);
 245 void    hat_kpm_delmem_mseg_update(struct memseg *, struct memseg **);
 246 void    hat_kpm_split_mseg_update(struct memseg *, struct memseg **,
 247                         struct memseg *, struct memseg *, struct memseg *);
 248 void    hat_kpm_walk(void (*)(void *, void *, size_t), void *);

 250 /*
 251  * Operations on all translations for a given page(s)
 252  *
 253  * void hat_page_setattr(pp, flag)
 254  * void hat_page_clrattr(pp, flag)
 255  *      used to set/clr red/mod bits.
 256  *
 257  * uint hat_page_getattr(pp, flag)
```



```
 258  *      If flag is specified, returns 0 if attribute is disabled
 259  *      and non zero if enabled.  If flag specifes multiple attributs
 260  *      then returns 0 if ALL atriibutes are disabled.  This is an advisory
 261  *      call.
 262  *
 263  * int hat_pageunload(pp, forceflag)
 264  *      unload all translations attached to pp.
 265  *
 266  * uint_t hat_pagesync(pp, flags)
 267  *      get hw stats from hardware into page struct and reset hw stats
 268  *      returns attributes of page
 269  *
 270  * ulong_t hat_page_getshare(pp)
 271  *      returns approx number of mappings to this pp.  A return of 0 implies
 272  *      there are no mappings to the page.
 273  *
 274  * faultcode_t hat_softlock(hat, addr, lenp, ppp, flags);
 275  *      called to softlock pages for zero copy tcp
 276  *
 277  * void hat_page_demote(pp);
 278  *      unload all large mappings to pp and decrease p_szc of all
 279  *      constituent pages according to the remaining mappings.
 280  */

 282 void    hat_page_setattr(struct page *, uint_t);
 283 void    hat_page_clrattr(struct page *, uint_t);
 284 uint_t  hat_page_getattr(struct page *, uint_t);
 285 int     hat_pageunload(struct page *, uint_t);
 286 uint_t  hat_pagesync(struct page *, uint_t);
 287 ulong_t hat_page_getshare(struct page *);
 288 int     hat_page_checkshare(struct page *, ulong_t);
 289 faultcode_t hat_softlock(struct hat *, caddr_t, size_t *,
 290                         struct page **, uint_t);
 291 void    hat_page_demote(struct page *);

 293 /*
 294  * Rountine to expose supported HAT features to PIM.
 295  */
 296 enum hat_features {
 297         HAT_SHARED_PT,          /* Shared page tables */
 298         HAT_DYNAMIC_ISM_UNMAP,  /* hat_pageunload() handles ISM pages */
 299         HAT_VMODSORT,           /* support for VMODSORT flag of vnode */
 300         HAT_SHARED_REGIONS      /* shared regions support */
 301 };
```

unchanged_portion_omitted_

```
**********************************************************
    9948 Fri May  8 18:10:34 2015
new/usr/src/uts/common/vm/seg.h
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
seg_inherit_notsup is redundant since segop_inherit checks for NULL properly
patch lower-case-segops
instead using SEGOP_* macros, define full-fledged segop_* functions
This will allow us to do some sanity checking or even implement stub
functionality in one place instead of duplicating it wherever these wrappers
are used.
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

 102 typedef struct seg {
 103         caddr_t s_base;               /* base virtual address */
 104         size_t  s_size;               /* size in bytes */
 105         uint_t  s_szc;                /* max page size code */
 106         uint_t  s_flags;              /* flags for segment, see below */
 107         struct  as *s_as;             /* containing address space */
 108         avl_node_t s_tree;            /* AVL tree links to segs in this as */
 109         const struct seg_ops *s_ops;  /* ops vector: see below */
 109         struct  seg_ops *s_ops;       /* ops vector: see below */
 110         void *s_data;                 /* private data for instance */
 111         kmutex_t s_pmtx;              /* protects seg's pcache list */
 112         pcache_link_t s_phead;        /* head of seg's pcache list */
 113 } seg_t;

 115 #define S_PURGE         (0x01)        /* seg should be purged in as_gap() */

 117 struct  seg_ops {
 118         int     (*dup)(struct seg *, struct seg *);
 119         int     (*unmap)(struct seg *, caddr_t, size_t);
 120         void    (*free)(struct seg *);
 121         faultcode_t (*fault)(struct hat *, struct seg *, caddr_t, size_t,
 122             enum fault_type, enum seg_rw);
 123         faultcode_t (*faulta)(struct seg *, caddr_t);
 124         int     (*setprot)(struct seg *, caddr_t, size_t, uint_t);
 125         int     (*checkprot)(struct seg *, caddr_t, size_t, uint_t);
 126         int     (*kluster)(struct seg *, caddr_t, ssize_t);
 127         size_t  (*swapout)(struct seg *);
 127         int     (*sync)(struct seg *, caddr_t, size_t, int, uint_t);
 128         size_t  (*incore)(struct seg *, caddr_t, size_t, char *);
 129         int     (*lockop)(struct seg *, caddr_t, size_t, int, int, ulong_t *,
 130                         size_t);
 131         int     (*getprot)(struct seg *, caddr_t, size_t, uint_t *);
 132         u_offset_t      (*getoffset)(struct seg *, caddr_t);
 133         int     (*gettype)(struct seg *, caddr_t);
 134         int     (*getvp)(struct seg *, caddr_t, struct vnode **);
 135         int     (*advise)(struct seg *, caddr_t, size_t, uint_t);
 136         void    (*dump)(struct seg *);
 137         int     (*pagelock)(struct seg *, caddr_t, size_t, struct page ***,
 138                         enum lock_type, enum seg_rw);
 139         int     (*setpagesize)(struct seg *, caddr_t, size_t, uint_t);
 140         int     (*getmemid)(struct seg *, caddr_t, memid_t *);
 141         struct lgrp_mem_policy_info     *(*getpolicy)(struct seg *, caddr_t);
 142         int     (*capable)(struct seg *, segcapability_t);
 143         int     (*inherit)(struct seg *, caddr_t, size_t, uint_t);
 144 };
```



```
 146 #ifdef _KERNEL

 148 /*
 149  * Generic segment operations
 150  */
 151 extern  void    seg_init(void);
 152 extern  struct  seg *seg_alloc(struct as *as, caddr_t base, size_t size);
 153 extern  int     seg_attach(struct as *as, caddr_t base, size_t size,
 154                         struct seg *seg);
 155 extern  void    seg_unmap(struct seg *seg);
 156 extern  void    seg_free(struct seg *seg);

 158 /*
 159  * functions for pagelock cache support
 160  */
 161 typedef int (*seg_preclaim_cbfunc_t)(void *, caddr_t, size_t,
 162     struct page **, enum seg_rw, int);

 164 extern  struct  page **seg_plookup(struct seg *seg, struct anon_map *amp,
 165     caddr_t addr, size_t len, enum seg_rw rw, uint_t flags);
 166 extern  void    seg_pinactive(struct seg *seg, struct anon_map *amp,
 167     caddr_t addr, size_t len, struct page **pp, enum seg_rw rw,
 168     uint_t flags, seg_preclaim_cbfunc_t callback);

 170 extern  void    seg_ppurge(struct seg *seg, struct anon_map *amp,
 171     uint_t flags);
 172 extern  void    seg_ppurge_wiredpp(struct page **pp);

 174 extern  int     seg_pinsert_check(struct seg *seg, struct anon_map *amp,
 175     caddr_t addr, size_t len, uint_t flags);
 176 extern  int     seg_pinsert(struct seg *seg, struct anon_map *amp,
 177     caddr_t addr, size_t len, size_t wlen, struct page **pp, enum seg_rw rw,
 178     uint_t flags, seg_preclaim_cbfunc_t callback);

 180 extern  void    seg_pasync_thread(void);
 181 extern  void    seg_preap(void);
 182 extern  int     seg_p_disable(void);
 183 extern  void    seg_p_enable(void);

 185 extern  segadvstat_t    segadvstat;

 187 /*
 188  * Flags for pagelock cache support.
 189  * Flags argument is passed as uint_t to pcache routines.  upper 16 bits of
 190  * the flags argument are reserved for alignment page shift when SEGP_PSHIFT
 191  * is set.
 192  */
 193 #define SEGP_FORCE_WIRED        0x1     /* skip check against seg_pwindow */
 194 #define SEGP_AMP                0x2     /* anon map's pcache entry */
 195 #define SEGP_PSHIFT             0x4     /* addr pgsz shift for hash function */

 197 /*
 198  * Return values for seg_pinsert and seg_pinsert_check functions.
 199  */
 200 #define SEGP_SUCCESS            0       /* seg_pinsert() succeeded */
 201 #define SEGP_FAIL               1       /* seg_pinsert() failed */

 203 /* Page status bits for segop_incore */
 204 #define SEG_PAGE_INCORE         0x01    /* VA has a page backing it */
 205 #define SEG_PAGE_LOCKED         0x02    /* VA has a page that is locked */
 206 #define SEG_PAGE_HASCOW         0x04    /* VA has a page with a copy-on-write */
 207 #define SEG_PAGE_SOFTLOCK       0x08    /* VA has a page with softlock held */
 208 #define SEG_PAGE_VNODEBACKED    0x10    /* Segment is backed by a vnode */
 209 #define SEG_PAGE_ANON           0x20    /* VA has an anonymous page */
 210 #define SEG_PAGE_VNODE          0x40    /* VA has a vnode page backing it */
```

```
#define SEGOP_DUP(s, n)             (*(s)->s_ops->dup)((s), (n))
#define SEGOP_UNMAP(s, a, l)        (*(s)->s_ops->unmap)((s), (a), (l))
#define SEGOP_FREE(s)               (*(s)->s_ops->free)((s))
#define SEGOP_FAULT(h, s, a, l, t, rw) \
                (*(s)->s_ops->fault)((h), (s), (a), (l), (t), (rw))
#define SEGOP_FAULTA(s, a)          (*(s)->s_ops->faulta)((s), (a))
#define SEGOP_SETPROT(s, a, l, p)   (*(s)->s_ops->setprot)((s), (a), (l), (p))
#define SEGOP_CHECKPROT(s, a, l, p) (*(s)->s_ops->checkprot)((s), (a), (l), (p))
#define SEGOP_KLUSTER(s, a, d)      (*(s)->s_ops->kluster)((s), (a), (d))
#define SEGOP_SWAPOUT(s)            (*(s)->s_ops->swapout)((s))
#define SEGOP_SYNC(s, a, l, atr, f) \
                (*(s)->s_ops->sync)((s), (a), (l), (atr), (f))
#define SEGOP_INCORE(s, a, l, v)    (*(s)->s_ops->incore)((s), (a), (l), (v))
#define SEGOP_LOCKOP(s, a, l, atr, op, b, p) \
                (*(s)->s_ops->lockop)((s), (a), (l), (atr), (op), (b), (p))
#define SEGOP_GETPROT(s, a, l, p)   (*(s)->s_ops->getprot)((s), (a), (l), (p))
#define SEGOP_GETOFFSET(s, a)       (*(s)->s_ops->getoffset)((s), (a))
#define SEGOP_GETTYPE(s, a)         (*(s)->s_ops->gettype)((s), (a))
#define SEGOP_GETVP(s, a, vpp)      (*(s)->s_ops->getvp)((s), (a), (vpp))
#define SEGOP_ADVISE(s, a, l, b)    (*(s)->s_ops->advise)((s), (a), (l), (b))
#define SEGOP_DUMP(s)               (*(s)->s_ops->dump)((s))
#define SEGOP_PAGELOCK(s, a, l, p, t, rw) \
                (*(s)->s_ops->pagelock)((s), (a), (l), (p), (t), (rw))
#define SEGOP_SETPAGESIZE(s, a, l, szc) \
                (*(s)->s_ops->setpagesize)((s), (a), (l), (szc))
#define SEGOP_GETMEMID(s, a, mp)    (*(s)->s_ops->getmemid)((s), (a), (mp))
#define SEGOP_GETPOLICY(s, a)       (*(s)->s_ops->getpolicy)((s), (a))
#define SEGOP_CAPABLE(s, c)         (*(s)->s_ops->capable)((s), (c))
#define SEGOP_INHERIT(s, a, l, b)   (*(s)->s_ops->inherit)((s), (a), (l), (b))

#define seg_page(seg, addr) \
        (((uintptr_t)((addr) - (seg)->s_base)) >> PAGESHIFT)

#define seg_pages(seg) \
        (((uintptr_t)((seg)->s_size + PAGEOFFSET)) >> PAGESHIFT)

#define IE_NOMEM        -1      /* internal to seg layer */
#define IE_RETRY        -2      /* internal to seg layer */
#define IE_REATTACH     -3      /* internal to seg layer */

/* Values for segop_inherit */
/* Values for SEGOP_INHERIT */
#define SEGP_INH_ZERO   0x01

int seg_inherit_notsup(struct seg *, caddr_t, size_t, uint_t);

/* Delay/retry factors for seg_p_mem_config_pre_del */
#define SEGP_PREDEL_DELAY_FACTOR        4
/*
 * As a workaround to being unable to purge the pagelock
 * cache during a DR delete memory operation, we use
 * a stall threshold that is twice the maximum seen
 * during testing.  This workaround will be removed
 * when a suitable fix is found.
 */
#define SEGP_STALL_SECONDS      25
#define SEGP_STALL_THRESHOLD \
        (SEGP_STALL_SECONDS * SEGP_PREDEL_DELAY_FACTOR)

#ifdef VMDEBUG

uint_t  seg_page(struct seg *, caddr_t);
uint_t  seg_pages(struct seg *);

#endif  /* VMDEBUG */
```



```
boolean_t       seg_can_change_zones(struct seg *);
size_t          seg_swresv(struct seg *);

/* segop wrappers */
int segop_dup(struct seg *, struct seg *);
int segop_unmap(struct seg *, caddr_t, size_t);
void segop_free(struct seg *);
faultcode_t segop_fault(struct hat *, struct seg *, caddr_t, size_t, enum fault_
faultcode_t segop_faulta(struct seg *, caddr_t);
int segop_setprot(struct seg *, caddr_t, size_t, uint_t);
int segop_checkprot(struct seg *, caddr_t, size_t, uint_t);
int segop_kluster(struct seg *, caddr_t, ssize_t);
int segop_sync(struct seg *, caddr_t, size_t, int, uint_t);
size_t segop_incore(struct seg *, caddr_t, size_t, char *);
int segop_lockop(struct seg *, caddr_t, size_t, int, int, ulong_t *, size_t );
int segop_getprot(struct seg *, caddr_t, size_t, uint_t *);
u_offset_t segop_getoffset(struct seg *, caddr_t);
int segop_gettype(struct seg *, caddr_t);
int segop_getvp(struct seg *, caddr_t, struct vnode **);
int segop_advise(struct seg *, caddr_t, size_t, uint_t);
void segop_dump(struct seg *);
int segop_pagelock(struct seg *, caddr_t, size_t, struct page ***, enum lock_typ
int segop_setpagesize(struct seg *, caddr_t, size_t, uint_t);
int segop_getmemid(struct seg *, caddr_t, memid_t *);
struct lgrp_mem_policy_info *segop_getpolicy(struct seg *, caddr_t);
int segop_capable(struct seg *, segcapability_t);
int segop_inherit(struct seg *, caddr_t, size_t, uint_t);
#endif /* ! codereview */

#endif  /* _KERNEL */

#ifdef  __cplusplus
}
#endif

#endif  /* _VM_SEG_H */
```

```
**********************************************************
  113266 Fri May  8 18:10:34 2015
new/usr/src/uts/common/vm/seg_dev.c
use NULL dump segop as a shorthand for no-op
Instead of forcing every segment driver to implement a dummy function that
does nothing, handle NULL dump segop function pointer as a no-op shorthand.
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
use NULL setpagesize segop as a shorthand for ENOTSUP
Instead of forcing every segment driver to implement a dummp function to
return (hopefully) ENOTSUP, handle NULL setpagesize segop function pointer
as "return ENOTSUP" shorthand.
use NULL capable segop as a shorthand for no-capabilities
Instead of forcing every segment driver to implement a dummy "return 0"
function, handle NULL capable segop function pointer as "no copabilities
supported" shorthand.
segop_getpolicy already checks for a NULL op
seg_inherit_notsup is redundant since segop_inherit checks for NULL properly
patch lower-case-segops
use C99 initializers in segment ops structures
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
```

```
/*
 * 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.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */
```



```
/*
 * VM - segment of a mapped device.
 *
 * This segment driver is used when mapping character special devices.
 */

#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/sysmacros.h>
#include <sys/vtrace.h>
#include <sys/systm.h>
#include <sys/vmsystm.h>
#include <sys/mman.h>
#include <sys/errno.h>
#include <sys/kmem.h>
#include <sys/cmn_err.h>
#include <sys/vnode.h>
#include <sys/proc.h>
#include <sys/conf.h>
#include <sys/debug.h>
#include <sys/ddidevmap.h>
#include <sys/ddi_implfuncs.h>
#include <sys/lgrp.h>

#include <vm/page.h>
#include <vm/hat.h>
#include <vm/as.h>
#include <vm/seg.h>
#include <vm/seg_dev.h>
#include <vm/seg_kp.h>
#include <vm/seg_kmem.h>
#include <vm/vpage.h>

#include <sys/sunddi.h>
#include <sys/esunddi.h>
#include <sys/fs/snode.h>


#if DEBUG
int segdev_debug;
#define DEBUGF(level, args) { if (segdev_debug >= (level)) cmn_err args; }
#else
#define DEBUGF(level, args)
#endif

/* Default timeout for devmap context management */
#define CTX_TIMEOUT_VALUE 0

#define HOLD_DHP_LOCK(dhp)  if (dhp->dh_flags & DEVMAP_ALLOW_REMAP) \
                        { mutex_enter(&dhp->dh_lock); }

#define RELE_DHP_LOCK(dhp) if (dhp->dh_flags & DEVMAP_ALLOW_REMAP) \
                        { mutex_exit(&dhp->dh_lock); }

#define round_down_p2(a, s)     ((a) & ~((s) - 1))
#define round_up_p2(a, s)       (((a) + (s) - 1) & ~((s) - 1))

/*
 * VA_PA_ALIGNED checks to see if both VA and PA are on pgsize boundary
 * VA_PA_PGSIZE_ALIGNED check to see if VA is aligned with PA w.r.t. pgsize
 */
#define VA_PA_ALIGNED(uvaddr, paddr, pgsize)            \
        (((uvaddr | paddr) & (pgsize - 1)) == 0)
#define VA_PA_PGSIZE_ALIGNED(uvaddr, paddr, pgsize)     \
        (((uvaddr ^ paddr) & (pgsize - 1)) == 0)
```

```
106 #define vpgtob(n)       ((n) * sizeof (struct vpage))   /* For brevity */

108 #define VTOCVP(vp)      (VTOS(vp)->s_commonvp)  /* we "know" it's an snode */

110 static struct devmap_ctx *devmapctx_list = NULL;
111 static struct devmap_softlock *devmap_slist = NULL;

113 /*
114  * mutex, vnode and page for the page of zeros we use for the trash mappings.
115  * One trash page is allocated on the first ddi_umem_setup call that uses it
116  * XXX Eventually, we may want to combine this with what segnf does when all
117  * hat layers implement HAT_NOFAULT.
118  *
119  * The trash page is used when the backing store for a userland mapping is
120  * removed but the application semantics do not take kindly to a SIGBUS.
121  * In that scenario, the applications pages are mapped to some dummy page
122  * which returns garbage on read and writes go into a common place.
123  * (Perfect for NO_FAULT semantics)
124  * The device driver is responsible to communicating to the app with some
125  * other mechanism that such remapping has happened and the app should take
126  * corrective action.
127  * We can also use an anonymous memory page as there is no requirement to
128  * keep the page locked, however this complicates the fault code. RFE.
129  */
130 static struct vnode trashvp;
131 static struct page *trashpp;

133 /* Non-pageable kernel memory is allocated from the umem_np_arena. */
134 static vmem_t *umem_np_arena;

136 /* Set the cookie to a value we know will never be a valid umem_cookie */
137 #define DEVMAP_DEVMEM_COOKIE    ((ddi_umem_cookie_t)0x1)

139 /*
140  * Macros to check if type of devmap handle
141  */
142 #define cookie_is_devmem(c)     \
143         ((c) == (struct ddi_umem_cookie *)DEVMAP_DEVMEM_COOKIE)

145 #define cookie_is_pmem(c)       \
146         ((c) == (struct ddi_umem_cookie *)DEVMAP_PMEM_COOKIE)

148 #define cookie_is_kpmem(c)      (!cookie_is_devmem(c) && !cookie_is_pmem(c) &&\
149         ((c)->type == KMEM_PAGEABLE))

151 #define dhp_is_devmem(dhp)      \
152         (cookie_is_devmem((struct ddi_umem_cookie *)((dhp)->dh_cookie)))

154 #define dhp_is_pmem(dhp)        \
155         (cookie_is_pmem((struct ddi_umem_cookie *)((dhp)->dh_cookie)))

157 #define dhp_is_kpmem(dhp)       \
158         (cookie_is_kpmem((struct ddi_umem_cookie *)((dhp)->dh_cookie)))

160 /*
161  * Private seg op routines.
162  */
163 static int      segdev_dup(struct seg *, struct seg *);
164 static int      segdev_unmap(struct seg *, caddr_t, size_t);
165 static void     segdev_free(struct seg *);
166 static faultcode_t segdev_fault(struct hat *, struct seg *, caddr_t, size_t,
167                     enum fault_type, enum seg_rw);
168 static faultcode_t segdev_faulta(struct seg *, caddr_t);
169 static int      segdev_setprot(struct seg *, caddr_t, size_t, uint_t);
170 static int      segdev_checkprot(struct seg *, caddr_t, size_t, uint_t);
```



```
171 static void     segdev_badop(void);
172 static int      segdev_sync(struct seg *, caddr_t, size_t, int, uint_t);
173 static size_t   segdev_incore(struct seg *, caddr_t, size_t, char *);
174 static int      segdev_lockop(struct seg *, caddr_t, size_t, int, int,
175                     ulong_t *, size_t);
176 static int      segdev_getprot(struct seg *, caddr_t, size_t, uint_t *);
177 static u_offset_t       segdev_getoffset(struct seg *, caddr_t);
178 static int      segdev_gettype(struct seg *, caddr_t);
179 static int      segdev_getvp(struct seg *, caddr_t, struct vnode **);
180 static int      segdev_advise(struct seg *, caddr_t, size_t, uint_t);
181 static void     segdev_dump(struct seg *);
181 static int      segdev_pagelock(struct seg *, caddr_t, size_t,
182                     struct page ***, enum lock_type, enum seg_rw);
184 static int      segdev_setpagesize(struct seg *, caddr_t, size_t, uint_t);
183 static int      segdev_getmemid(struct seg *, caddr_t, memid_t *);
186 static lgrp_mem_policy_info_t   *segdev_getpolicy(struct seg *, caddr_t);
187 static int      segdev_capable(struct seg *, segcapability_t);

185 /*
186  * XXX  this struct is used by rootnex_map_fault to identify
187  *      the segment it has been passed. So if you make it
188  *      "static" you'll need to fix rootnex_map_fault.
189  */
190 const struct seg_ops segdev_ops = {
191         .dup            = segdev_dup,
192         .unmap          = segdev_unmap,
193         .free           = segdev_free,
194         .fault          = segdev_fault,
195         .faulta         = segdev_faulta,
196         .setprot        = segdev_setprot,
197         .checkprot      = segdev_checkprot,
198         .kluster        = (int (*)())segdev_badop,
199         .sync           = segdev_sync,
200         .incore         = segdev_incore,
201         .lockop         = segdev_lockop,
202         .getprot        = segdev_getprot,
203         .getoffset      = segdev_getoffset,
204         .gettype        = segdev_gettype,
205         .getvp          = segdev_getvp,
206         .advise         = segdev_advise,
207         .pagelock       = segdev_pagelock,
208         .getmemid       = segdev_getmemid,
194 struct seg_ops segdev_ops = {
195         segdev_dup,
196         segdev_unmap,
197         segdev_free,
198         segdev_fault,
199         segdev_faulta,
200         segdev_setprot,
201         segdev_checkprot,
202         (int (*)())segdev_badop,        /* kluster */
203         (size_t (*)(struct seg *))NULL, /* swapout */
204         segdev_sync,                    /* sync */
205         segdev_incore,
206         segdev_lockop,                  /* lockop */
207         segdev_getprot,
208         segdev_getoffset,
209         segdev_gettype,
210         segdev_getvp,
211         segdev_advise,
212         segdev_dump,
213         segdev_pagelock,
214         segdev_setpagesize,
215         segdev_getmemid,
216         segdev_getpolicy,
217         segdev_capable,
```

```
218         seg_inherit_notsup
209 };
_____unchanged_portion_omitted_

346 /*
347  * Create a device segment.
348  */
349 int
350 segdev_create(struct seg *seg, void *argsp)
351 {
352         struct segdev_data *sdp;
353         struct segdev_crargs *a = (struct segdev_crargs *)argsp;
354         devmap_handle_t *dhp = (devmap_handle_t *)a->devmap_data;
355         int error;

357         /*
358          * Since the address space is "write" locked, we
359          * don't need the segment lock to protect "segdev" data.
360          */
361         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

363         hat_map(seg->s_as->a_hat, seg->s_base, seg->s_size, HAT_MAP);

365         sdp = sdp_alloc();

367         sdp->mapfunc = a->mapfunc;
368         sdp->offset = a->offset;
369         sdp->prot = a->prot;
370         sdp->maxprot = a->maxprot;
371         sdp->type = a->type;
372         sdp->pageprot = 0;
373         sdp->softlockcnt = 0;
374         sdp->vpage = NULL;

376         if (sdp->mapfunc == NULL)
377                 sdp->devmap_data = dhp;
378         else
379                 sdp->devmap_data = dhp = NULL;

381         sdp->hat_flags = a->hat_flags;
382         sdp->hat_attr = a->hat_attr;

384         /*
385          * Currently, hat_flags supports only HAT_LOAD_NOCONSIST
386          */
387         ASSERT(!(sdp->hat_flags & ~HAT_LOAD_NOCONSIST));

389         /*
390          * Hold shadow vnode -- segdev only deals with
391          * character (VCHR) devices. We use the common
392          * vp to hang pages on.
393          */
394         sdp->vp = specfind(a->dev, VCHR);
395         ASSERT(sdp->vp != NULL);

397         seg->s_ops = &segdev_ops;
398         seg->s_data = sdp;

400         while (dhp != NULL) {
401                 dhp->dh_seg = seg;
402                 dhp = dhp->dh_next;
403         }

405         /*
406          * Inform the vnode of the new mapping.
407          */
```



```
408         /*
409          * It is ok to use pass sdp->maxprot to ADDMAP rather than to use
410          * dhp specific maxprot because spec_addmap does not use maxprot.
411          */
412         error = VOP_ADDMAP(VTOCVP(sdp->vp), sdp->offset,
413             seg->s_as, seg->s_base, seg->s_size,
414             sdp->prot, sdp->maxprot, sdp->type, CRED(), NULL);

416         if (error != 0) {
417                 sdp->devmap_data = NULL;
418                 hat_unload(seg->s_as->a_hat, seg->s_base, seg->s_size,
419                     HAT_UNLOAD_UNMAP);
420         } else {
421                 /*
422                  * Mappings of /dev/null don't count towards the VSZ of a
423                  * process.  Mappings of /dev/null have no mapping type.
424                  */
425                 if ((segop_gettype(seg, seg->s_base) & (MAP_SHARED |
435                 if ((SEGOP_GETTYPE(seg, (seg)->s_base) & (MAP_SHARED |
426                     MAP_PRIVATE)) == 0) {
427                         seg->s_as->a_resvsize -= seg->s_size;
428                 }
429         }

431         return (error);
432 }
_____unchanged_portion_omitted_

2373 /*
2384  * segdev pages are not dumped, so we just return
2385  */
2386 /*ARGSUSED*/
2387 static void
2388 segdev_dump(struct seg *seg)
2389 {}

2391 /*
2374  * ddi_segmap_setup:    Used by drivers who wish specify mapping attributes
2375  *                      for a segment.  Called from a drivers segmap(9E)
2376  *                      routine.
2377  */
2378 /*ARGSUSED*/
2379 int
2380 ddi_segmap_setup(dev_t dev, off_t offset, struct as *as, caddr_t *addrp,
2381     off_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *cred,
2382     ddi_device_acc_attr_t *accattrp, uint_t rnumber)
2383 {
2384         struct segdev_crargs dev_a;
2385         int (*mapfunc)(dev_t dev, off_t off, int prot);
2386         uint_t hat_attr;
2387         pfn_t pfn;
2388         int     error, i;

2390         TRACE_0(TR_FAC_DEVMAP, TR_DEVMAP_SEGMAP_SETUP,
2391             "ddi_segmap_setup:start");

2393         if ((mapfunc = devopsp[getmajor(dev)]->devo_cb_ops->cb_mmap) == nodev)
2394                 return (ENODEV);

2396         /*
2397          * Character devices that support the d_mmap
2398          * interface can only be mmap'ed shared.
2399          */
2400         if ((flags & MAP_TYPE) != MAP_SHARED)
2401                 return (EINVAL);
```

```
2403         /*
2404          * Check that this region is indeed mappable on this platform.
2405          * Use the mapping function.
2406          */
2407         if (ddi_device_mapping_check(dev, accattrp, rnumber, &hat_attr) == -1)
2408                 return (ENXIO);

2410         /*
2411          * Check to ensure that the entire range is
2412          * legal and we are not trying to map in
2413          * more than the device will let us.
2414          */
2415         for (i = 0; i < len; i += PAGESIZE) {
2416                 if (i == 0) {
2417                         /*
2418                          * Save the pfn at offset here. This pfn will be
2419                          * used later to get user address.
2420                          */
2421                         if ((pfn = (pfn_t)cdev_mmap(mapfunc, dev, offset,
2422                             maxprot)) == PFN_INVALID)
2423                                 return (ENXIO);
2424                 } else {
2425                         if (cdev_mmap(mapfunc, dev, offset + i, maxprot) ==
2426                             PFN_INVALID)
2427                                 return (ENXIO);
2428                 }
2429         }

2431         as_rangelock(as);
2432         /* Pick an address w/o worrying about any vac alignment constraints. */
2433         error = choose_addr(as, addrp, len, ptob(pfn), ADDR_NOVACALIGN, flags);
2434         if (error != 0) {
2435                 as_rangeunlock(as);
2436                 return (error);
2437         }

2439         dev_a.mapfunc = mapfunc;
2440         dev_a.dev = dev;
2441         dev_a.offset = (offset_t)offset;
2442         dev_a.type = flags & MAP_TYPE;
2443         dev_a.prot = (uchar_t)prot;
2444         dev_a.maxprot = (uchar_t)maxprot;
2445         dev_a.hat_attr = hat_attr;
2446         dev_a.hat_flags = 0;
2447         dev_a.devmap_data = NULL;

2449         error = as_map(as, *addrp, len, segdev_create, &dev_a);
2450         as_rangeunlock(as);
2451         return (error);

2453 }
```

_____unchanged_portion_omitted_

```
2483 /*ARGSUSED*/
2484 static int
2485 segdev_setpagesize(struct seg *seg, caddr_t addr, size_t len,
2486     uint_t szc)
2487 {
2488         return (ENOTSUP);
2489 }

2465 /*
2466  * devmap_device: Used by devmap framework to establish mapping
2467  *                called by devmap_seup(9F) during map setup time.
2468  */
2469 /*ARGSUSED*/
```



```
2470 static int
2471 devmap_device(devmap_handle_t *dhp, struct as *as, caddr_t *addr,
2472     offset_t off, size_t len, uint_t flags)
2473 {
2474         devmap_handle_t *rdhp, *maxdhp;
2475         struct segdev_crargs dev_a;
2476         int     err;
2477         uint_t maxprot = PROT_ALL;
2478         offset_t offset = 0;
2479         pfn_t pfn;
2480         struct devmap_pmem_cookie *pcp;

2482         TRACE_4(TR_FAC_DEVMAP, TR_DEVMAP_DEVICE,
2483             "devmap_device:start dhp=%p addr=%p off=%llx, len=%lx",
2484             (void *)dhp, (void *)addr, off, len);

2486         DEBUGF(2, (CE_CONT, "devmap_device: dhp %p addr %p off %llx len %lx\n",
2487             (void *)dhp, (void *)addr, off, len));

2489         as_rangelock(as);
2490         if ((flags & MAP_FIXED) == 0) {
2491                 offset_t aligned_off;

2493                 rdhp = maxdhp = dhp;
2494                 while (rdhp != NULL) {
2495                         maxdhp = (maxdhp->dh_len > rdhp->dh_len) ?
2496                             maxdhp : rdhp;
2497                         rdhp = rdhp->dh_next;
2498                         maxprot |= dhp->dh_maxprot;
2499                 }
2500                 offset = maxdhp->dh_uoff - dhp->dh_uoff;

2502                 /*
2503                  * Use the dhp that has the
2504                  * largest len to get user address.
2505                  */
2506                 /*
2507                  * If MAPPING_INVALID, cannot use dh_pfn/dh_cvaddr,
2508                  * use 0 which is as good as any other.
2509                  */
2510                 if (maxdhp->dh_flags & DEVMAP_MAPPING_INVALID) {
2511                         aligned_off = (offset_t)0;
2512                 } else if (dhp_is_devmem(maxdhp)) {
2513                         aligned_off = (offset_t)ptob(maxdhp->dh_pfn) - offset;
2514                 } else if (dhp_is_pmem(maxdhp)) {
2515                         pcp = (struct devmap_pmem_cookie *)maxdhp->dh_pcookie;
2516                         pfn = page_pptonum(
2517                             pcp->dp_pparray[btop(maxdhp->dh_roff)]);
2518                         aligned_off = (offset_t)ptob(pfn) - offset;
2519                 } else {
2520                         aligned_off = (offset_t)(uintptr_t)maxdhp->dh_cvaddr -
2521                             offset;
2522                 }

2524                 /*
2525                  * Pick an address aligned to dh_cookie.
2526                  * for kernel memory/user memory, cookie is cvaddr.
2527                  * for device memory, cookie is physical address.
2528                  */
2529                 map_addr(addr, len, aligned_off, 1, flags);
2530                 if (*addr == NULL) {
2531                         as_rangeunlock(as);
2532                         return (ENOMEM);
2533                 }
2534         } else {
2535                 /*
```

```
2536                  * User-specified address; blow away any previous mappings.
2537                  */
2538                 (void) as_unmap(as, *addr, len);
2539         }

2541         dev_a.mapfunc = NULL;
2542         dev_a.dev = dhp->dh_dev;
2543         dev_a.type = flags & MAP_TYPE;
2544         dev_a.offset = off;
2545         /*
2546          * sdp->maxprot has the least restrict protection of all dhps.
2547          */
2548         dev_a.maxprot = maxprot;
2549         dev_a.prot = dhp->dh_prot;
2550         /*
2551          * devmap uses dhp->dh_hat_attr for hat.
2552          */
2553         dev_a.hat_flags = 0;
2554         dev_a.hat_attr = 0;
2555         dev_a.devmap_data = (void *)dhp;

2557         err = as_map(as, *addr, len, segdev_create, &dev_a);
2558         as_rangeunlock(as);
2559         return (err);
2560 }
```

_____unchanged_portion_omitted_

```
3994 static int
3995 segdev_getmemid(struct seg *seg, caddr_t addr, memid_t *memidp)
3996 {
3997         struct segdev_data *sdp = (struct segdev_data *)seg->s_data;

3999         /*
4000          * It looks as if it is always mapped shared
4001          */
4002         TRACE_0(TR_FAC_DEVMAP, TR_DEVMAP_GETMEMID,
4003             "segdev_getmemid:start");
4004         memidp->val[0] = (uintptr_t)VTOCVP(sdp->vp);
4005         memidp->val[1] = sdp->offset + (uintptr_t)(addr - seg->s_base);
4032         return (0);
4033 }

4035 /*ARGSUSED*/
4036 static lgrp_mem_policy_info_t *
4037 segdev_getpolicy(struct seg *seg, caddr_t addr)
4038 {
4039         return (NULL);
4040 }

4042 /*ARGSUSED*/
4043 static int
4044 segdev_capable(struct seg *seg, segcapability_t capability)
4045 {
4006         return (0);
4007 }
```

_____unchanged_portion_omitted_

****
4470 Fri May 8 18:10:35 2015
new/usr/src/uts/common/vm/seg_dev.h
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
patch lower-case-segops
****
_____***unchanged_portion_omitted_***

```
113 #ifdef _KERNEL

115 /*
116  * Mappings of /dev/null come from segdev and have no mapping type.
117  */

119 #define SEG_IS_DEVNULL_MAPPING(seg)     \
120         ((seg)->s_ops == &segdev_ops && \
121         ((segop_gettype((seg), (seg)->s_base) & (MAP_SHARED | MAP_PRIVATE)) == 0
121         ((SEGOP_GETTYPE(seg, (seg)->s_base) & (MAP_SHARED | MAP_PRIVATE)) == 0))

123 extern void segdev_init(void);

125 extern int segdev_create(struct seg *, void *);

127 extern int segdev_copyto(struct seg *, caddr_t, const void *, void *, size_t);
128 extern int segdev_copyfrom(struct seg *, caddr_t, const void *, void *, size_t);
129 extern const struct seg_ops segdev_ops;
129 extern struct seg_ops segdev_ops;

131 #endif  /* _KERNEL */

133 #ifdef  __cplusplus
134 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   44758 Fri May  8 18:10:35 2015
new/usr/src/uts/common/vm/seg_kmem.c
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
segop_getpolicy already checks for a NULL op
seg_inherit_notsup is redundant since segop_inherit checks for NULL properly
no need for bad-op segment op functions
The segment drivers have a number of bad-op functions that simply panic.
Keeping the function pointer NULL will accomplish the same thing in most
cases.  In other cases, keeping the function pointer NULL will result in
proper error code being returned.
patch lower-case-segops
use C99 initializers in segment ops structures
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

 431 static void
 432 segkmem_badop()
 433 {
 434         panic("segkmem_badop");
 435 }

 437 #define SEGKMEM_BADOP(t)        (t(*)())segkmem_badop

 431 /*ARGSUSED*/
 432 static faultcode_t
 433 segkmem_fault(struct hat *hat, struct seg *seg, caddr_t addr, size_t size,
 434         enum fault_type type, enum seg_rw rw)
 435 {
 436         pgcnt_t npages;
 437         spgcnt_t pg;
 438         page_t *pp;
 439         struct vnode *vp = seg->s_data;

 441         ASSERT(RW_READ_HELD(&seg->s_as->a_lock));

 443         if (seg->s_as != &kas || size > seg->s_size ||
 444             addr < seg->s_base || addr + size > seg->s_base + seg->s_size)
 445                 panic("segkmem_fault: bad args");

 447         /*
 448          * If it is one of segkp pages, call segkp_fault.
 449          */
 450         if (segkp_bitmap && seg == &kvseg &&
 451             BT_TEST(segkp_bitmap, btop((uintptr_t)(addr - seg->s_base))))
 452                 return (segop_fault(hat, segkp, addr, size, type, rw));
 460                 return (SEGOP_FAULT(hat, segkp, addr, size, type, rw));

 454         if (rw != S_READ && rw != S_WRITE && rw != S_OTHER)
 455                 return (FC_NOSUPPORT);

 457         npages = btopr(size);

 459         switch (type) {
 460         case F_SOFTLOCK:        /* lock down already-loaded translations */
 461                 for (pg = 0; pg < npages; pg++) {
 462                         pp = page_lookup(vp, (u_offset_t)(uintptr_t)addr,
 463                             SE_SHARED);
```



```
 464                         if (pp == NULL) {
 465                                 /*
 466                                  * Hmm, no page. Does a kernel mapping
 467                                  * exist for it?
 468                                  */
 469                                 if (!hat_probe(kas.a_hat, addr)) {
 470                                         addr -= PAGESIZE;
 471                                         while (--pg >= 0) {
 472                                                 pp = page_find(vp, (u_offset_t)
 473                                                     (uintptr_t)addr);
 474                                                 if (pp)
 475                                                         page_unlock(pp);
 476                                                 addr -= PAGESIZE;
 477                                         }
 478                                         return (FC_NOMAP);
 479                                 }
 480                         }
 481                         addr += PAGESIZE;
 482                 }
 483                 if (rw == S_OTHER)
 484                         hat_reserve(seg->s_as, addr, size);
 485                 return (0);
 486         case F_SOFTUNLOCK:
 487                 while (npages--) {
 488                         pp = page_find(vp, (u_offset_t)(uintptr_t)addr);
 489                         if (pp)
 490                                 page_unlock(pp);
 491                         addr += PAGESIZE;
 492                 }
 493                 return (0);
 494         default:
 495                 return (FC_NOSUPPORT);
 496         }
 497         /*NOTREACHED*/
 498 }

 500 static int
 501 segkmem_setprot(struct seg *seg, caddr_t addr, size_t size, uint_t prot)
 502 {
 503         ASSERT(RW_LOCK_HELD(&seg->s_as->a_lock));

 505         if (seg->s_as != &kas || size > seg->s_size ||
 506             addr < seg->s_base || addr + size > seg->s_base + seg->s_size)
 507                 panic("segkmem_setprot: bad args");

 509         /*
 510          * If it is one of segkp pages, call segkp.
 511          */
 512         if (segkp_bitmap && seg == &kvseg &&
 513             BT_TEST(segkp_bitmap, btop((uintptr_t)(addr - seg->s_base))))
 514                 return (segop_setprot(segkp, addr, size, prot));
 522                 return (SEGOP_SETPROT(segkp, addr, size, prot));

 516         if (prot == 0)
 517                 hat_unload(kas.a_hat, addr, size, HAT_UNLOAD);
 518         else
 519                 hat_chgprot(kas.a_hat, addr, size, prot);
 520         return (0);
 521 }

 523 /*
 524  * This is a dummy segkmem function overloaded to call segkp
 525  * when segkp is under the heap.
 526  */
 527 /* ARGSUSED */
 528 static int
```

```
529 segkmem_checkprot(struct seg *seg, caddr_t addr, size_t size, uint_t prot)
530 {
531         ASSERT(RW_LOCK_HELD(&seg->s_as->a_lock));

533         if (seg->s_as != &kas)
534                 panic("segkmem badop");
542                 segkmem_badop();

536         /*
537          * If it is one of segkp pages, call into segkp.
538          */
539         if (segkp_bitmap && seg == &kvseg &&
540             BT_TEST(segkp_bitmap, btop((uintptr_t)(addr - seg->s_base))))
541                 return (segop_checkprot(segkp, addr, size, prot));
549                 return (SEGOP_CHECKPROT(segkp, addr, size, prot));

543         panic("segkmem badop");
551         segkmem_badop();
544         return (0);
545 }

547 /*
548  * This is a dummy segkmem function overloaded to call segkp
549  * when segkp is under the heap.
550  */
551 /* ARGSUSED */
552 static int
553 segkmem_kluster(struct seg *seg, caddr_t addr, ssize_t delta)
554 {
555         ASSERT(RW_LOCK_HELD(&seg->s_as->a_lock));

557         if (seg->s_as != &kas)
558                 panic("segkmem badop");
566                 segkmem_badop();

560         /*
561          * If it is one of segkp pages, call into segkp.
562          */
563         if (segkp_bitmap && seg == &kvseg &&
564             BT_TEST(segkp_bitmap, btop((uintptr_t)(addr - seg->s_base))))
565                 return (segop_kluster(segkp, addr, delta));
573                 return (SEGOP_KLUSTER(segkp, addr, delta));

567         panic("segkmem badop");
575         segkmem_badop();
568         return (0);
569 }
_____unchanged_portion_omitted_

661 /*
662  * lock/unlock kmem pages over a given range [addr, addr+len).
663  * Returns a shadow list of pages in ppp. If there are holes
664  * in the range (e.g. some of the kernel mappings do not have
665  * underlying page_ts) returns ENOTSUP so that as_pagelock()
666  * will handle the range via as_fault(F_SOFTLOCK).
667  */
668 /*ARGSUSED*/
669 static int
670 segkmem_pagelock(struct seg *seg, caddr_t addr, size_t len,
671         page_t ***ppp, enum lock_type type, enum seg_rw rw)
672 {
673         page_t **pplist, *pp;
674         pgcnt_t npages;
675         spgcnt_t pg;
676         size_t nb;
677         struct vnode *vp = seg->s_data;
```



```
679         ASSERT(ppp != NULL);

681         /*
682          * If it is one of segkp pages, call into segkp.
683          */
684         if (segkp_bitmap && seg == &kvseg &&
685             BT_TEST(segkp_bitmap, btop((uintptr_t)(addr - seg->s_base))))
686                 return (segop_pagelock(segkp, addr, len, ppp, type, rw));
694                 return (SEGOP_PAGELOCK(segkp, addr, len, ppp, type, rw));

688         npages = btopr(len);
689         nb = sizeof (page_t *) * npages;

691         if (type == L_PAGEUNLOCK) {
692                 pplist = *ppp;
693                 ASSERT(pplist != NULL);

695                 for (pg = 0; pg < npages; pg++) {
696                         pp = pplist[pg];
697                         page_unlock(pp);
698                 }
699                 kmem_free(pplist, nb);
700                 return (0);
701         }

703         ASSERT(type == L_PAGELOCK);

705         pplist = kmem_alloc(nb, KM_NOSLEEP);
706         if (pplist == NULL) {
707                 *ppp = NULL;
708                 return (ENOTSUP);       /* take the slow path */
709         }

711         for (pg = 0; pg < npages; pg++) {
712                 pp = page_lookup(vp, (u_offset_t)(uintptr_t)addr, SE_SHARED);
713                 if (pp == NULL) {
714                         while (--pg >= 0)
715                                 page_unlock(pplist[pg]);
716                         kmem_free(pplist, nb);
717                         *ppp = NULL;
718                         return (ENOTSUP);
719                 }
720                 pplist[pg] = pp;
721                 addr += PAGESIZE;
722         }

724         *ppp = pplist;
725         return (0);
726 }

728 /*
729  * This is a dummy segkmem function overloaded to call segkp
730  * when segkp is under the heap.
731  */
732 /* ARGSUSED */
733 static int
734 segkmem_getmemid(struct seg *seg, caddr_t addr, memid_t *memidp)
735 {
736         ASSERT(RW_LOCK_HELD(&seg->s_as->a_lock));

738         if (seg->s_as != &kas)
739                 panic("segkmem badop");
747                 segkmem_badop();

741         /*
```

```
 742          * If it is one of segkp pages, call into segkp.
 743          */
 744         if (segkp_bitmap && seg == &kvseg &&
 745             BT_TEST(segkp_bitmap, btop((uintptr_t)(addr - seg->s_base))))
 746                 return (segop_getmemid(segkp, addr, memidp));
 754                 return (SEGOP_GETMEMID(segkp, addr, memidp));

 748         panic("segkmem badop");
 756         segkmem_badop();
 749         return (0);
 750 }

 752 /*ARGSUSED*/
 761 static lgrp_mem_policy_info_t *
 762 segkmem_getpolicy(struct seg *seg, caddr_t addr)
 763 {
 764         return (NULL);
 765 }

 767 /*ARGSUSED*/
 753 static int
 754 segkmem_capable(struct seg *seg, segcapability_t capability)
 755 {
 756         if (capability == S_CAPABILITY_NOMINFLT)
 757                 return (1);
 758         return (0);
 759 }

 761 static const struct seg_ops segkmem_ops = {
 762         .fault          = segkmem_fault,
 763         .setprot        = segkmem_setprot,
 764         .checkprot      = segkmem_checkprot,
 765         .kluster        = segkmem_kluster,
 766         .dump           = segkmem_dump,
 767         .pagelock       = segkmem_pagelock,
 768         .getmemid       = segkmem_getmemid,
 769         .capable        = segkmem_capable,
 776 static struct seg_ops segkmem_ops = {
 777         SEGKMEM_BADOP(int),             /* dup */
 778         SEGKMEM_BADOP(int),             /* unmap */
 779         SEGKMEM_BADOP(void),            /* free */
 780         segkmem_fault,
 781         SEGKMEM_BADOP(faultcode_t),     /* faulta */
 782         segkmem_setprot,
 783         segkmem_checkprot,
 784         segkmem_kluster,
 785         SEGKMEM_BADOP(size_t),          /* swapout */
 786         SEGKMEM_BADOP(int),             /* sync */
 787         SEGKMEM_BADOP(size_t),          /* incore */
 788         SEGKMEM_BADOP(int),             /* lockop */
 789         SEGKMEM_BADOP(int),             /* getprot */
 790         SEGKMEM_BADOP(u_offset_t),      /* getoffset */
 791         SEGKMEM_BADOP(int),             /* gettype */
 792         SEGKMEM_BADOP(int),             /* getvp */
 793         SEGKMEM_BADOP(int),             /* advise */
 794         segkmem_dump,
 795         segkmem_pagelock,
 796         SEGKMEM_BADOP(int),             /* setpgsz */
 797         segkmem_getmemid,
 798         segkmem_getpolicy,              /* getpolicy */
 799         segkmem_capable,                /* capable */
 800         seg_inherit_notsup              /* inherit */
 770 };
```

*unchanged_portion_omitted_*

```
1613 #endif
```

```
**********************************************************
   35647 Fri May  8 18:10:35 2015
new/usr/src/uts/common/vm/seg_kp.c
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
use NULL getmemid segop as a shorthand for ENODEV
Instead of forcing every segment driver to implement a dummy function to
return (hopefully) ENODEV, handle NULL getmemid segop function pointer as
"return ENODEV" shorthand.
use NULL capable segop as a shorthand for no-capabilities
Instead of forcing every segment driver to implement a dummy "return 0"
function, handle NULL capable segop function pointer as "no copabilities
supported" shorthand.
segop_getpolicy already checks for a NULL op
seg_inherit_notsup is redundant since segop_inherit checks for NULL properly
no need for bad-op segment op functions
The segment drivers have a number of bad-op functions that simply panic.
Keeping the function pointer NULL will accomplish the same thing in most
cases.  In other cases, keeping the function pointer NULL will result in
proper error code being returned.
use C99 initializers in segment ops structures
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
```

```
   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) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
  23  */

  25 /* Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
  26 /*       All Rights Reserved   */

  28 /*
  29  * Portions of this source code were derived from Berkeley 4.3 BSD
  30  * under license from the Regents of the University of California.
  31  */

  33 /*
  34  * segkp is a segment driver that administers the allocation and deallocation
  35  * of pageable variable size chunks of kernel virtual address space. Each
  36  * allocated resource is page-aligned.
  37  *
```



```
  38  * The user may specify whether the resource should be initialized to 0,
  39  * include a redzone, or locked in memory.
  40  */

  42 #include <sys/types.h>
  43 #include <sys/t_lock.h>
  44 #include <sys/thread.h>
  45 #include <sys/param.h>
  46 #include <sys/errno.h>
  47 #include <sys/sysmacros.h>
  48 #include <sys/systm.h>
  49 #include <sys/buf.h>
  50 #include <sys/mman.h>
  51 #include <sys/vnode.h>
  52 #include <sys/cmn_err.h>
  53 #include <sys/swap.h>
  54 #include <sys/tuneable.h>
  55 #include <sys/kmem.h>
  56 #include <sys/vmem.h>
  57 #include <sys/cred.h>
  58 #include <sys/dumphdr.h>
  59 #include <sys/debug.h>
  60 #include <sys/vtrace.h>
  61 #include <sys/stack.h>
  62 #include <sys/atomic.h>
  63 #include <sys/archsystm.h>
  64 #include <sys/lgrp.h>

  66 #include <vm/as.h>
  67 #include <vm/seg.h>
  68 #include <vm/seg_kp.h>
  69 #include <vm/seg_kmem.h>
  70 #include <vm/anon.h>
  71 #include <vm/page.h>
  72 #include <vm/hat.h>
  73 #include <sys/bitmap.h>

  75 /*
  76  * Private seg op routines
  77  */
  78 static void     segkp_badop(void);
  78 static void     segkp_dump(struct seg *seg);
  79 static int      segkp_checkprot(struct seg *seg, caddr_t addr, size_t len,
  80                         uint_t prot);
  81 static int      segkp_kluster(struct seg *seg, caddr_t addr, ssize_t delta);
  82 static int      segkp_pagelock(struct seg *seg, caddr_t addr, size_t len,
  83                         struct page ***page, enum lock_type type,
  84                         enum seg_rw rw);
  85 static void     segkp_insert(struct seg *seg, struct segkp_data *kpd);
  86 static void     segkp_delete(struct seg *seg, struct segkp_data *kpd);
  87 static caddr_t  segkp_get_internal(struct seg *seg, size_t len, uint_t flags,
  88                         struct segkp_data **tkpd, struct anon_map *amp);
  89 static void     segkp_release_internal(struct seg *seg,
  90                         struct segkp_data *kpd, size_t len);
  91 static int      segkp_unlock(struct hat *hat, struct seg *seg, caddr_t vaddr,
  92                         size_t len, struct segkp_data *kpd, uint_t flags);
  93 static int      segkp_load(struct hat *hat, struct seg *seg, caddr_t vaddr,
  94                         size_t len, struct segkp_data *kpd, uint_t flags);
  95 static struct   segkp_data *segkp_find(struct seg *seg, caddr_t vaddr);
  97 static int      segkp_getmemid(struct seg *seg, caddr_t addr, memid_t *memidp);
  98 static lgrp_mem_policy_info_t   *segkp_getpolicy(struct seg *seg,
  99     caddr_t addr);
 100 static int      segkp_capable(struct seg *seg, segcapability_t capability);

  97 /*
  98  * Lock used to protect the hash table(s) and caches.
```

```
 99  */
100 static kmutex_t segkp_lock;

102 /*
103  * The segkp caches
104  */
105 static struct segkp_cache segkp_cache[SEGKP_MAX_CACHE];

112 #define SEGKP_BADOP(t)  (t(*)())segkp_badop

107 /*
108  * When there are fewer than red_minavail bytes left on the stack,
109  * segkp_map_red() will map in the redzone (if called).  5000 seems
110  * to work reasonably well...
111  */
112 long            red_minavail = 5000;

114 /*
115  * will be set to 1 for 32 bit x86 systems only, in startup.c
116  */
117 int     segkp_fromheap = 0;
118 ulong_t *segkp_bitmap;

120 /*
121  * If segkp_map_red() is called with the redzone already mapped and
122  * with less than RED_DEEP_THRESHOLD bytes available on the stack,
123  * then the stack situation has become quite serious;  if much more stack
124  * is consumed, we have the potential of scrogging the next thread/LWP
125  * structure.  To help debug the "can't happen" panics which may
126  * result from this condition, we record hrestime and the calling thread
127  * in red_deep_hires and red_deep_thread respectively.
128  */
129 #define RED_DEEP_THRESHOLD      2000

131 hrtime_t        red_deep_hires;
132 kthread_t       *red_deep_thread;

134 uint32_t        red_nmapped;
135 uint32_t        red_closest = UINT_MAX;
136 uint32_t        red_ndoubles;

138 pgcnt_t anon_segkp_pages_locked;        /* See vm/anon.h */
139 pgcnt_t anon_segkp_pages_resv;          /* anon reserved by seg_kp */

141 static const struct seg_ops segkp_ops = {
142         .fault          = segkp_fault,
143         .checkprot      = segkp_checkprot,
144         .kluster        = segkp_kluster,
145         .dump           = segkp_dump,
146         .pagelock       = segkp_pagelock,
148 static struct   seg_ops segkp_ops = {
149         SEGKP_BADOP(int),               /* dup */
150         SEGKP_BADOP(int),               /* unmap */
151         SEGKP_BADOP(void),              /* free */
152         segkp_fault,
153         SEGKP_BADOP(faultcode_t),       /* faulta */
154         SEGKP_BADOP(int),               /* setprot */
155         segkp_checkprot,
156         segkp_kluster,
157         SEGKP_BADOP(size_t),            /* swapout */
158         SEGKP_BADOP(int),               /* sync */
159         SEGKP_BADOP(size_t),            /* incore */
160         SEGKP_BADOP(int),               /* lockop */
161         SEGKP_BADOP(int),               /* getprot */
162         SEGKP_BADOP(u_offset_t),                /* getoffset */
163         SEGKP_BADOP(int),               /* gettype */
```



```
164         SEGKP_BADOP(int),               /* getvp */
165         SEGKP_BADOP(int),               /* advise */
166         segkp_dump,                     /* dump */
167         segkp_pagelock,                 /* pagelock */
168         SEGKP_BADOP(int),               /* setpgsz */
169         segkp_getmemid,                 /* getmemid */
170         segkp_getpolicy,                /* getpolicy */
171         segkp_capable,                  /* capable */
172         seg_inherit_notsup              /* inherit */
147 };

176 static void
177 segkp_badop(void)
178 {
179         panic("segkp_badop");
180         /*NOTREACHED*/
181 }

150 static void segkpinit_mem_config(struct seg *);

152 static uint32_t segkp_indel;

154 /*
155  * Allocate the segment specific private data struct and fill it in
156  * with the per kp segment mutex, anon ptr. array and hash table.
157  */
158 int
159 segkp_create(struct seg *seg)
160 {
161         struct segkp_segdata *kpsd;
162         size_t  np;

164         ASSERT(seg != NULL && seg->s_as == &kas);
165         ASSERT(RW_WRITE_HELD(&seg->s_as->a_lock));

167         if (seg->s_size & PAGEOFFSET) {
168                 panic("Bad segkp size");
169                 /*NOTREACHED*/
170         }

172         kpsd = kmem_zalloc(sizeof (struct segkp_segdata), KM_SLEEP);

174         /*
175          * Allocate the virtual memory for segkp and initialize it
176          */
177         if (segkp_fromheap) {
178                 np = btop(kvseg.s_size);
179                 segkp_bitmap = kmem_zalloc(BT_SIZEOFMAP(np), KM_SLEEP);
180                 kpsd->kpsd_arena = vmem_create("segkp", NULL, 0, PAGESIZE,
181                     vmem_alloc, vmem_free, heap_arena, 5 * PAGESIZE, VM_SLEEP);
182         } else {
183                 segkp_bitmap = NULL;
184                 np = btop(seg->s_size);
185                 kpsd->kpsd_arena = vmem_create("segkp", seg->s_base,
186                     seg->s_size, PAGESIZE, NULL, NULL, NULL, 5 * PAGESIZE,
187                     VM_SLEEP);
188         }

190         kpsd->kpsd_anon = anon_create(np, ANON_SLEEP | ANON_ALLOC_FORCE);

192         kpsd->kpsd_hash = kmem_zalloc(SEGKP_HASHSZ * sizeof (struct segkp *),
193             KM_SLEEP);
194         seg->s_data = (void *)kpsd;
195         seg->s_ops = &segkp_ops;
196         segkpinit_mem_config(seg);
```

```
 197         return (0);
 198 }
_____unchanged_portion_omitted_

 725 /*
 726  * segkp_map_red() will check the current frame pointer against the
 727  * stack base.  If the amount of stack remaining is questionable
 728  * (less than red_minavail), then segkp_map_red() will map in the redzone
 729  * and return 1.  Otherwise, it will return 0.  segkp_map_red() can
 730  * _only_ be called when it is safe to sleep on page_create_va().
 763  * _only_ be called when:
 764  *
 765  *   - it is safe to sleep on page_create_va().
 766  *   - the caller is non-swappable.
 731  *
 732  * It is up to the caller to remember whether segkp_map_red() successfully
 733  * mapped the redzone, and, if so, to call segkp_unmap_red() at a later
 734  * time.
 770  * time.  Note that the caller must _remain_ non-swappable until after
 771  * calling segkp_unmap_red().
 735  *
 736  * Currently, this routine is only called from pagefault() (which necessarily
 737  * satisfies the above conditions).
 738  */
 739 #if defined(STACK_GROWTH_DOWN)
 740 int
 741 segkp_map_red(void)
 742 {
 743         uintptr_t fp = STACK_BIAS + (uintptr_t)getfp();
 744 #ifndef _LP64
 745         caddr_t stkbase;
 746 #endif

 785         ASSERT(curthread->t_schedflag & TS_DONT_SWAP);

 748         /*
 749          * Optimize for the common case where we simply return.
 750          */
 751         if ((curthread->t_red_pp == NULL) &&
 752             (fp - (uintptr_t)curthread->t_stkbase >= red_minavail))
 753                 return (0);

 755 #if defined(_LP64)
 756         /*
 757          * XXX  We probably need something better than this.
 758          */
 759         panic("kernel stack overflow");
 760         /*NOTREACHED*/
 761 #else /* _LP64 */
 762         if (curthread->t_red_pp == NULL) {
 763                 page_t *red_pp;
 764                 struct seg kseg;

 766                 caddr_t red_va = (caddr_t)
 767                     (((uintptr_t)curthread->t_stkbase & (uintptr_t)PAGEMASK) -
 768                     PAGESIZE);

 770                 ASSERT(page_exists(&kvp, (u_offset_t)(uintptr_t)red_va) ==
 771                     NULL);

 773                 /*
 774                  * Allocate the physical for the red page.
 775                  */
 776                 /*
 777                  * No PG_NORELOC here to avoid waits. Unlikely to get
 778                  * a relocate happening in the short time the page exists
```



```
 779                  * and it will be OK anyway.
 780                  */

 782                 kseg.s_as = &kas;
 783                 red_pp = page_create_va(&kvp, (u_offset_t)(uintptr_t)red_va,
 784                     PAGESIZE, PG_WAIT | PG_EXCL, &kseg, red_va);
 785                 ASSERT(red_pp != NULL);

 787                 /*
 788                  * So we now have a page to jam into the redzone...
 789                  */
 790                 page_io_unlock(red_pp);

 792                 hat_memload(kas.a_hat, red_va, red_pp,
 793                     (PROT_READ|PROT_WRITE), HAT_LOAD_LOCK);
 794                 page_downgrade(red_pp);

 796                 /*
 797                  * The page is left SE_SHARED locked so we can hold on to
 798                  * the page_t pointer.
 799                  */
 800                 curthread->t_red_pp = red_pp;

 802                 atomic_inc_32(&red_nmapped);
 803                 while (fp - (uintptr_t)curthread->t_stkbase < red_closest) {
 804                         (void) atomic_cas_32(&red_closest, red_closest,
 805                             (uint32_t)(fp - (uintptr_t)curthread->t_stkbase));
 806                 }
 807                 return (1);
 808         }

 810         stkbase = (caddr_t)(((uintptr_t)curthread->t_stkbase &
 811             (uintptr_t)PAGEMASK) - PAGESIZE);

 813         atomic_inc_32(&red_ndoubles);

 815         if (fp - (uintptr_t)stkbase < RED_DEEP_THRESHOLD) {
 816                 /*
 817                  * Oh boy.  We're already deep within the mapped-in
 818                  * redzone page, and the caller is trying to prepare
 819                  * for a deep stack run.  We're running without a
 820                  * redzone right now:  if the caller plows off the
 821                  * end of the stack, it'll plow another thread or
 822                  * LWP structure.  That situation could result in
 823                  * a very hard-to-debug panic, so, in the spirit of
 824                  * recording the name of one's killer in one's own
 825                  * blood, we're going to record hrestime and the calling
 826                  * thread.
 827                  */
 828                 red_deep_hires = hrestime.tv_nsec;
 829                 red_deep_thread = curthread;
 830         }

 832         /*
 833          * If this is a DEBUG kernel, and we've run too deep for comfort, toss.
 834          */
 835         ASSERT(fp - (uintptr_t)stkbase >= RED_DEEP_THRESHOLD);
 836         return (0);
 837 #endif /* _LP64 */
 838 }

 840 void
 841 segkp_unmap_red(void)
 842 {
 843         page_t *pp;
 844         caddr_t red_va = (caddr_t)(((uintptr_t)curthread->t_stkbase &
```

```
 845             (uintptr_t)PAGEMASK) - PAGESIZE);

 847         ASSERT(curthread->t_red_pp != NULL);
 887         ASSERT(curthread->t_schedflag & TS_DONT_SWAP);

 849         /*
 850          * Because we locked the mapping down, we can't simply rely
 851          * on page_destroy() to clean everything up;  we need to call
 852          * hat_unload() to explicitly unlock the mapping resources.
 853          */
 854         hat_unload(kas.a_hat, red_va, PAGESIZE, HAT_UNLOAD_UNLOCK);

 856         pp = curthread->t_red_pp;

 858         ASSERT(pp == page_find(&kvp, (u_offset_t)(uintptr_t)red_va));

 860         /*
 861          * Need to upgrade the SE_SHARED lock to SE_EXCL.
 862          */
 863         if (!page_tryupgrade(pp)) {
 864                 /*
 865                  * As there is now wait for upgrade, release the
 866                  * SE_SHARED lock and wait for SE_EXCL.
 867                  */
 868                 page_unlock(pp);
 869                 pp = page_lookup(&kvp, (u_offset_t)(uintptr_t)red_va, SE_EXCL);
 870                 /* pp may be NULL here, hence the test below */
 871         }

 873         /*
 874          * Destroy the page, with dontfree set to zero (i.e. free it).
 875          */
 876         if (pp != NULL)
 877                 page_destroy(pp, 0);
 878         curthread->t_red_pp = NULL;
 879 }
______unchanged_portion_omitted_

1354 /*ARGSUSED*/
1355 static int
1356 segkp_pagelock(struct seg *seg, caddr_t addr, size_t len,
1357     struct page ***ppp, enum lock_type type, enum seg_rw rw)
1358 {
1359         return (ENOTSUP);
1400 }

1402 /*ARGSUSED*/
1403 static int
1404 segkp_getmemid(struct seg *seg, caddr_t addr, memid_t *memidp)
1405 {
1406         return (ENODEV);
1407 }

1409 /*ARGSUSED*/
1410 static lgrp_mem_policy_info_t   *
1411 segkp_getpolicy(struct seg *seg, caddr_t addr)
1412 {
1413         return (NULL);
1414 }

1416 /*ARGSUSED*/
1417 static int
1418 segkp_capable(struct seg *seg, segcapability_t capability)
1419 {
1420         return (0);
1360 }
______unchanged_portion_omitted_
```

```
**********************************************************
    9136 Fri May  8 18:10:35 2015
new/usr/src/uts/common/vm/seg_kpm.c
use NULL dump segop as a shorthand for no-op
Instead of forcing every segment driver to implement a dummy function that
does nothing, handle NULL dump segop function pointer as a no-op shorthand.
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
use NULL setpagesize segop as a shorthand for ENOTSUP
Instead of forcing every segment driver to implement a dummp function to
return (hopefully) ENOTSUP, handle NULL setpagesize segop function pointer
as "return ENOTSUP" shorthand.
use NULL getmemid segop as a shorthand for ENODEV
Instead of forcing every segment driver to implement a dummy function to
return (hopefully) ENODEV, handle NULL getmemid segop function pointer as
"return ENODEV" shorthand.
use NULL capable segop as a shorthand for no-capabilities
Instead of forcing every segment driver to implement a dummy "return 0"
function, handle NULL capable segop function pointer as "no copabilities
supported" shorthand.
seg_inherit_notsup is redundant since segop_inherit checks for NULL properly
no need for bad-op segment op functions
The segment drivers have a number of bad-op functions that simply panic.
Keeping the function pointer NULL will accomplish the same thing in most
cases.  In other cases, keeping the function pointer NULL will result in
proper error code being returned.
use C99 initializers in segment ops structures
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
```

```
   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, Version 1.0 only
   6  * (the "License").  You may not use this file except in compliance
   7  * with the License.
   8  *
   9  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
  10  * or http://www.opensolaris.org/os/licensing.
  11  * See the License for the specific language governing permissions
  12  * and limitations under the License.
  13  *
  14  * When distributing Covered Code, include this CDDL HEADER in each
  15  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  16  * If applicable, add the following below this CDDL HEADER, with the
  17  * fields enclosed by brackets "[]" replaced with your own identifying
  18  * information: Portions Copyright [yyyy] [name of copyright owner]
  19  *
  20  * CDDL HEADER END
  21  */
  22 /*
  23  * Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
  24  * Use is subject to license terms.
  25  */

  27 /*
  28  * Kernel Physical Mapping (kpm) segment driver (segkpm).
  29  *
  30  * This driver delivers along with the hat_kpm* interfaces an alternative
  31  * mechanism for kernel mappings within the 64-bit Solaris operating system,
```



```
  32  * which allows the mapping of all physical memory into the kernel address
  33  * space at once. This is feasible in 64 bit kernels, e.g. for Ultrasparc II
  34  * and beyond processors, since the available VA range is much larger than
  35  * possible physical memory. Momentarily all physical memory is supported,
  36  * that is represented by the list of memory segments (memsegs).
  37  *
  38  * Segkpm mappings have also very low overhead and large pages are used
  39  * (when possible) to minimize the TLB and TSB footprint. It is also
  40  * extentable for other than Sparc architectures (e.g. AMD64). Main
  41  * advantage is the avoidance of the TLB-shootdown X-calls, which are
  42  * normally needed when a kernel (global) mapping has to be removed.
  43  *
  44  * First example of a kernel facility that uses the segkpm mapping scheme
  45  * is seg_map, where it is used as an alternative to hat_memload().
  46  * See also hat layer for more information about the hat_kpm* routines.
  47  * The kpm facilty can be turned off at boot time (e.g. /etc/system).
  48  */

  50 #include <sys/types.h>
  51 #include <sys/param.h>
  52 #include <sys/sysmacros.h>
  53 #include <sys/systm.h>
  54 #include <sys/vnode.h>
  55 #include <sys/cmn_err.h>
  56 #include <sys/debug.h>
  57 #include <sys/thread.h>
  58 #include <sys/cpuvar.h>
  59 #include <sys/bitmap.h>
  60 #include <sys/atomic.h>
  61 #include <sys/lgrp.h>

  63 #include <vm/seg_kmem.h>
  64 #include <vm/seg_kpm.h>
  65 #include <vm/hat.h>
  66 #include <vm/as.h>
  67 #include <vm/seg.h>
  68 #include <vm/page.h>

  70 /*
  71  * Global kpm controls.
  72  * See also platform and mmu specific controls.
  73  *
  74  * kpm_enable -- global on/off switch for segkpm.
  75  * . Set by default on 64bit platforms that have kpm support.
  76  * . Will be disabled from platform layer if not supported.
  77  * . Can be disabled via /etc/system.
  78  *
  79  * kpm_smallpages -- use only regular/system pagesize for kpm mappings.
  80  * . Can be useful for critical debugging of kpm clients.
  81  * . Set to zero by default for platforms that support kpm large pages.
  82  *   The use of kpm large pages reduces the footprint of kpm meta data
  83  *   and has all the other advantages of using large pages (e.g TLB
  84  *   miss reduction).
  85  * . Set by default for platforms that don't support kpm large pages or
  86  *   where large pages cannot be used for other reasons (e.g. there are
  87  *   only few full associative TLB entries available for large pages).
  88  *
  89  * segmap_kpm -- separate on/off switch for segmap using segkpm:
  90  * . Set by default.
  91  * . Will be disabled when kpm_enable is zero.
  92  * . Will be disabled when MAXBSIZE != PAGESIZE.
  93  * . Can be disabled via /etc/system.
  94  *
  95  */
  96 int kpm_enable = 1;
  97 int kpm_smallpages = 0;
```

```
 98 int segmap_kpm = 1;

100 /*
101  * Private seg op routines.
102  */
103 faultcode_t segkpm_fault(struct hat *hat, struct seg *seg, caddr_t addr,
104                         size_t len, enum fault_type type, enum seg_rw rw);
105 static int      segkpm_pagelock(struct seg *seg, caddr_t addr, size_t len,
106                         struct page ***page, enum lock_type type,
107                         enum seg_rw rw);

109 static const struct seg_ops segkpm_ops = {
110         .fault          = segkpm_fault,
111         .pagelock       = segkpm_pagelock,
112 //#ifndef SEGKPM_SUPPORT
113 #if 0
114 #error FIXME: define nop
115         .dup            = nop,
116         .unmap          = nop,
117         .free           = nop,
118         .faulta         = nop,
119         .setprot        = nop,
120         .checkprot      = nop,
121         .kluster        = nop,
122         .sync           = nop,
123         .incore         = nop,
124         .lockop         = nop,
125         .getprot        = nop,
126         .getoffset      = nop,
127         .gettype        = nop,
128         .getvp          = nop,
129         .advise         = nop,
130         .getpolicy      = nop,
131 #endif
105 static void     segkpm_dump(struct seg *);
106 static void     segkpm_badop(void);
107 static int      segkpm_notsup(void);
108 static int      segkpm_capable(struct seg *, segcapability_t);

110 #define SEGKPM_BADOP(t) (t(*)())segkpm_badop
111 #define SEGKPM_NOTSUP   (int(*)())segkpm_notsup

113 static struct seg_ops segkpm_ops = {
114         SEGKPM_BADOP(int),      /* dup */
115         SEGKPM_BADOP(int),      /* unmap */
116         SEGKPM_BADOP(void),     /* free */
117         segkpm_fault,
118         SEGKPM_BADOP(int),      /* faulta */
119         SEGKPM_BADOP(int),      /* setprot */
120         SEGKPM_BADOP(int),      /* checkprot */
121         SEGKPM_BADOP(int),      /* kluster */
122         SEGKPM_BADOP(size_t),   /* swapout */
123         SEGKPM_BADOP(int),      /* sync */
124         SEGKPM_BADOP(size_t),   /* incore */
125         SEGKPM_BADOP(int),      /* lockop */
126         SEGKPM_BADOP(int),      /* getprot */
127         SEGKPM_BADOP(u_offset_t), /* getoffset */
128         SEGKPM_BADOP(int),      /* gettype */
129         SEGKPM_BADOP(int),      /* getvp */
130         SEGKPM_BADOP(int),      /* advise */
131         segkpm_dump,            /* dump */
132         SEGKPM_NOTSUP,          /* pagelock */
133         SEGKPM_BADOP(int),      /* setpgsz */
134         SEGKPM_BADOP(int),      /* getmemid */
135         SEGKPM_BADOP(lgrp_mem_policy_info_t *), /* getpolicy */
136         segkpm_capable,         /* capable */
```



```
137         seg_inherit_notsup      /* inherit */
132 };
_____unchanged_portion_omitted_

287 static void
288 segkpm_badop()
289 {
290         panic("segkpm_badop");
291 }

281 #else   /* SEGKPM_SUPPORT */

283 /* segkpm stubs */

285 /*ARGSUSED*/
286 int segkpm_create(struct seg *seg, void *argsp)
287 {
288         return (0);
289 }
298 int segkpm_create(struct seg *seg, void *argsp) { return (0); }

291 /* ARGSUSED */
292 faultcode_t
293 segkpm_fault(struct hat *hat, struct seg *seg, caddr_t addr, size_t len,
294         enum fault_type type, enum seg_rw rw)
295 {
296         return (0);
305         return ((faultcode_t)0);
297 }

299 /* ARGSUSED */
300 caddr_t segkpm_create_va(u_offset_t off)
301 {
302         return (NULL);
303 }
309 caddr_t segkpm_create_va(u_offset_t off) { return (NULL); }

305 /* ARGSUSED */
306 void segkpm_mapout_validkpme(struct kpme *kpme)
307 {
308 }
312 void segkpm_mapout_validkpme(struct kpme *kpme) {}

314 static void
315 segkpm_badop() {}

310 #endif  /* SEGKPM_SUPPORT */

312 /* ARGSUSED */
313 #endif /* ! codereview */
314 static int
315 segkpm_pagelock(struct seg *seg, caddr_t addr, size_t len,
316         struct page ***page, enum lock_type type, enum seg_rw rw)
319 segkpm_notsup()
317 {
318         return (ENOTSUP);
322 }

324 /*
325  * segkpm pages are not dumped, so we just return
326  */
327 /*ARGSUSED*/
328 static void
329 segkpm_dump(struct seg *seg)
330 {}
```

```
332 /*
333  * We claim to have no special capabilities.
334  */
335 /*ARGSUSED*/
336 static int
337 segkpm_capable(struct seg *seg, segcapability_t capability)
338 {
339         return (0);
319 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   57271 Fri May  8 18:10:36 2015
new/usr/src/uts/common/vm/seg_map.c
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
use NULL capable segop as a shorthand for no-capabilities
Instead of forcing every segment driver to implement a dummy "return 0"
function, handle NULL capable segop function pointer as "no copabilities
supported" shorthand.
segop_getpolicy already checks for a NULL op
seg_inherit_notsup is redundant since segop_inherit checks for NULL properly
no need for bad-op segment op functions
The segment drivers have a number of bad-op functions that simply panic.
Keeping the function pointer NULL will accomplish the same thing in most
cases.  In other cases, keeping the function pointer NULL will result in
proper error code being returned.
use C99 initializers in segment ops structures
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
```

```
/*
 * 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.
 */

/*      Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T     */
/*        All Rights Reserved   */

/*
 * Portions of this source code were derived from Berkeley 4.3 BSD
 * under license from the Regents of the University of California.
 */

/*
 * VM - generic vnode mapping segment.
 *
 * The segmap driver is used only by the kernel to get faster (than seg_vn)
 * mappings [lower routine overhead; more persistent cache] to random
 * vnode/offsets.  Note than the kernel may (and does) use seg_vn as well.
 */
```



```
#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/buf.h>
#include <sys/systm.h>
#include <sys/vnode.h>
#include <sys/mman.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/kmem.h>
#include <sys/vtrace.h>
#include <sys/cmn_err.h>
#include <sys/debug.h>
#include <sys/thread.h>
#include <sys/dumphdr.h>
#include <sys/bitmap.h>
#include <sys/lgrp.h>

#include <vm/seg_kmem.h>
#include <vm/hat.h>
#include <vm/as.h>
#include <vm/seg.h>
#include <vm/seg_kpm.h>
#include <vm/seg_map.h>
#include <vm/page.h>
#include <vm/pvn.h>
#include <vm/rm.h>

/*
 * Private seg op routines.
 */
static void     segmap_free(struct seg *seg);
faultcode_t segmap_fault(struct hat *hat, struct seg *seg, caddr_t addr,
                        size_t len, enum fault_type type, enum seg_rw rw);
static faultcode_t segmap_faulta(struct seg *seg, caddr_t addr);
static int      segmap_checkprot(struct seg *seg, caddr_t addr, size_t len,
                        uint_t prot);
static int      segmap_kluster(struct seg *seg, caddr_t addr, ssize_t);
static int      segmap_getprot(struct seg *seg, caddr_t addr, size_t len,
                        uint_t *protv);
static u_offset_t       segmap_getoffset(struct seg *seg, caddr_t addr);
static int      segmap_gettype(struct seg *seg, caddr_t addr);
static int      segmap_getvp(struct seg *seg, caddr_t addr, struct vnode **vpp);
static void     segmap_dump(struct seg *seg);
static int      segmap_pagelock(struct seg *seg, caddr_t addr, size_t len,
                        struct page ***ppp, enum lock_type type,
                        enum seg_rw rw);
static void     segmap_badop(void);
static int      segmap_getmemid(struct seg *seg, caddr_t addr, memid_t *memidp);
static lgrp_mem_policy_info_t   *segmap_getpolicy(struct seg *seg,
    caddr_t addr);
static int      segmap_capable(struct seg *seg, segcapability_t capability);

/* segkpm support */
static caddr_t  segmap_pagecreate_kpm(struct seg *, vnode_t *, u_offset_t,
                        struct smap *, enum seg_rw);
struct smap     *get_smap_kpm(caddr_t, page_t **);

static struct seg_ops segmap_ops = {
        .free           = segmap_free,
        .fault          = segmap_fault,
        .faulta         = segmap_faulta,
        .checkprot      = segmap_checkprot,
        .kluster        = segmap_kluster,
        .getprot        = segmap_getprot,
```

```
104         .getoffset      = segmap_getoffset,
105         .gettype        = segmap_gettype,
106         .getvp          = segmap_getvp,
107         .dump           = segmap_dump,
108         .pagelock       = segmap_pagelock,
109         .getmemid       = segmap_getmemid,
101 #define SEGMAP_BADOP(t) (t(*)())segmap_badop

103 static struct seg_ops segmap_ops = {
104         SEGMAP_BADOP(int),      /* dup */
105         SEGMAP_BADOP(int),      /* unmap */
106         segmap_free,
107         segmap_fault,
108         segmap_faulta,
109         SEGMAP_BADOP(int),      /* setprot */
110         segmap_checkprot,
111         segmap_kluster,
112         SEGMAP_BADOP(size_t),   /* swapout */
113         SEGMAP_BADOP(int),      /* sync */
114         SEGMAP_BADOP(size_t),   /* incore */
115         SEGMAP_BADOP(int),      /* lockop */
116         segmap_getprot,
117         segmap_getoffset,
118         segmap_gettype,
119         segmap_getvp,
120         SEGMAP_BADOP(int),      /* advise */
121         segmap_dump,
122         segmap_pagelock,        /* pagelock */
123         SEGMAP_BADOP(int),      /* setpgsz */
124         segmap_getmemid,        /* getmemid */
125         segmap_getpolicy,       /* getpolicy */
126         segmap_capable,         /* capable */
127         seg_inherit_notsup      /* inherit */
110 };
```

_____unchanged_portion_omitted_

```
912 static void
913 segmap_badop()
914 {
915         panic("segmap_badop");
916         /*NOTREACHED*/
917 }

894 /*
895  * Special private segmap operations
896  */

898 /*
899  * Add smap to the appropriate free list.
900  */
901 static void
902 segmap_smapadd(struct smap *smp)
903 {
904         struct smfree *sm;
905         struct smap *smpfreelist;
906         struct sm_freeq *releq;

908         ASSERT(MUTEX_HELD(SMAPMTX(smp)));

910         if (smp->sm_refcnt != 0) {
911                 panic("segmap_smapadd");
912                 /*NOTREACHED*/
913         }

915         sm = &smd_free[smp->sm_free_ndx];
916         /*
```



```
917          * Add to the tail of the release queue
918          * Note that sm_releq and sm_allocq could toggle
919          * before we get the lock. This does not affect
920          * correctness as the 2 queues are only maintained
921          * to reduce lock pressure.
922          */
923         releq = sm->sm_releq;
924         if (releq == &sm->sm_freeq[0])
925                 smp->sm_flags |= SM_QNDX_ZERO;
926         else
927                 smp->sm_flags &= ~SM_QNDX_ZERO;
928         mutex_enter(&releq->smq_mtx);
929         smpfreelist = releq->smq_free;
930         if (smpfreelist == 0) {
931                 int want;

933                 releq->smq_free = smp->sm_next = smp->sm_prev = smp;
934                 /*
935                  * Both queue mutexes held to set sm_want;
936                  * snapshot the value before dropping releq mutex.
937                  * If sm_want appears after the releq mutex is dropped,
938                  * then the smap just freed is already gone.
939                  */
940                 want = sm->sm_want;
941                 mutex_exit(&releq->smq_mtx);
942                 /*
943                  * See if there was a waiter before dropping the releq mutex
944                  * then recheck after obtaining sm_freeq[0] mutex as
945                  * the another thread may have already signaled.
946                  */
947                 if (want) {
948                         mutex_enter(&sm->sm_freeq[0].smq_mtx);
949                         if (sm->sm_want)
950                                 cv_signal(&sm->sm_free_cv);
951                         mutex_exit(&sm->sm_freeq[0].smq_mtx);
952                 }
953         } else {
954                 smp->sm_next = smpfreelist;
955                 smp->sm_prev = smpfreelist->sm_prev;
956                 smpfreelist->sm_prev = smp;
957                 smp->sm_prev->sm_next = smp;
958                 mutex_exit(&releq->smq_mtx);
959         }
960 }
```

_____unchanged_portion_omitted_

```
2159 static int
2160 segmap_getmemid(struct seg *seg, caddr_t addr, memid_t *memidp)
2161 {
2162         struct segmap_data *smd = (struct segmap_data *)seg->s_data;

2164         memidp->val[0] = (uintptr_t)smd->smd_sm->sm_vp;
2165         memidp->val[1] = smd->smd_sm->sm_off + (uintptr_t)(addr - seg->s_base);
2191         return (0);
2192 }

2194 /*ARGSUSED*/
2195 static lgrp_mem_policy_info_t *
2196 segmap_getpolicy(struct seg *seg, caddr_t addr)
2197 {
2198         return (NULL);
2199 }

2201 /*ARGSUSED*/
2202 static int
2203 segmap_capable(struct seg *seg, segcapability_t capability)
```

```
2204 {
2166         return (0);
2167 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   82163 Fri May  8 18:10:36 2015
new/usr/src/uts/common/vm/seg_spt.c
use NULL dump segop as a shorthand for no-op
Instead of forcing every segment driver to implement a dummy function that
does nothing, handle NULL dump segop function pointer as a no-op shorthand.
segspt_ops can be static
There is nothing that needs access to this structure outside of the spt
segment driver itself.
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
use NULL setpagesize segop as a shorthand for ENOTSUP
Instead of forcing every segment driver to implement a dummp function to
return (hopefully) ENOTSUP, handle NULL setpagesize segop function pointer
as "return ENOTSUP" shorthand.
use NULL capable segop as a shorthand for no-capabilities
Instead of forcing every segment driver to implement a dummy "return 0"
function, handle NULL capable segop function pointer as "no copabilities
supported" shorthand.
seg_inherit_notsup is redundant since segop_inherit checks for NULL properly
no need for bad-op segment op functions
The segment drivers have a number of bad-op functions that simply panic.
Keeping the function pointer NULL will accomplish the same thing in most
cases.  In other cases, keeping the function pointer NULL will result in
proper error code being returned.
use C99 initializers in segment ops structures
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
remove xhat
The xhat infrastructure was added to support hardware such as the zulu
graphics card - hardware which had on-board MMUs.  The VM used the xhat code
to keep the CPU's and Zulu's page tables in-sync.  Since the only xhat user
was zulu (which is gone), we can safely remove it simplifying the whole VM
subsystem.
Assorted notes:
- AS_BUSY flag was used solely by xhat
**********************************************************
```

```
/*
 * 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) 1993, 2010, Oracle and/or its affiliates. All rights reserved.
 */
```



```
#include <sys/param.h>
#include <sys/user.h>
#include <sys/mman.h>
#include <sys/kmem.h>
#include <sys/sysmacros.h>
#include <sys/cmn_err.h>
#include <sys/systm.h>
#include <sys/tuneable.h>
#include <vm/hat.h>
#include <vm/seg.h>
#include <vm/as.h>
#include <vm/anon.h>
#include <vm/page.h>
#include <sys/buf.h>
#include <sys/swap.h>
#include <sys/atomic.h>
#include <vm/seg_spt.h>
#include <sys/debug.h>
#include <sys/vtrace.h>
#include <sys/shm.h>
#include <sys/shm_impl.h>
#include <sys/lgrp.h>
#include <sys/vmsystm.h>
#include <sys/policy.h>
#include <sys/project.h>
#include <sys/tnf_probe.h>
#include <sys/zone.h>

#define SEGSPTADDR      (caddr_t)0x0

/*
 * # pages used for spt
 */
size_t  spt_used;

/*
 * segspt_minfree is the memory left for system after ISM
 * locked its pages; it is set up to 5% of availrmem in
 * sptcreate when ISM is created.  ISM should not use more
 * than ~90% of availrmem; if it does, then the performance
 * of the system may decrease. Machines with large memories may
 * be able to use up more memory for ISM so we set the default
 * segspt_minfree to 5% (which gives ISM max 95% of availrmem.
 * If somebody wants even more memory for ISM (risking hanging
 * the system) they can patch the segspt_minfree to smaller number.
 */
pgcnt_t segspt_minfree = 0;

static int segspt_create(struct seg *seg, caddr_t argsp);
static int segspt_unmap(struct seg *seg, caddr_t raddr, size_t ssize);
static void segspt_free(struct seg *seg);
static void segspt_free_pages(struct seg *seg, caddr_t addr, size_t len);
static lgrp_mem_policy_info_t *segspt_getpolicy(struct seg *seg, caddr_t addr);

static const struct seg_ops segspt_ops = {
        .unmap          = segspt_unmap,
        .free           = segspt_free,
        .getpolicy      = segspt_getpolicy,
static void
segspt_badop()
{
        panic("segspt_badop called");
        /*NOTREACHED*/
}

#define SEGSPT_BADOP(t) (t(*)())segspt_badop
```

```
 88 struct seg_ops segspt_ops = {
 89         SEGSPT_BADOP(int),              /* dup */
 90         segspt_unmap,
 91         segspt_free,
 92         SEGSPT_BADOP(int),              /* fault */
 93         SEGSPT_BADOP(faultcode_t),      /* faulta */
 94         SEGSPT_BADOP(int),              /* setprot */
 95         SEGSPT_BADOP(int),              /* checkprot */
 96         SEGSPT_BADOP(int),              /* kluster */
 97         SEGSPT_BADOP(size_t),           /* swapout */
 98         SEGSPT_BADOP(int),              /* sync */
 99         SEGSPT_BADOP(size_t),           /* incore */
100         SEGSPT_BADOP(int),              /* lockop */
101         SEGSPT_BADOP(int),              /* getprot */
102         SEGSPT_BADOP(u_offset_t),       /* getoffset */
103         SEGSPT_BADOP(int),              /* gettype */
104         SEGSPT_BADOP(int),              /* getvp */
105         SEGSPT_BADOP(int),              /* advise */
106         SEGSPT_BADOP(void),             /* dump */
107         SEGSPT_BADOP(int),              /* pagelock */
108         SEGSPT_BADOP(int),              /* setpgsz */
109         SEGSPT_BADOP(int),              /* getmemid */
110         segspt_getpolicy,               /* getpolicy */
111         SEGSPT_BADOP(int),              /* capable */
112         seg_inherit_notsup              /* inherit */
 83 };

 85 static int segspt_shmdup(struct seg *seg, struct seg *newseg);
 86 static int segspt_shmunmap(struct seg *seg, caddr_t raddr, size_t ssize);
 87 static void segspt_shmfree(struct seg *seg);
 88 static faultcode_t segspt_shmfault(struct hat *hat, struct seg *seg,
 89                 caddr_t addr, size_t len, enum fault_type type, enum seg_rw rw);
 90 static faultcode_t segspt_shmfaulta(struct seg *seg, caddr_t addr);
 91 static int segspt_shmsetprot(register struct seg *seg, register caddr_t addr,
 92                         register size_t len, register uint_t prot);
 93 static int segspt_shmcheckprot(struct seg *seg, caddr_t addr, size_t size,
 94                         uint_t prot);
 95 static int      segspt_shmkluster(struct seg *seg, caddr_t addr, ssize_t delta);
126 static size_t   segspt_shmswapout(struct seg *seg);
 96 static size_t segspt_shmincore(struct seg *seg, caddr_t addr, size_t len,
 97                         register char *vec);
 98 static int segspt_shmsync(struct seg *seg, register caddr_t addr, size_t len,
 99                         int attr, uint_t flags);
100 static int segspt_shmlockop(struct seg *seg, caddr_t addr, size_t len,
101                         int attr, int op, ulong_t *lockmap, size_t pos);
102 static int segspt_shmgetprot(struct seg *seg, caddr_t addr, size_t len,
103                         uint_t *protv);
104 static u_offset_t segspt_shmgetoffset(struct seg *seg, caddr_t addr);
105 static int segspt_shmgettype(struct seg *seg, caddr_t addr);
106 static int segspt_shmgetvp(struct seg *seg, caddr_t addr, struct vnode **vpp);
107 static int segspt_shmadvise(struct seg *seg, caddr_t addr, size_t len,
108                         uint_t behav);
140 static void segspt_shmdump(struct seg *seg);
109 static int segspt_shmpagelock(struct seg *, caddr_t, size_t,
110                         struct page ***, enum lock_type, enum seg_rw);
143 static int segspt_shmsetpgsz(struct seg *, caddr_t, size_t, uint_t);
111 static int segspt_shmgetmemid(struct seg *, caddr_t, memid_t *);
112 static lgrp_mem_policy_info_t *segspt_shmgetpolicy(struct seg *, caddr_t);
146 static int segspt_shmcapable(struct seg *, segcapability_t);

114 const struct seg_ops segspt_shmops = {
115         .dup            = segspt_shmdup,
116         .unmap          = segspt_shmunmap,
117         .free           = segspt_shmfree,
118         .fault          = segspt_shmfault,
```



```
119         .faulta         = segspt_shmfaulta,
120         .setprot        = segspt_shmsetprot,
121         .checkprot      = segspt_shmcheckprot,
122         .kluster        = segspt_shmkluster,
123         .sync           = segspt_shmsync,
124         .incore         = segspt_shmincore,
125         .lockop         = segspt_shmlockop,
126         .getprot        = segspt_shmgetprot,
127         .getoffset      = segspt_shmgetoffset,
128         .gettype        = segspt_shmgettype,
129         .getvp          = segspt_shmgetvp,
130         .advise         = segspt_shmadvise,
131         .pagelock       = segspt_shmpagelock,
132         .getmemid       = segspt_shmgetmemid,
133         .getpolicy      = segspt_shmgetpolicy,
148 struct seg_ops segspt_shmops = {
149         segspt_shmdup,
150         segspt_shmunmap,
151         segspt_shmfree,
152         segspt_shmfault,
153         segspt_shmfaulta,
154         segspt_shmsetprot,
155         segspt_shmcheckprot,
156         segspt_shmkluster,
157         segspt_shmswapout,
158         segspt_shmsync,
159         segspt_shmincore,
160         segspt_shmlockop,
161         segspt_shmgetprot,
162         segspt_shmgetoffset,
163         segspt_shmgettype,
164         segspt_shmgetvp,
165         segspt_shmadvise,       /* advise */
166         segspt_shmdump,
167         segspt_shmpagelock,
168         segspt_shmsetpgsz,
169         segspt_shmgetmemid,
170         segspt_shmgetpolicy,
171         segspt_shmcapable,
172         seg_inherit_notsup
134 };
______unchanged_portion_omitted_


1780 faultcode_t
1781 segspt_dismfault(struct hat *hat, struct seg *seg, caddr_t addr,
1782     size_t len, enum fault_type type, enum seg_rw rw)
1783 {
1784         struct  shm_data        *shmd = (struct shm_data *)seg->s_data;
1785         struct  seg             *sptseg = shmd->shm_sptseg;
1786         struct  as              *curspt = shmd->shm_sptas;
1787         struct  spt_data        *sptd = sptseg->s_data;
1788         pgcnt_t npages;
1789         size_t  size;
1790         caddr_t segspt_addr, shm_addr;
1791         page_t  **ppa;
1792         int     i;
1793         ulong_t an_idx = 0;
1794         int     err = 0;
1795         int     dyn_ism_unmap = hat_supported(HAT_DYNAMIC_ISM_UNMAP, (void *)0);
1796         size_t  pgsz;
1797         pgcnt_t pgcnt;
1798         caddr_t a;
1799         pgcnt_t pidx;

1801 #ifdef lint
```

```
1802         hat = hat;
1803 #endif
1804         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

1806         /*
1807          * Because of the way spt is implemented
1808          * the realsize of the segment does not have to be
1809          * equal to the segment size itself. The segment size is
1810          * often in multiples of a page size larger than PAGESIZE.
1811          * The realsize is rounded up to the nearest PAGESIZE
1812          * based on what the user requested. This is a bit of
1813          * ungliness that is historical but not easily fixed
1814          * without re-designing the higher levels of ISM.
1815          */
1816         ASSERT(addr >= seg->s_base);
1817         if (((addr + len) - seg->s_base) > sptd->spt_realsize)
1818                 return (FC_NOMAP);
1819         /*
1820          * For all of the following cases except F_PROT, we need to
1821          * make any necessary adjustments to addr and len
1822          * and get all of the necessary page_t's into an array called ppa[].
1823          *
1824          * The code in shmat() forces base addr and len of ISM segment
1825          * to be aligned to largest page size supported. Therefore,
1826          * we are able to handle F_SOFTLOCK and F_INVAL calls in "large
1827          * pagesize" chunks. We want to make sure that we HAT_LOAD_LOCK
1828          * in large pagesize chunks, or else we will screw up the HAT
1829          * layer by calling hat_memload_array() with differing page sizes
1830          * over a given virtual range.
1831          */
1832         pgsz = page_get_pagesize(sptseg->s_szc);
1833         pgcnt = page_get_pagecnt(sptseg->s_szc);
1834         shm_addr = (caddr_t)P2ALIGN((uintptr_t)(addr), pgsz);
1835         size = P2ROUNDUP((uintptr_t)(((addr + len) - shm_addr)), pgsz);
1836         npages = btopr(size);

1838         /*
1839          * Now we need to convert from addr in segshm to addr in segspt.
1840          */
1841         an_idx = seg_page(seg, shm_addr);
1842         segspt_addr = sptseg->s_base + ptob(an_idx);

1844         ASSERT((segspt_addr + ptob(npages)) <=
1845             (sptseg->s_base + sptd->spt_realsize));
1846         ASSERT(segspt_addr < (sptseg->s_base + sptseg->s_size));

1848         switch (type) {

1850         case F_SOFTLOCK:

1852                 atomic_add_long((ulong_t *)(&(shmd->shm_softlockcnt)), npages);
1853                 /*
1854                  * Fall through to the F_INVAL case to load up the hat layer
1855                  * entries with the HAT_LOAD_LOCK flag.
1856                  */
1857                 /* FALLTHRU */
1858         case F_INVAL:

1860                 if ((rw == S_EXEC) && !(sptd->spt_prot & PROT_EXEC))
1861                         return (FC_NOMAP);

1863                 ppa = kmem_zalloc(npages * sizeof (page_t *), KM_SLEEP);

1865                 err = spt_anon_getpages(sptseg, segspt_addr, size, ppa);
1866                 if (err != 0) {
1867                         if (type == F_SOFTLOCK) {
```



```
1868                                 atomic_add_long((ulong_t *)(
1869                                     &(shmd->shm_softlockcnt)), -npages);
1870                         }
1871                         goto dism_err;
1872                 }
1873                 AS_LOCK_ENTER(sptseg->s_as, &sptseg->s_as->a_lock, RW_READER);
1874                 a = segspt_addr;
1875                 pidx = 0;
1876                 if (type == F_SOFTLOCK) {

1878                         /*
1879                          * Load up the translation keeping it
1880                          * locked and don't unlock the page.
1881                          */
1882                         for (; pidx < npages; a += pgsz, pidx += pgcnt) {
1883                                 hat_memload_array(sptseg->s_as->a_hat,
1884                                     a, pgsz, &ppa[pidx], sptd->spt_prot,
1885                                     HAT_LOAD_LOCK | HAT_LOAD_SHARE);
1886                         }
1887                 } else {
1927                         if (hat == seg->s_as->a_hat) {

1888                         /*
1889                          * Migrate pages marked for migration
1890                          */
1891                         if (lgrp_optimizations())
1892                                 page_migrate(seg, shm_addr, ppa, npages);
1933                                         page_migrate(seg, shm_addr, ppa,
1934                                             npages);

1894                         for (; pidx < npages; a += pgsz, pidx += pgcnt) {
1936                                 /* CPU HAT */
1937                                 for (; pidx < npages;
1938                                     a += pgsz, pidx += pgcnt) {
1895                                 hat_memload_array(sptseg->s_as->a_hat,
1896                                     a, pgsz, &ppa[pidx],
1897                                     sptd->spt_prot,
1898                                     HAT_LOAD_SHARE);
1899                         }
1944                         } else {
1945                                 /* XHAT. Pass real address */
1946                                 hat_memload_array(hat, shm_addr,
1947                                     size, ppa, sptd->spt_prot, HAT_LOAD_SHARE);
1948                         }

1901                         /*
1902                          * And now drop the SE_SHARED lock(s).
1903                          */
1904                         if (dyn_ism_unmap) {
1905                                 for (i = 0; i < npages; i++) {
1906                                         page_unlock(ppa[i]);
1907                                 }
1908                         }
1909                 }

1911                 if (!dyn_ism_unmap) {
1912                         if (hat_share(seg->s_as->a_hat, shm_addr,
1913                             curspt->a_hat, segspt_addr, ptob(npages),
1914                             seg->s_szc) != 0) {
1915                                 panic("hat_share err in DISM fault");
1916                                 /* NOTREACHED */
1917                         }
1918                         if (type == F_INVAL) {
1919                                 for (i = 0; i < npages; i++) {
1920                                         page_unlock(ppa[i]);
1921                                 }
```

```
1922                        }
1923                }
1924                AS_LOCK_EXIT(sptseg->s_as, &sptseg->s_as->a_lock);
1925 dism_err:
1926                kmem_free(ppa, npages * sizeof (page_t *));
1927                return (err);

1929        case F_SOFTUNLOCK:

1931                /*
1932                 * This is a bit ugly, we pass in the real seg pointer,
1933                 * but the segspt_addr is the virtual address within the
1934                 * dummy seg.
1935                 */
1936                segspt_softunlock(seg, segspt_addr, size, rw);
1937                return (0);

1939        case F_PROT:

1941                /*
1942                 * This takes care of the unusual case where a user
1943                 * allocates a stack in shared memory and a register
1944                 * window overflow is written to that stack page before
1945                 * it is otherwise modified.
1946                 *
1947                 * We can get away with this because ISM segments are
1948                 * always rw. Other than this unusual case, there
1949                 * should be no instances of protection violations.
1950                 */
1951                return (0);

1953        default:
1954 #ifdef DEBUG
1955                panic("segspt_dismfault default type?");
1956 #else
1957                return (FC_NOMAP);
1958 #endif
1959        }
1960 }


1963 faultcode_t
1964 segspt_shmfault(struct hat *hat, struct seg *seg, caddr_t addr,
1965     size_t len, enum fault_type type, enum seg_rw rw)
1966 {
1967        struct shm_data         *shmd = (struct shm_data *)seg->s_data;
1968        struct seg              *sptseg = shmd->shm_sptseg;
1969        struct as               *curspt = shmd->shm_sptas;
1970        struct spt_data         *sptd   = sptseg->s_data;
1971        pgcnt_t npages;
1972        size_t size;
1973        caddr_t sptseg_addr, shm_addr;
1974        page_t *pp, **ppa;
1975        int     i;
1976        u_offset_t offset;
1977        ulong_t anon_index = 0;
1978        struct vnode *vp;
1979        struct anon_map *amp;           /* XXX - for locknest */
1980        struct anon *ap = NULL;
1981        size_t          pgsz;
1982        pgcnt_t         pgcnt;
1983        caddr_t         a;
1984        pgcnt_t         pidx;
1985        size_t          sz;

1987 #ifdef lint
```



```
1988        hat = hat;
1989 #endif

1991        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

1993        if (sptd->spt_flags & SHM_PAGEABLE) {
1994                return (segspt_dismfault(hat, seg, addr, len, type, rw));
1995        }

1997        /*
1998         * Because of the way spt is implemented
1999         * the realsize of the segment does not have to be
2000         * equal to the segment size itself. The segment size is
2001         * often in multiples of a page size larger than PAGESIZE.
2002         * The realsize is rounded up to the nearest PAGESIZE
2003         * based on what the user requested. This is a bit of
2004         * ungliness that is historical but not easily fixed
2005         * without re-designing the higher levels of ISM.
2006         */
2007        ASSERT(addr >= seg->s_base);
2008        if (((addr + len) - seg->s_base) > sptd->spt_realsize)
2009                return (FC_NOMAP);
2010        /*
2011         * For all of the following cases except F_PROT, we need to
2012         * make any necessary adjustments to addr and len
2013         * and get all of the necessary page_t's into an array called ppa[].
2014         *
2015         * The code in shmat() forces base addr and len of ISM segment
2016         * to be aligned to largest page size supported. Therefore,
2017         * we are able to handle F_SOFTLOCK and F_INVAL calls in "large
2018         * pagesize" chunks. We want to make sure that we HAT_LOAD_LOCK
2019         * in large pagesize chunks, or else we will screw up the HAT
2020         * layer by calling hat_memload_array() with differing page sizes
2021         * over a given virtual range.
2022         */
2023        pgsz = page_get_pagesize(sptseg->s_szc);
2024        pgcnt = page_get_pagecnt(sptseg->s_szc);
2025        shm_addr = (caddr_t)P2ALIGN((uintptr_t)(addr), pgsz);
2026        size = P2ROUNDUP((uintptr_t)(((addr + len) - shm_addr)), pgsz);
2027        npages = btopr(size);

2029        /*
2030         * Now we need to convert from addr in segshm to addr in segspt.
2031         */
2032        anon_index = seg_page(seg, shm_addr);
2033        sptseg_addr = sptseg->s_base + ptob(anon_index);

2035        /*
2036         * And now we may have to adjust npages downward if we have
2037         * exceeded the realsize of the segment or initial anon
2038         * allocations.
2039         */
2040        if ((sptseg_addr + ptob(npages)) >
2041            (sptseg->s_base + sptd->spt_realsize))
2042                size = (sptseg->s_base + sptd->spt_realsize) - sptseg_addr;

2044        npages = btopr(size);

2046        ASSERT(sptseg_addr < (sptseg->s_base + sptseg->s_size));
2047        ASSERT((sptd->spt_flags & SHM_PAGEABLE) == 0);

2049        switch (type) {

2051        case F_SOFTLOCK:

2053                /*
```

```
2054                 * availrmem is decremented once during anon_swap_adjust()
2055                 * and is incremented during the anon_unresv(), which is
2056                 * called from shm_rm_amp() when the segment is destroyed.
2057                 */
2058                atomic_add_long((ulong_t *)(&(shmd->shm_softlockcnt)), npages);
2059                /*
2060                 * Some platforms assume that ISM pages are SE_SHARED
2061                 * locked for the entire life of the segment.
2062                 */
2063                if (!hat_supported(HAT_DYNAMIC_ISM_UNMAP, (void *)0))
2064                        return (0);
2065                /*
2066                 * Fall through to the F_INVAL case to load up the hat layer
2067                 * entries with the HAT_LOAD_LOCK flag.
2068                 */

2070                /* FALLTHRU */
2071        case F_INVAL:

2073                if ((rw == S_EXEC) && !(sptd->spt_prot & PROT_EXEC))
2074                        return (FC_NOMAP);

2076                /*
2077                 * Some platforms that do NOT support DYNAMIC_ISM_UNMAP
2078                 * may still rely on this call to hat_share(). That
2079                 * would imply that those hat's can fault on a
2080                 * HAT_LOAD_LOCK translation, which would seem
2081                 * contradictory.
2082                 */
2083                if (!hat_supported(HAT_DYNAMIC_ISM_UNMAP, (void *)0)) {
2084                        if (hat_share(seg->s_as->a_hat, seg->s_base,
2085                            curspt->a_hat, sptseg->s_base,
2086                            sptseg->s_size, sptseg->s_szc) != 0) {
2087                                panic("hat_share error in ISM fault");
2088                                /*NOTREACHED*/
2089                        }
2090                        return (0);
2091                }
2092                ppa = kmem_zalloc(sizeof (page_t *) * npages, KM_SLEEP);

2094                /*
2095                 * I see no need to lock the real seg,
2096                 * here, because all of our work will be on the underlying
2097                 * dummy seg.
2098                 *
2099                 * sptseg_addr and npages now account for large pages.
2100                 */
2101                amp = sptd->spt_amp;
2102                ASSERT(amp != NULL);
2103                anon_index = seg_page(sptseg, sptseg_addr);

2105                ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
2106                for (i = 0; i < npages; i++) {
2107                        ap = anon_get_ptr(amp->ahp, anon_index++);
2108                        ASSERT(ap != NULL);
2109                        swap_xlate(ap, &vp, &offset);
2110                        pp = page_lookup(vp, offset, SE_SHARED);
2111                        ASSERT(pp != NULL);
2112                        ppa[i] = pp;
2113                }
2114                ANON_LOCK_EXIT(&amp->a_rwlock);
2115                ASSERT(i == npages);

2117                /*
2118                 * We are already holding the as->a_lock on the user's
2119                 * real segment, but we need to hold the a_lock on the
```



```
2120                 * underlying dummy as. This is mostly to satisfy the
2121                 * underlying HAT layer.
2122                 */
2123                AS_LOCK_ENTER(sptseg->s_as, &sptseg->s_as->a_lock, RW_READER);
2124                a = sptseg_addr;
2125                pidx = 0;
2126                if (type == F_SOFTLOCK) {
2127                        /*
2128                         * Load up the translation keeping it
2129                         * locked and don't unlock the page.
2130                         */
2131                        for (; pidx < npages; a += pgsz, pidx += pgcnt) {
2132                                sz = MIN(pgsz, ptob(npages - pidx));
2133                                hat_memload_array(sptseg->s_as->a_hat, a,
2134                                    sz, &ppa[pidx], sptd->spt_prot,
2135                                    HAT_LOAD_LOCK | HAT_LOAD_SHARE);
2136                        }
2137                } else {
2187                        if (hat == seg->s_as->a_hat) {

2138                        /*
2139                         * Migrate pages marked for migration.
2140                         */
2141                        if (lgrp_optimizations())
2142                                page_migrate(seg, shm_addr, ppa, npages);
2193                                        page_migrate(seg, shm_addr, ppa,
2194                                            npages);

2144                        for (; pidx < npages; a += pgsz, pidx += pgcnt) {
2196                                /* CPU HAT */
2197                                for (; pidx < npages;
2198                                    a += pgsz, pidx += pgcnt) {
2145                                sz = MIN(pgsz, ptob(npages - pidx));
2146                                hat_memload_array(sptseg->s_as->a_hat,
2147                                    a, sz, &ppa[pidx],
2148                                    sptd->spt_prot, HAT_LOAD_SHARE);
2149                        }
2204                        } else {
2205                                /* XHAT. Pass real address */
2206                                hat_memload_array(hat, shm_addr,
2207                                    ptob(npages), ppa, sptd->spt_prot,
2208                                    HAT_LOAD_SHARE);
2209                        }

2151                        /*
2152                         * And now drop the SE_SHARED lock(s).
2153                         */
2154                        for (i = 0; i < npages; i++)
2155                                page_unlock(ppa[i]);
2156                }
2157                AS_LOCK_EXIT(sptseg->s_as, &sptseg->s_as->a_lock);

2159                kmem_free(ppa, sizeof (page_t *) * npages);
2160                return (0);
2161        case F_SOFTUNLOCK:

2163                /*
2164                 * This is a bit ugly, we pass in the real seg pointer,
2165                 * but the sptseg_addr is the virtual address within the
2166                 * dummy seg.
2167                 */
2168                segspt_softunlock(seg, sptseg_addr, ptob(npages), rw);
2169                return (0);

2171        case F_PROT:
```

```
2173                 /*
2174                  * This takes care of the unusual case where a user
2175                  * allocates a stack in shared memory and a register
2176                  * window overflow is written to that stack page before
2177                  * it is otherwise modified.
2178                  *
2179                  * We can get away with this because ISM segments are
2180                  * always rw. Other than this unusual case, there
2181                  * should be no instances of protection violations.
2182                  */
2183                 return (0);

2185         default:
2186 #ifdef DEBUG
2187                 cmn_err(CE_WARN, "segspt_shmfault default type?");
2188 #endif
2189                 return (FC_NOMAP);
2190         }
2191 }
```

_____unchanged_portion_omitted_

```
2267 /*ARGSUSED*/
2268 static size_t
2269 segspt_shmswapout(struct seg *seg)
2270 {
2271         return (0);
2272 }

2207 /*
2208  * duplicate the shared page tables
2209  */
2210 int
2211 segspt_shmdup(struct seg *seg, struct seg *newseg)
2212 {
2213         struct shm_data         *shmd = (struct shm_data *)seg->s_data;
2214         struct anon_map         *amp = shmd->shm_amp;
2215         struct shm_data         *shmd_new;
2216         struct seg              *spt_seg = shmd->shm_sptseg;
2217         struct spt_data         *sptd = spt_seg->s_data;
2218         int                     error = 0;

2220         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

2222         shmd_new = kmem_zalloc((sizeof (*shmd_new)), KM_SLEEP);
2223         newseg->s_data = (void *)shmd_new;
2224         shmd_new->shm_sptas = shmd->shm_sptas;
2225         shmd_new->shm_amp = amp;
2226         shmd_new->shm_sptseg = shmd->shm_sptseg;
2227         newseg->s_ops = &segspt_shmops;
2228         newseg->s_szc = seg->s_szc;
2229         ASSERT(seg->s_szc == shmd->shm_sptseg->s_szc);

2231         ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
2232         amp->refcnt++;
2233         ANON_LOCK_EXIT(&amp->a_rwlock);

2235         if (sptd->spt_flags & SHM_PAGEABLE) {
2236                 shmd_new->shm_vpage = kmem_zalloc(btopr(amp->size), KM_SLEEP);
2237                 shmd_new->shm_lckpgs = 0;
2238                 if (hat_supported(HAT_DYNAMIC_ISM_UNMAP, (void *)0)) {
2239                         if ((error = hat_share(newseg->s_as->a_hat,
2240                             newseg->s_base, shmd->shm_sptas->a_hat, SEGSPTADDR,
2241                             seg->s_size, seg->s_szc)) != 0) {
2242                                 kmem_free(shmd_new->shm_vpage,
2243                                     btopr(amp->size));
2244                         }
```



```
2245                 }
2246                 return (error);
2247         } else {
2248                 return (hat_share(newseg->s_as->a_hat, newseg->s_base,
2249                     shmd->shm_sptas->a_hat, SEGSPTADDR, seg->s_size,
2250                     seg->s_szc));

2252         }
2253 }
```

_____unchanged_portion_omitted_

```
3017 /*ARGSUSED*/
3018 void
3019 segspt_shmdump(struct seg *seg)
3020 {
3021         /* no-op for ISM segment */
3022 }

3024 /*ARGSUSED*/
3025 static faultcode_t
3026 segspt_shmsetpgsz(struct seg *seg, caddr_t addr, size_t len, uint_t szc)
3027 {
3028         return (ENOTSUP);
3029 }

2950 /*
2951  * get a memory ID for an addr in a given segment
2952  */
2953 static int
2954 segspt_shmgetmemid(struct seg *seg, caddr_t addr, memid_t *memidp)
2955 {
2956         struct shm_data *shmd = (struct shm_data *)seg->s_data;
2957         struct anon     *ap;
2958         size_t          anon_index;
2959         struct anon_map *amp = shmd->shm_amp;
2960         struct spt_data *sptd = shmd->shm_sptseg->s_data;
2961         struct seg      *sptseg = shmd->shm_sptseg;
2962         anon_sync_obj_t cookie;

2964         anon_index = seg_page(seg, addr);

2966         if (addr > (seg->s_base + sptd->spt_realsize)) {
2967                 return (EFAULT);
2968         }

2970         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
2971         anon_array_enter(amp, anon_index, &cookie);
2972         ap = anon_get_ptr(amp->ahp, anon_index);
2973         if (ap == NULL) {
2974                 struct page *pp;
2975                 caddr_t spt_addr = sptseg->s_base + ptob(anon_index);

2977                 pp = anon_zero(sptseg, spt_addr, &ap, kcred);
2978                 if (pp == NULL) {
2979                         anon_array_exit(&cookie);
2980                         ANON_LOCK_EXIT(&amp->a_rwlock);
2981                         return (ENOMEM);
2982                 }
2983                 (void) anon_set_ptr(amp->ahp, anon_index, ap, ANON_SLEEP);
2984                 page_unlock(pp);
2985         }
2986         anon_array_exit(&cookie);
2987         ANON_LOCK_EXIT(&amp->a_rwlock);
2988         memidp->val[0] = (uintptr_t)ap;
2989         memidp->val[1] = (uintptr_t)addr & PAGEOFFSET;
2990         return (0);
```

```
2991 }

2993 /*
2994  * Get memory allocation policy info for specified address in given segment
2995  */
2996 static lgrp_mem_policy_info_t *
2997 segspt_shmgetpolicy(struct seg *seg, caddr_t addr)
2998 {
2999         struct anon_map         *amp;
3000         ulong_t                 anon_index;
3001         lgrp_mem_policy_info_t  *policy_info;
3002         struct shm_data         *shm_data;

3004         ASSERT(seg != NULL);

3006         /*
3007          * Get anon_map from segshm
3008          *
3009          * Assume that no lock needs to be held on anon_map, since
3010          * it should be protected by its reference count which must be
3011          * nonzero for an existing segment
3012          * Need to grab readers lock on policy tree though
3013          */
3014         shm_data = (struct shm_data *)seg->s_data;
3015         if (shm_data == NULL)
3016                 return (NULL);
3017         amp = shm_data->shm_amp;
3018         ASSERT(amp->refcnt != 0);

3020         /*
3021          * Get policy info
3022          *
3023          * Assume starting anon index of 0
3024          */
3025         anon_index = seg_page(seg, addr);
3026         policy_info = lgrp_shm_policy_get(amp, anon_index, NULL, 0);

3028         return (policy_info);
3110 }

3112 /*ARGSUSED*/
3113 static int
3114 segspt_shmcapable(struct seg *seg, segcapability_t capability)
3115 {
3116         return (0);
3029 }
```

_____unchanged_portion_omitted_

```
**********************************************************
  280113 Fri May  8 18:10:36 2015
new/usr/src/uts/common/vm/seg_vn.c
PVN_GETPAGE_{SZ,NUM} are misnamed and unnecessarily complicated
There is really no reason to not allow 8 pages all the time.  With the
current logic, we get the following:
Assuming 4kB pages (x86):
    _SZ  = ptob(8) /* 32kB */
    _NUM = 8
Assuming 8kB pages (sparc):
    _SZ  = ptob(8) /* 64kB */
    _NUM = 8
We'd have to deal with 16kB base pages in order for the _NUM #define to not
be 8 (it'd be 4 in that case).  So, in the spirit of simplicity, let's just
always grab 8 pages as there are no interesting systems with 16kB+ base pages.
Finally, the defines are poorly named.
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
use NULL capable segop as a shorthand for no-capabilities
Instead of forcing every segment driver to implement a dummy "return 0"
function, handle NULL capable segop function pointer as "no copabilities
supported" shorthand.
patch lower-case-segops
use C99 initializers in segment ops structures
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
remove xhat
The xhat infrastructure was added to support hardware such as the zulu
graphics card - hardware which had on-board MMUs.  The VM used the xhat code
to keep the CPU's and Zulu's page tables in-sync.  Since the only xhat user
was zulu (which is gone), we can safely remove it simplifying the whole VM
subsystem.
Assorted notes:
- AS_BUSY flag was used solely by xhat
**********************************************************
```

```
/*
 * 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) 1986, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2015, Joyent, Inc. All rights reserved.
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
 */
```



```
/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */

/*
 * VM - shared or copy-on-write from a vnode/anonymous memory.
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/t_lock.h>
#include <sys/errno.h>
#include <sys/systm.h>
#include <sys/mman.h>
#include <sys/debug.h>
#include <sys/cred.h>
#include <sys/vmsystm.h>
#include <sys/tuneable.h>
#include <sys/bitmap.h>
#include <sys/swap.h>
#include <sys/kmem.h>
#include <sys/sysmacros.h>
#include <sys/vtrace.h>
#include <sys/cmn_err.h>
#include <sys/callb.h>
#include <sys/vm.h>
#include <sys/dumphdr.h>
#include <sys/lgrp.h>

#include <vm/hat.h>
#include <vm/as.h>
#include <vm/seg.h>
#include <vm/seg_vn.h>
#include <vm/pvn.h>
#include <vm/anon.h>
#include <vm/page.h>
#include <vm/vpage.h>
#include <sys/proc.h>
#include <sys/task.h>
#include <sys/project.h>
#include <sys/zone.h>
#include <sys/shm_impl.h>

/*
 * segvn_fault needs a temporary page list array.  To avoid calling kmem all
 * the time, it creates a small (FAULT_TMP_PAGES_NUM entry) array and uses
 * it if it can.  In the rare case when this page list is not large enough,
 * it goes and gets a large enough array from kmem.
 */
#define FAULT_TMP_PAGES_NUM     0x8
#define FAULT_TMP_PAGES_SZ      ptob(FAULT_TMP_PAGES_NUM)
 * the time, it creates a small (PVN_GETPAGE_NUM entry) array and uses it if
 * it can.  In the rare case when this page list is not large enough, it
 * goes and gets a large enough array from kmem.
 *
 * This small page list array covers either 8 pages or 64kB worth of pages -
 * whichever is smaller.
```

```
 87  */
 88 #define PVN_MAX_GETPAGE_SZ      0x10000
 89 #define PVN_MAX_GETPAGE_NUM     0x8

 91 #if PVN_MAX_GETPAGE_SZ > PVN_MAX_GETPAGE_NUM * PAGESIZE
 92 #define PVN_GETPAGE_SZ  ptob(PVN_MAX_GETPAGE_NUM)
 93 #define PVN_GETPAGE_NUM PVN_MAX_GETPAGE_NUM
 94 #else
 95 #define PVN_GETPAGE_SZ  PVN_MAX_GETPAGE_SZ
 96 #define PVN_GETPAGE_NUM btop(PVN_MAX_GETPAGE_SZ)
 97 #endif

 88 /*
 89  * Private seg op routines.
 90  */
 91 static int      segvn_dup(struct seg *seg, struct seg *newseg);
 92 static int      segvn_unmap(struct seg *seg, caddr_t addr, size_t len);
 93 static void     segvn_free(struct seg *seg);
 94 static faultcode_t segvn_fault(struct hat *hat, struct seg *seg,
 95                     caddr_t addr, size_t len, enum fault_type type,
 96                     enum seg_rw rw);
 97 static faultcode_t segvn_faulta(struct seg *seg, caddr_t addr);
 98 static int      segvn_setprot(struct seg *seg, caddr_t addr,
 99                     size_t len, uint_t prot);
100 static int      segvn_checkprot(struct seg *seg, caddr_t addr,
101                     size_t len, uint_t prot);
102 static int      segvn_kluster(struct seg *seg, caddr_t addr, ssize_t delta);
114 static size_t   segvn_swapout(struct seg *seg);
103 static int      segvn_sync(struct seg *seg, caddr_t addr, size_t len,
104                     int attr, uint_t flags);
105 static size_t   segvn_incore(struct seg *seg, caddr_t addr, size_t len,
106                     char *vec);
107 static int      segvn_lockop(struct seg *seg, caddr_t addr, size_t len,
108                     int attr, int op, ulong_t *lockmap, size_t pos);
109 static int      segvn_getprot(struct seg *seg, caddr_t addr, size_t len,
110                     uint_t *protv);
111 static u_offset_t       segvn_getoffset(struct seg *seg, caddr_t addr);
112 static int      segvn_gettype(struct seg *seg, caddr_t addr);
113 static int      segvn_getvp(struct seg *seg, caddr_t addr,
114                     struct vnode **vpp);
115 static int      segvn_advise(struct seg *seg, caddr_t addr, size_t len,
116                     uint_t behav);
117 static void     segvn_dump(struct seg *seg);
118 static int      segvn_pagelock(struct seg *seg, caddr_t addr, size_t len,
119                     struct page ***ppp, enum lock_type type, enum seg_rw rw);
120 static int      segvn_setpagesize(struct seg *seg, caddr_t addr, size_t len,
121                     uint_t szc);
122 static int      segvn_getmemid(struct seg *seg, caddr_t addr,
123                     memid_t *memidp);
124 static lgrp_mem_policy_info_t   *segvn_getpolicy(struct seg *, caddr_t);
137 static int      segvn_capable(struct seg *seg, segcapability_t capable);
125 static int      segvn_inherit(struct seg *, caddr_t, size_t, uint_t);

127 const struct seg_ops segvn_ops = {
128         .dup            = segvn_dup,
129         .unmap          = segvn_unmap,
130         .free           = segvn_free,
131         .fault          = segvn_fault,
132         .faulta         = segvn_faulta,
133         .setprot        = segvn_setprot,
134         .checkprot      = segvn_checkprot,
135         .kluster        = segvn_kluster,
136         .sync           = segvn_sync,
137         .incore         = segvn_incore,
138         .lockop         = segvn_lockop,
139         .getprot        = segvn_getprot,
```



```
140         .getoffset      = segvn_getoffset,
141         .gettype        = segvn_gettype,
142         .getvp          = segvn_getvp,
143         .advise         = segvn_advise,
144         .dump           = segvn_dump,
145         .pagelock       = segvn_pagelock,
146         .setpagesize    = segvn_setpagesize,
147         .getmemid       = segvn_getmemid,
148         .getpolicy      = segvn_getpolicy,
149         .inherit        = segvn_inherit,
140 struct  seg_ops segvn_ops = {
141         segvn_dup,
142         segvn_unmap,
143         segvn_free,
144         segvn_fault,
145         segvn_faulta,
146         segvn_setprot,
147         segvn_checkprot,
148         segvn_kluster,
149         segvn_swapout,
150         segvn_sync,
151         segvn_incore,
152         segvn_lockop,
153         segvn_getprot,
154         segvn_getoffset,
155         segvn_gettype,
156         segvn_getvp,
157         segvn_advise,
158         segvn_dump,
159         segvn_pagelock,
160         segvn_setpagesize,
161         segvn_getmemid,
162         segvn_getpolicy,
163         segvn_capable,
164         segvn_inherit
150 };
```

_____***unchanged_portion_omitted_***

```
3768 int segvn_anypgsz = 0;

3770 #define SEGVN_RESTORE_SOFTLOCK_VP(type, pages)                          \
3771                 if ((type) == F_SOFTLOCK) {                             \
3772                         atomic_add_long((ulong_t *)&(svd)->softlockcnt, \
3773                             -(pages));                                  \
3774                 }

3776 #define SEGVN_UPDATE_MODBITS(ppa, pages, rw, prot, vpprot)              \
3777                 if (IS_VMODSORT((ppa)[0]->p_vnode)) {                   \
3778                         if ((rw) == S_WRITE) {                          \
3779                                 for (i = 0; i < (pages); i++) {         \
3780                                         ASSERT((ppa)[i]->p_vnode ==     \
3781                                             (ppa)[0]->p_vnode);         \
3782                                         hat_setmod((ppa)[i]);           \
3783                                 }                                       \
3784                         } else if ((rw) != S_OTHER &&                   \
3785                             ((prot) & (vpprot) & PROT_WRITE)) {         \
3786                                 for (i = 0; i < (pages); i++) {         \
3787                                         ASSERT((ppa)[i]->p_vnode ==     \
3788                                             (ppa)[0]->p_vnode);         \
3789                                         if (!hat_ismod((ppa)[i])) {     \
3790                                                 prot &= ~PROT_WRITE;    \
3791                                                 break;                  \
3792                                         }                               \
3793                                 }                                       \
3794                         }                                               \
3795                 }
```

```
3797 #ifdef  VM_STATS

3799 #define SEGVN_VMSTAT_FLTVNPAGES(idx)                                    \
3800                 VM_STAT_ADD(segvnvmstats.fltvnpages[(idx)]);

3802 #else /* VM_STATS */

3804 #define SEGVN_VMSTAT_FLTVNPAGES(idx)

3806 #endif

3808 static faultcode_t
3809 segvn_fault_vnodepages(struct hat *hat, struct seg *seg, caddr_t lpgaddr,
3810     caddr_t lpgeaddr, enum fault_type type, enum seg_rw rw, caddr_t addr,
3811     caddr_t eaddr, int brkcow)
3812 {
3813         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
3814         struct anon_map *amp = svd->amp;
3815         uchar_t segtype = svd->type;
3816         uint_t szc = seg->s_szc;
3817         size_t pgsz = page_get_pagesize(szc);
3818         size_t maxpgsz = pgsz;
3819         pgcnt_t pages = btop(pgsz);
3820         pgcnt_t maxpages = pages;
3821         size_t ppasize = (pages + 1) * sizeof (page_t *);
3822         caddr_t a = lpgaddr;
3823         caddr_t maxlpgeaddr = lpgeaddr;
3824         u_offset_t off = svd->offset + (uintptr_t)(a - seg->s_base);
3825         ulong_t aindx = svd->anon_index + seg_page(seg, a);
3826         struct vpage *vpage = (svd->vpage != NULL) ?
3827             &svd->vpage[seg_page(seg, a)] : NULL;
3828         vnode_t *vp = svd->vp;
3829         page_t **ppa;
3830         uint_t  pszc;
3831         size_t  ppgsz;
3832         pgcnt_t ppages;
3833         faultcode_t err = 0;
3834         int ierr;
3835         int vop_size_err = 0;
3836         uint_t protchk, prot, vpprot;
3837         ulong_t i;
3838         int hat_flag = (type == F_SOFTLOCK) ? HAT_LOAD_LOCK : HAT_LOAD;
3839         anon_sync_obj_t an_cookie;
3840         enum seg_rw arw;
3841         int alloc_failed = 0;
3842         int adjszc_chk;
3843         struct vattr va;
3859         int xhat = 0;
3844         page_t *pplist;
3845         pfn_t pfn;
3846         int physcontig;
3847         int upgrdfail;
3848         int segvn_anypgsz_vnode = 0; /* for now map vnode with 2 page sizes */
3849         int tron = (svd->tr_state == SEGVN_TR_ON);

3851         ASSERT(szc != 0);
3852         ASSERT(vp != NULL);
3853         ASSERT(brkcow == 0 || amp != NULL);
3854         ASSERT(tron == 0 || amp != NULL);
3855         ASSERT(enable_mbit_wa == 0); /* no mbit simulations with large pages */
3856         ASSERT(!(svd->flags & MAP_NORESERVE));
3857         ASSERT(type != F_SOFTUNLOCK);
3858         ASSERT(IS_P2ALIGNED(a, maxpgsz));
3859         ASSERT(amp == NULL || IS_P2ALIGNED(aindx, maxpages));
3860         ASSERT(SEGVN_LOCK_HELD(seg->s_as, &svd->lock));
```



```
3861         ASSERT(seg->s_szc < NBBY * sizeof (int));
3862         ASSERT(type != F_SOFTLOCK || lpgeaddr - a == maxpgsz);
3863         ASSERT(svd->tr_state != SEGVN_TR_INIT);

3865         VM_STAT_COND_ADD(type == F_SOFTLOCK, segvnvmstats.fltvnpages[0]);
3866         VM_STAT_COND_ADD(type != F_SOFTLOCK, segvnvmstats.fltvnpages[1]);

3868         if (svd->flags & MAP_TEXT) {
3869                 hat_flag |= HAT_LOAD_TEXT;
3870         }

3872         if (svd->pageprot) {
3873                 switch (rw) {
3874                 case S_READ:
3875                         protchk = PROT_READ;
3876                         break;
3877                 case S_WRITE:
3878                         protchk = PROT_WRITE;
3879                         break;
3880                 case S_EXEC:
3881                         protchk = PROT_EXEC;
3882                         break;
3883                 case S_OTHER:
3884                 default:
3885                         protchk = PROT_READ | PROT_WRITE | PROT_EXEC;
3886                         break;
3887                 }
3888         } else {
3889                 prot = svd->prot;
3890                 /* caller has already done segment level protection check. */
3891         }

3909         if (seg->s_as->a_hat != hat) {
3910                 xhat = 1;
3911         }

3893         if (rw == S_WRITE && segtype == MAP_PRIVATE) {
3894                 SEGVN_VMSTAT_FLTVNPAGES(2);
3895                 arw = S_READ;
3896         } else {
3897                 arw = rw;
3898         }

3900         ppa = kmem_alloc(ppasize, KM_SLEEP);

3902         VM_STAT_COND_ADD(amp != NULL, segvnvmstats.fltvnpages[3]);

3904         for (;;) {
3905                 adjszc_chk = 0;
3906                 for (; a < lpgeaddr; a += pgsz, off += pgsz, aindx += pages) {
3907                         if (adjszc_chk) {
3908                                 while (szc < seg->s_szc) {
3909                                         uintptr_t e;
3910                                         uint_t tszc;
3911                                         tszc = segvn_anypgsz_vnode ? szc + 1 :
3912                                             seg->s_szc;
3913                                         ppgsz = page_get_pagesize(tszc);
3914                                         if (!IS_P2ALIGNED(a, ppgsz) ||
3915                                             ((alloc_failed >> tszc) & 0x1)) {
3916                                                 break;
3917                                         }
3918                                         SEGVN_VMSTAT_FLTVNPAGES(4);
3919                                         szc = tszc;
3920                                         pgsz = ppgsz;
3921                                         pages = btop(pgsz);
3922                                         e = P2ROUNDUP((uintptr_t)eaddr, pgsz);
```

```
3923                                    lpgeaddr = (caddr_t)e;
3924                            }
3925                    }

3927            again:
3928                    if (IS_P2ALIGNED(a, maxpgsz) && amp != NULL) {
3929                            ASSERT(IS_P2ALIGNED(aindx, maxpages));
3930                            ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
3931                            anon_array_enter(amp, aindx, &an_cookie);
3932                            if (anon_get_ptr(amp->ahp, aindx) != NULL) {
3933                                    SEGVN_VMSTAT_FLTVNPAGES(5);
3934                                    ASSERT(anon_pages(amp->ahp, aindx,
3935                                        maxpages) == maxpages);
3936                                    anon_array_exit(&an_cookie);
3937                                    ANON_LOCK_EXIT(&amp->a_rwlock);
3938                                    err = segvn_fault_anonpages(hat, seg,
3939                                        a, a + maxpgsz, type, rw,
3940                                        MAX(a, addr),
3941                                        MIN(a + maxpgsz, eaddr), brkcow);
3942                                    if (err != 0) {
3943                                            SEGVN_VMSTAT_FLTVNPAGES(6);
3944                                            goto out;
3945                                    }
3946                                    if (szc < seg->s_szc) {
3947                                            szc = seg->s_szc;
3948                                            pgsz = maxpgsz;
3949                                            pages = maxpages;
3950                                            lpgeaddr = maxlpgeaddr;
3951                                    }
3952                                    goto next;
3953                            } else {
3954                                    ASSERT(anon_pages(amp->ahp, aindx,
3955                                        maxpages) == 0);
3956                                    SEGVN_VMSTAT_FLTVNPAGES(7);
3957                                    anon_array_exit(&an_cookie);
3958                                    ANON_LOCK_EXIT(&amp->a_rwlock);
3959                            }
3960                    }
3961                    ASSERT(!brkcow || IS_P2ALIGNED(a, maxpgsz));
3962                    ASSERT(!tron || IS_P2ALIGNED(a, maxpgsz));

3964                    if (svd->pageprot != 0 && IS_P2ALIGNED(a, maxpgsz)) {
3965                            ASSERT(vpage != NULL);
3966                            prot = VPP_PROT(vpage);
3967                            ASSERT(sameprot(seg, a, maxpgsz));
3968                            if ((prot & protchk) == 0) {
3969                                    SEGVN_VMSTAT_FLTVNPAGES(8);
3970                                    err = FC_PROT;
3971                                    goto out;
3972                            }
3973                    }
3974                    if (type == F_SOFTLOCK) {
3975                            atomic_add_long((ulong_t *)&svd->softlockcnt,
3976                                pages);
3977                    }

3979                    pplist = NULL;
3980                    physcontig = 0;
3981                    ppa[0] = NULL;
3982                    if (!brkcow && !tron && szc &&
3983                        !page_exists_physcontig(vp, off, szc,
3984                        segtype == MAP_PRIVATE ? ppa : NULL)) {
3985                            SEGVN_VMSTAT_FLTVNPAGES(9);
3986                            if (page_alloc_pages(vp, seg, a, &pplist, NULL,
3987                                szc, 0, 0) && type != F_SOFTLOCK) {
3988                                    SEGVN_VMSTAT_FLTVNPAGES(10);
```



```
3989                                    pszc = 0;
3990                                    ierr = -1;
3991                                    alloc_failed |= (1 << szc);
3992                                    break;
3993                            }
3994                            if (pplist != NULL &&
3995                                vp->v_mpssdata == SEGVN_PAGEIO) {
3996                                    int downsize;
3997                                    SEGVN_VMSTAT_FLTVNPAGES(11);
3998                                    physcontig = segvn_fill_vp_pages(svd,
3999                                        vp, off, szc, ppa, &pplist,
4000                                        &pszc, &downsize);
4001                                    ASSERT(!physcontig || pplist == NULL);
4002                                    if (!physcontig && downsize &&
4003                                        type != F_SOFTLOCK) {
4004                                            ASSERT(pplist == NULL);
4005                                            SEGVN_VMSTAT_FLTVNPAGES(12);
4006                                            ierr = -1;
4007                                            break;
4008                                    }
4009                                    ASSERT(!physcontig ||
4010                                        segtype == MAP_PRIVATE ||
4011                                        ppa[0] == NULL);
4012                                    if (physcontig && ppa[0] == NULL) {
4013                                            physcontig = 0;
4014                                    }
4015                            }
4016                    } else if (!brkcow && !tron && szc && ppa[0] != NULL) {
4017                            SEGVN_VMSTAT_FLTVNPAGES(13);
4018                            ASSERT(segtype == MAP_PRIVATE);
4019                            physcontig = 1;
4020                    }

4022                    if (!physcontig) {
4023                            SEGVN_VMSTAT_FLTVNPAGES(14);
4024                            ppa[0] = NULL;
4025                            ierr = VOP_GETPAGE(vp, (offset_t)off, pgsz,
4026                                &vpprot, ppa, pgsz, seg, a, arw,
4027                                svd->cred, NULL);
4028 #ifdef DEBUG
4029                            if (ierr == 0) {
4030                                    for (i = 0; i < pages; i++) {
4031                                            ASSERT(PAGE_LOCKED(ppa[i]));
4032                                            ASSERT(!PP_ISFREE(ppa[i]));
4033                                            ASSERT(ppa[i]->p_vnode == vp);
4034                                            ASSERT(ppa[i]->p_offset ==
4035                                                off + (i << PAGESHIFT));
4036                                    }
4037                            }
4038 #endif /* DEBUG */
4039                            if (segtype == MAP_PRIVATE) {
4040                                    SEGVN_VMSTAT_FLTVNPAGES(15);
4041                                    vpprot &= ~PROT_WRITE;
4042                            }
4043                    } else {
4044                            ASSERT(segtype == MAP_PRIVATE);
4045                            SEGVN_VMSTAT_FLTVNPAGES(16);
4046                            vpprot = PROT_ALL & ~PROT_WRITE;
4047                            ierr = 0;
4048                    }

4050                    if (ierr != 0) {
4051                            SEGVN_VMSTAT_FLTVNPAGES(17);
4052                            if (pplist != NULL) {
4053                                    SEGVN_VMSTAT_FLTVNPAGES(18);
4054                                    page_free_replacement_page(pplist);
```

```
4055                                    page_create_putback(pages);
4056                            }
4057                            SEGVN_RESTORE_SOFTLOCK_VP(type, pages);
4058                            if (a + pgsz <= eaddr) {
4059                                    SEGVN_VMSTAT_FLTVNPAGES(19);
4060                                    err = FC_MAKE_ERR(ierr);
4061                                    goto out;
4062                            }
4063                            va.va_mask = AT_SIZE;
4064                            if (VOP_GETATTR(vp, &va, 0, svd->cred, NULL)) {
4065                                    SEGVN_VMSTAT_FLTVNPAGES(20);
4066                                    err = FC_MAKE_ERR(EIO);
4067                                    goto out;
4068                            }
4069                            if (btopr(va.va_size) >= btopr(off + pgsz)) {
4070                                    SEGVN_VMSTAT_FLTVNPAGES(21);
4071                                    err = FC_MAKE_ERR(ierr);
4072                                    goto out;
4073                            }
4074                            if (btopr(va.va_size) <
4075                                btopr(off + (eaddr - a))) {
4076                                    SEGVN_VMSTAT_FLTVNPAGES(22);
4077                                    err = FC_MAKE_ERR(ierr);
4078                                    goto out;
4079                            }
4080                            if (brkcow || tron || type == F_SOFTLOCK) {
4081                                    /* can't reduce map area */
4082                                    SEGVN_VMSTAT_FLTVNPAGES(23);
4083                                    vop_size_err = 1;
4084                                    goto out;
4085                            }
4086                            SEGVN_VMSTAT_FLTVNPAGES(24);
4087                            ASSERT(szc != 0);
4088                            pszc = 0;
4089                            ierr = -1;
4090                            break;
4091                    }

4093                    if (amp != NULL) {
4094                            ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
4095                            anon_array_enter(amp, aindx, &an_cookie);
4096                    }
4097                    if (amp != NULL &&
4098                        anon_get_ptr(amp->ahp, aindx) != NULL) {
4099                            ulong_t taindx = P2ALIGN(aindx, maxpages);

4101                            SEGVN_VMSTAT_FLTVNPAGES(25);
4102                            ASSERT(anon_pages(amp->ahp, taindx,
4103                                maxpages) == maxpages);
4104                            for (i = 0; i < pages; i++) {
4105                                    page_unlock(ppa[i]);
4106                            }
4107                            anon_array_exit(&an_cookie);
4108                            ANON_LOCK_EXIT(&amp->a_rwlock);
4109                            if (pplist != NULL) {
4110                                    page_free_replacement_page(pplist);
4111                                    page_create_putback(pages);
4112                            }
4113                            SEGVN_RESTORE_SOFTLOCK_VP(type, pages);
4114                            if (szc < seg->s_szc) {
4115                                    SEGVN_VMSTAT_FLTVNPAGES(26);
4116                                    /*
4117                                     * For private segments SOFTLOCK
4118                                     * either always breaks cow (any rw
4119                                     * type except S_READ_NOCOW) or
4120                                     * address space is locked as writer
```



```
4121                                     * (S_READ_NOCOW case) and anon slots
4122                                     * can't show up on second check.
4123                                     * Therefore if we are here for
4124                                     * SOFTLOCK case it must be a cow
4125                                     * break but cow break never reduces
4126                                     * szc. text replication (tron) in
4127                                     * this case works as cow break.
4128                                     * Thus the assert below.
4129                                     */
4130                                    ASSERT(!brkcow && !tron &&
4131                                        type != F_SOFTLOCK);
4132                                    pszc = seg->s_szc;
4133                                    ierr = -2;
4134                                    break;
4135                            }
4136                            ASSERT(IS_P2ALIGNED(a, maxpgsz));
4137                            goto again;
4138                    }
4139 #ifdef DEBUG
4140                    if (amp != NULL) {
4141                            ulong_t taindx = P2ALIGN(aindx, maxpages);
4142                            ASSERT(!anon_pages(amp->ahp, taindx, maxpages));
4143                    }
4144 #endif /* DEBUG */

4146                    if (brkcow || tron) {
4147                            ASSERT(amp != NULL);
4148                            ASSERT(pplist == NULL);
4149                            ASSERT(szc == seg->s_szc);
4150                            ASSERT(IS_P2ALIGNED(a, maxpgsz));
4151                            ASSERT(IS_P2ALIGNED(aindx, maxpages));
4152                            SEGVN_VMSTAT_FLTVNPAGES(27);
4153                            ierr = anon_map_privatepages(amp, aindx, szc,
4154                                seg, a, prot, ppa, vpage, segvn_anypgsz,
4155                                tron ? PG_LOCAL : 0, svd->cred);
4156                            if (ierr != 0) {
4157                                    SEGVN_VMSTAT_FLTVNPAGES(28);
4158                                    anon_array_exit(&an_cookie);
4159                                    ANON_LOCK_EXIT(&amp->a_rwlock);
4160                                    SEGVN_RESTORE_SOFTLOCK_VP(type, pages);
4161                                    err = FC_MAKE_ERR(ierr);
4162                                    goto out;
4163                            }

4165                            ASSERT(!IS_VMODSORT(ppa[0]->p_vnode));
4166                            /*
4167                             * p_szc can't be changed for locked
4168                             * swapfs pages.
4169                             */
4170                            ASSERT(svd->rcookie ==
4171                                HAT_INVALID_REGION_COOKIE);
4172                            hat_memload_array(hat, a, pgsz, ppa, prot,
4173                                hat_flag);

4175                            if (!(hat_flag & HAT_LOAD_LOCK)) {
4176                                    SEGVN_VMSTAT_FLTVNPAGES(29);
4177                                    for (i = 0; i < pages; i++) {
4178                                            page_unlock(ppa[i]);
4179                                    }
4180                            }
4181                            anon_array_exit(&an_cookie);
4182                            ANON_LOCK_EXIT(&amp->a_rwlock);
4183                            goto next;
4184                    }

4186                    ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE ||
```

```
4187                            (!svd->pageprot && svd->prot == (prot & vpprot)));
4188
4189                        pfn = page_pptonum(ppa[0]);
4190                        /*
4191                         * hat_page_demote() needs an SE_EXCL lock on one of
4192                         * constituent page_t's and it decreases root's p_szc
4193                         * last. This means if root's p_szc is equal szc and
4194                         * all its constituent pages are locked
4195                         * hat_page_demote() that could have changed p_szc to
4196                         * szc is already done and no new have page_demote()
4197                         * can start for this large page.
4198                         */
4199
4200                        /*
4201                         * we need to make sure same mapping size is used for
4202                         * the same address range if there's a possibility the
4203                         * adddress is already mapped because hat layer panics
4204                         * when translation is loaded for the range already
4205                         * mapped with a different page size.  We achieve it
4206                         * by always using largest page size possible subject
4207                         * to the constraints of page size, segment page size
4208                         * and page alignment.  Since mappings are invalidated
4209                         * when those constraints change and make it
4210                         * impossible to use previously used mapping size no
4211                         * mapping size conflicts should happen.
4212                         */
4213
4214                chkszc:
4215                        if ((pszc = ppa[0]->p_szc) == szc &&
4216                            IS_P2ALIGNED(pfn, pages)) {
4217
4218                                SEGVN_VMSTAT_FLTVNPAGES(30);
4219 #ifdef DEBUG
4220                                for (i = 0; i < pages; i++) {
4221                                        ASSERT(PAGE_LOCKED(ppa[i]));
4222                                        ASSERT(!PP_ISFREE(ppa[i]));
4223                                        ASSERT(page_pptonum(ppa[i]) ==
4224                                            pfn + i);
4225                                        ASSERT(ppa[i]->p_szc == szc);
4226                                        ASSERT(ppa[i]->p_vnode == vp);
4227                                        ASSERT(ppa[i]->p_offset ==
4228                                            off + (i << PAGESHIFT));
4229                                }
4230 #endif /* DEBUG */
4231                                /*
4232                                 * All pages are of szc we need and they are
4233                                 * all locked so they can't change szc. load
4234                                 * translations.
4235                                 *
4236                                 * if page got promoted since last check
4237                                 * we don't need pplist.
4238                                 */
4239                                if (pplist != NULL) {
4240                                        page_free_replacement_page(pplist);
4241                                        page_create_putback(pages);
4242                                }
4243                                if (PP_ISMIGRATE(ppa[0])) {
4244                                        page_migrate(seg, a, ppa, pages);
4245                                }
4246                                SEGVN_UPDATE_MODBITS(ppa, pages, rw,
4247                                    prot, vpprot);
4248                                if (!xhat) {
4249                                hat_memload_array_region(hat, a, pgsz,
4250                                    ppa, prot & vpprot, hat_flag,
4272                                    svd->rcookie);
                                } else {
```



```
4273                                        /*
4274                                         * avoid large xhat mappings to FS
4275                                         * pages so that hat_page_demote()
4276                                         * doesn't need to check for xhat
4277                                         * large mappings.
4278                                         * Don't use regions with xhats.
4279                                         */
4280                                        for (i = 0; i < pages; i++) {
4281                                                hat_memload(hat,
4282                                                    a + (i << PAGESHIFT),
4283                                                    ppa[i], prot & vpprot,
4284                                                    hat_flag);
4285                                        }
4286                                }
4287
4288                                if (!(hat_flag & HAT_LOAD_LOCK)) {
4289                                        for (i = 0; i < pages; i++) {
4290                                                page_unlock(ppa[i]);
4291                                        }
4292                                }
4293                                if (amp != NULL) {
4294                                        anon_array_exit(&an_cookie);
4295                                        ANON_LOCK_EXIT(&amp->a_rwlock);
4296                                }
4297                                goto next;
4298                        }
4299
4300                        /*
4301                         * See if upsize is possible.
4302                         */
4303                        if (pszc > szc && szc < seg->s_szc &&
4304                            (segvn_anypgsz_vnode || pszc >= seg->s_szc)) {
4305                                pgcnt_t aphase;
4306                                uint_t pszc1 = MIN(pszc, seg->s_szc);
4307                                ppgsz = page_get_pagesize(pszc1);
4308                                ppages = btop(ppgsz);
4309                                aphase = btop(P2PHASE((uintptr_t)a, ppgsz));
4310
4311                                ASSERT(type != F_SOFTLOCK);
4312
4313                                SEGVN_VMSTAT_FLTVNPAGES(31);
4314                                if (aphase != P2PHASE(pfn, ppages)) {
4315                                        segvn_faultvnmpss_align_err4++;
4316                                } else {
4317                                        SEGVN_VMSTAT_FLTVNPAGES(32);
4318                                        if (pplist != NULL) {
4319                                                page_t *pl = pplist;
4320                                                page_free_replacement_page(pl);
4321                                                page_create_putback(pages);
4322                                        }
4323                                        for (i = 0; i < pages; i++) {
4324                                                page_unlock(ppa[i]);
4325                                        }
4326                                        if (amp != NULL) {
4327                                                anon_array_exit(&an_cookie);
4328                                                ANON_LOCK_EXIT(&amp->a_rwlock);
4329                                        }
4330                                        pszc = pszc1;
4331                                        ierr = -2;
4332                                        break;
4333                                }
4334                        }
4335
4336                        /*
4337                         * check if we should use smallest mapping size.
4338                         */
```

```
4303                        upgrdfail = 0;
4304                        if (szc == 0 ||
4340                        if (szc == 0 || xhat ||
4305                            (pszc >= szc &&
4306                            !IS_P2ALIGNED(pfn, pages)) ||
4307                            (pszc < szc &&
4308                            !segvn_full_szcpages(ppa, szc, &upgrdfail,
4309                            &pszc))) {

4311                                if (upgrdfail && type != F_SOFTLOCK) {
4312                                        /*
4313                                         * segvn_full_szcpages failed to lock
4314                                         * all pages EXCL. Size down.
4315                                         */
4316                                        ASSERT(pszc < szc);

4318                                        SEGVN_VMSTAT_FLTVNPAGES(33);

4320                                        if (pplist != NULL) {
4321                                                page_t *pl = pplist;
4322                                                page_free_replacement_page(pl);
4323                                                page_create_putback(pages);
4324                                        }

4326                                        for (i = 0; i < pages; i++) {
4327                                                page_unlock(ppa[i]);
4328                                        }
4329                                        if (amp != NULL) {
4330                                                anon_array_exit(&an_cookie);
4331                                                ANON_LOCK_EXIT(&amp->a_rwlock);
4332                                        }
4333                                        ierr = -1;
4334                                        break;
4335                                }
4336                                if (szc != 0 && !upgrdfail) {
4372                                if (szc != 0 && !xhat && !upgrdfail) {
4337                                        segvn_faultvnmpss_align_err5++;
4338                                }
4339                                SEGVN_VMSTAT_FLTVNPAGES(34);
4340                                if (pplist != NULL) {
4341                                        page_free_replacement_page(pplist);
4342                                        page_create_putback(pages);
4343                                }
4344                                SEGVN_UPDATE_MODBITS(ppa, pages, rw,
4345                                    prot, vpprot);
4346                                if (upgrdfail && segvn_anypgsz_vnode) {
4347                                        /* SOFTLOCK case */
4348                                        hat_memload_array_region(hat, a, pgsz,
4349                                            ppa, prot & vpprot, hat_flag,
4350                                            svd->rcookie);
4351                                } else {
4352                                        for (i = 0; i < pages; i++) {
4353                                                hat_memload_region(hat,
4354                                                    a + (i << PAGESHIFT),
4355                                                    ppa[i], prot & vpprot,
4356                                                    hat_flag, svd->rcookie);
4357                                        }
4358                                }
4359                                if (!(hat_flag & HAT_LOAD_LOCK)) {
4360                                        for (i = 0; i < pages; i++) {
4361                                                page_unlock(ppa[i]);
4362                                        }
4363                                }
4364                                if (amp != NULL) {
4365                                        anon_array_exit(&an_cookie);
4366                                        ANON_LOCK_EXIT(&amp->a_rwlock);
```



```
4367                                }
4368                                goto next;
4369                        }

4371                        if (pszc == szc) {
4372                                /*
4373                                 * segvn_full_szcpages() upgraded pages szc.
4374                                 */
4375                                ASSERT(pszc == ppa[0]->p_szc);
4376                                ASSERT(IS_P2ALIGNED(pfn, pages));
4377                                goto chkszc;
4378                        }

4380                        if (pszc > szc) {
4381                                kmutex_t *szcmtx;
4382                                SEGVN_VMSTAT_FLTVNPAGES(35);
4383                                /*
4384                                 * p_szc of ppa[0] can change since we haven't
4385                                 * locked all constituent pages. Call
4386                                 * page_lock_szc() to prevent szc changes.
4387                                 * This should be a rare case that happens when
4388                                 * multiple segments use a different page size
4389                                 * to map the same file offsets.
4390                                 */
4391                                szcmtx = page_szc_lock(ppa[0]);
4392                                pszc = ppa[0]->p_szc;
4393                                ASSERT(szcmtx != NULL || pszc == 0);
4394                                ASSERT(ppa[0]->p_szc <= pszc);
4395                                if (pszc <= szc) {
4396                                        SEGVN_VMSTAT_FLTVNPAGES(36);
4397                                        if (szcmtx != NULL) {
4398                                                mutex_exit(szcmtx);
4399                                        }
4400                                        goto chkszc;
4401                                }
4402                                if (pplist != NULL) {
4403                                        /*
4404                                         * page got promoted since last check.
4405                                         * we don't need preaalocated large
4406                                         * page.
4407                                         */
4408                                        SEGVN_VMSTAT_FLTVNPAGES(37);
4409                                        page_free_replacement_page(pplist);
4410                                        page_create_putback(pages);
4411                                }
4412                                SEGVN_UPDATE_MODBITS(ppa, pages, rw,
4413                                    prot, vpprot);
4414                                hat_memload_array_region(hat, a, pgsz, ppa,
4415                                    prot & vpprot, hat_flag, svd->rcookie);
4416                                mutex_exit(szcmtx);
4417                                if (!(hat_flag & HAT_LOAD_LOCK)) {
4418                                        for (i = 0; i < pages; i++) {
4419                                                page_unlock(ppa[i]);
4420                                        }
4421                                }
4422                                if (amp != NULL) {
4423                                        anon_array_exit(&an_cookie);
4424                                        ANON_LOCK_EXIT(&amp->a_rwlock);
4425                                }
4426                                goto next;
4427                        }

4429                        /*
4430                         * if page got demoted since last check
4431                         * we could have not allocated larger page.
4432                         * allocate now.
```

```
4433                         */
4434                        if (pplist == NULL &&
4435                            page_alloc_pages(vp, seg, a, &pplist, NULL,
4436                            szc, 0, 0) && type != F_SOFTLOCK) {
4437                                SEGVN_VMSTAT_FLTVNPAGES(38);
4438                                for (i = 0; i < pages; i++) {
4439                                        page_unlock(ppa[i]);
4440                                }
4441                                if (amp != NULL) {
4442                                        anon_array_exit(&an_cookie);
4443                                        ANON_LOCK_EXIT(&amp->a_rwlock);
4444                                }
4445                                ierr = -1;
4446                                alloc_failed |= (1 << szc);
4447                                break;
4448                        }

4450                        SEGVN_VMSTAT_FLTVNPAGES(39);

4452                        if (pplist != NULL) {
4453                                segvn_relocate_pages(ppa, pplist);
4454 #ifdef DEBUG
4455                        } else {
4456                                ASSERT(type == F_SOFTLOCK);
4457                                SEGVN_VMSTAT_FLTVNPAGES(40);
4458 #endif /* DEBUG */
4459                        }

4461                        SEGVN_UPDATE_MODBITS(ppa, pages, rw, prot, vpprot);

4463                        if (pplist == NULL && segvn_anypgsz_vnode == 0) {
4464                                ASSERT(type == F_SOFTLOCK);
4465                                for (i = 0; i < pages; i++) {
4466                                        ASSERT(ppa[i]->p_szc < szc);
4467                                        hat_memload_region(hat,
4468                                            a + (i << PAGESHIFT),
4469                                            ppa[i], prot & vpprot, hat_flag,
4470                                            svd->rcookie);
4471                                }
4472                        } else {
4473                                ASSERT(pplist != NULL || type == F_SOFTLOCK);
4474                                hat_memload_array_region(hat, a, pgsz, ppa,
4475                                    prot & vpprot, hat_flag, svd->rcookie);
4476                        }
4477                        if (!(hat_flag & HAT_LOAD_LOCK)) {
4478                                for (i = 0; i < pages; i++) {
4479                                        ASSERT(PAGE_SHARED(ppa[i]));
4480                                        page_unlock(ppa[i]);
4481                                }
4482                        }
4483                        if (amp != NULL) {
4484                                anon_array_exit(&an_cookie);
4485                                ANON_LOCK_EXIT(&amp->a_rwlock);
4486                        }

4488                next:
4489                        if (vpage != NULL) {
4490                                vpage += pages;
4491                        }
4492                        adjszc_chk = 1;
4493                }
4494                if (a == lpgeaddr)
4495                        break;
4496                ASSERT(a < lpgeaddr);

4498                ASSERT(!brkcow && !tron && type != F_SOFTLOCK);
```



```
4500                /*
4501                 * ierr == -1 means we failed to map with a large page.
4502                 * (either due to allocation/relocation failures or
4503                 * misalignment with other mappings to this file.
4504                 *
4505                 * ierr == -2 means some other thread allocated a large page
4506                 * after we gave up tp map with a large page.  retry with
4507                 * larger mapping.
4508                 */
4509                ASSERT(ierr == -1 || ierr == -2);
4510                ASSERT(ierr == -2 || szc != 0);
4511                ASSERT(ierr == -1 || szc < seg->s_szc);
4512                if (ierr == -2) {
4513                        SEGVN_VMSTAT_FLTVNPAGES(41);
4514                        ASSERT(pszc > szc && pszc <= seg->s_szc);
4515                        szc = pszc;
4516                } else if (segvn_anypgsz_vnode) {
4517                        SEGVN_VMSTAT_FLTVNPAGES(42);
4518                        szc--;
4519                } else {
4520                        SEGVN_VMSTAT_FLTVNPAGES(43);
4521                        ASSERT(pszc < szc);
4522                        /*
4523                         * other process created pszc large page.
4524                         * but we still have to drop to 0 szc.
4525                         */
4526                        szc = 0;
4527                }

4529                pgsz = page_get_pagesize(szc);
4530                pages = btop(pgsz);
4531                if (ierr == -2) {
4532                        /*
4533                         * Size up case. Note lpgaddr may only be needed for
4534                         * softlock case so we don't adjust it here.
4535                         */
4536                        a = (caddr_t)P2ALIGN((uintptr_t)a, pgsz);
4537                        ASSERT(a >= lpgaddr);
4538                        lpgeaddr = (caddr_t)P2ROUNDUP((uintptr_t)eaddr, pgsz);
4539                        off = svd->offset + (uintptr_t)(a - seg->s_base);
4540                        aindx = svd->anon_index + seg_page(seg, a);
4541                        vpage = (svd->vpage != NULL) ?
4542                            &svd->vpage[seg_page(seg, a)] : NULL;
4543                } else {
4544                        /*
4545                         * Size down case. Note lpgaddr may only be needed for
4546                         * softlock case so we don't adjust it here.
4547                         */
4548                        ASSERT(IS_P2ALIGNED(a, pgsz));
4549                        ASSERT(IS_P2ALIGNED(lpgeaddr, pgsz));
4550                        lpgeaddr = (caddr_t)P2ROUNDUP((uintptr_t)eaddr, pgsz);
4551                        ASSERT(a < lpgeaddr);
4552                        if (a < addr) {
4553                                SEGVN_VMSTAT_FLTVNPAGES(44);
4554                                /*
4555                                 * The beginning of the large page region can
4556                                 * be pulled to the right to make a smaller
4557                                 * region. We haven't yet faulted a single
4558                                 * page.
4559                                 */
4560                                a = (caddr_t)P2ALIGN((uintptr_t)addr, pgsz);
4561                                ASSERT(a >= lpgaddr);
4562                                off = svd->offset +
4563                                    (uintptr_t)(a - seg->s_base);
4564                                aindx = svd->anon_index + seg_page(seg, a);
```

```
4565                                vpage = (svd->vpage != NULL) ?
4566                                    &svd->vpage[seg_page(seg, a)] : NULL;
4567                        }
4568                }
4569        }
4570 out:
4571        kmem_free(ppa, ppasize);
4572        if (!err && !vop_size_err) {
4573                SEGVN_VMSTAT_FLTVNPAGES(45);
4574                return (0);
4575        }
4576        if (type == F_SOFTLOCK && a > lpgaddr) {
4577                SEGVN_VMSTAT_FLTVNPAGES(46);
4578                segvn_softunlock(seg, lpgaddr, a - lpgaddr, S_OTHER);
4579        }
4580        if (!vop_size_err) {
4581                SEGVN_VMSTAT_FLTVNPAGES(47);
4582                return (err);
4583        }
4584        ASSERT(brkcow || tron || type == F_SOFTLOCK);
4585        /*
4586         * Large page end is mapped beyond the end of file and it's a cow
4587         * fault (can be a text replication induced cow) or softlock so we can't
4588         * reduce the map area.  For now just demote the segment. This should
4589         * really only happen if the end of the file changed after the mapping
4590         * was established since when large page segments are created we make
4591         * sure they don't extend beyond the end of the file.
4592         */
4593        SEGVN_VMSTAT_FLTVNPAGES(48);

4595        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4596        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
4597        err = 0;
4598        if (seg->s_szc != 0) {
4599                segvn_fltvnpages_clrszc_cnt++;
4600                ASSERT(svd->softlockcnt == 0);
4601                err = segvn_clrszc(seg);
4602                if (err != 0) {
4603                        segvn_fltvnpages_clrszc_err++;
4604                }
4605        }
4606        ASSERT(err || seg->s_szc == 0);
4607        SEGVN_LOCK_DOWNGRADE(seg->s_as, &svd->lock);
4608        /* segvn_fault will do its job as if szc had been zero to begin with */
4609        return (err == 0 ? IE_RETRY : FC_MAKE_ERR(err));
4610 }
```

*unchanged_portion_omitted_*

```
4883 int fltadvice = 1;      /* set to free behind pages for sequential access */

4885 /*
4886  * This routine is called via a machine specific fault handling routine.
4887  * It is also called by software routines wishing to lock or unlock
4888  * a range of addresses.
4889  *
4890  * Here is the basic algorithm:
4891  *      If unlocking
4892  *              Call segvn_softunlock
4893  *              Return
4894  *      endif
4895  *      Checking and set up work
4896  *      If we will need some non-anonymous pages
4897  *              Call VOP_GETPAGE over the range of non-anonymous pages
4898  *      endif
4899  *      Loop over all addresses requested
4900  *              Call segvn_faultpage passing in page list
```



```
4901  *                  to load up translations and handle anonymous pages
4902  *      endloop
4903  *      Load up translation to any additional pages in page list not
4904  *          already handled that fit into this segment
4905  */
4906 static faultcode_t
4907 segvn_fault(struct hat *hat, struct seg *seg, caddr_t addr, size_t len,
4908     enum fault_type type, enum seg_rw rw)
4909 {
4910         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
4911         page_t **plp, **ppp, *pp;
4912         u_offset_t off;
4913         caddr_t a;
4914         struct vpage *vpage;
4915         uint_t vpprot, prot;
4916         int err;
4917         page_t *pl[FAULT_TMP_PAGES_NUM + 1];
4953         page_t *pl[PVN_GETPAGE_NUM + 1];
4918         size_t plsz, pl_alloc_sz;
4919         size_t page;
4920         ulong_t anon_index;
4921         struct anon_map *amp;
4922         int dogetpage = 0;
4923         caddr_t lpgaddr, lpgeaddr;
4924         size_t pgsz;
4925         anon_sync_obj_t cookie;
4926         int brkcow = BREAK_COW_SHARE(rw, type, svd->type);

4928         ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
4929         ASSERT(svd->amp == NULL || svd->rcookie == HAT_INVALID_REGION_COOKIE);

4931         /*
4932          * First handle the easy stuff
4933          */
4934         if (type == F_SOFTUNLOCK) {
4935                 if (rw == S_READ_NOCOW) {
4936                         rw = S_READ;
4937                         ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
4938                 }
4939                 SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
4940                 pgsz = (seg->s_szc == 0) ? PAGESIZE :
4941                     page_get_pagesize(seg->s_szc);
4942                 VM_STAT_COND_ADD(pgsz > PAGESIZE, segvnvmstats.fltanpages[16]);
4943                 CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
4944                 segvn_softunlock(seg, lpgaddr, lpgeaddr - lpgaddr, rw);
4945                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4946                 return (0);
4947         }

4949         ASSERT(svd->tr_state == SEGVN_TR_OFF ||
4950             !HAT_IS_REGION_COOKIE_VALID(svd->rcookie));
4951         if (brkcow == 0) {
4952                 if (svd->tr_state == SEGVN_TR_INIT) {
4953                         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
4954                         if (svd->tr_state == SEGVN_TR_INIT) {
4955                                 ASSERT(svd->vp != NULL && svd->amp == NULL);
4956                                 ASSERT(svd->flags & MAP_TEXT);
4957                                 ASSERT(svd->type == MAP_PRIVATE);
4958                                 segvn_textrepl(seg);
4959                                 ASSERT(svd->tr_state != SEGVN_TR_INIT);
4960                                 ASSERT(svd->tr_state != SEGVN_TR_ON ||
4961                                     svd->amp != NULL);
4962                         }
4963                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4964                 }
4965         } else if (svd->tr_state != SEGVN_TR_OFF) {
```

```
4966                 SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);

4968                 if (rw == S_WRITE && svd->tr_state != SEGVN_TR_OFF) {
4969                         ASSERT(!svd->pageprot && !(svd->prot & PROT_WRITE));
4970                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4971                         return (FC_PROT);
4972                 }

4974                 if (svd->tr_state == SEGVN_TR_ON) {
4975                         ASSERT(svd->vp != NULL && svd->amp != NULL);
4976                         segvn_textunrepl(seg, 0);
4977                         ASSERT(svd->amp == NULL &&
4978                             svd->tr_state == SEGVN_TR_OFF);
4979                 } else if (svd->tr_state != SEGVN_TR_OFF) {
4980                         svd->tr_state = SEGVN_TR_OFF;
4981                 }
4982                 ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
4983                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4984         }

4986 top:
4987         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);

4989         /*
4990          * If we have the same protections for the entire segment,
4991          * insure that the access being attempted is legitimate.
4992          */

4994         if (svd->pageprot == 0) {
4995                 uint_t protchk;

4997                 switch (rw) {
4998                 case S_READ:
4999                 case S_READ_NOCOW:
5000                         protchk = PROT_READ;
5001                         break;
5002                 case S_WRITE:
5003                         protchk = PROT_WRITE;
5004                         break;
5005                 case S_EXEC:
5006                         protchk = PROT_EXEC;
5007                         break;
5008                 case S_OTHER:
5009                 default:
5010                         protchk = PROT_READ | PROT_WRITE | PROT_EXEC;
5011                         break;
5012                 }

5014                 if ((svd->prot & protchk) == 0) {
5015                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5016                         return (FC_PROT);       /* illegal access type */
5017                 }
5018         }

5020         if (brkcow && HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
5021                 /* this must be SOFTLOCK S_READ fault */
5022                 ASSERT(svd->amp == NULL);
5023                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
5024                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5025                 SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
5026                 if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
5027                         /*
5028                          * this must be the first ever non S_READ_NOCOW
5029                          * softlock for this segment.
5030                          */
5031                         ASSERT(svd->softlockcnt == 0);
```



```
5032                         hat_leave_region(seg->s_as->a_hat, svd->rcookie,
5033                             HAT_REGION_TEXT);
5034                         svd->rcookie = HAT_INVALID_REGION_COOKIE;
5035                 }
5036                 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5037                 goto top;
5038         }

5040         /*
5041          * We can't allow the long term use of softlocks for vmpss segments,
5042          * because in some file truncation cases we should be able to demote
5043          * the segment, which requires that there are no softlocks.  The
5044          * only case where it's ok to allow a SOFTLOCK fault against a vmpss
5045          * segment is S_READ_NOCOW, where the caller holds the address space
5046          * locked as writer and calls softunlock before dropping the as lock.
5047          * S_READ_NOCOW is used by /proc to read memory from another user.
5048          *
5049          * Another deadlock between SOFTLOCK and file truncation can happen
5050          * because segvn_fault_vnodepages() calls the FS one pagesize at
5051          * a time. A second VOP_GETPAGE() call by segvn_fault_vnodepages()
5052          * can cause a deadlock because the first set of page_t's remain
5053          * locked SE_SHARED.  To avoid this, we demote segments on a first
5054          * SOFTLOCK if they have a length greater than the segment's
5055          * page size.
5056          *
5057          * So for now, we only avoid demoting a segment on a SOFTLOCK when
5058          * the access type is S_READ_NOCOW and the fault length is less than
5059          * or equal to the segment's page size. While this is quite restrictive,
5060          * it should be the most common case of SOFTLOCK against a vmpss
5061          * segment.
5062          *
5063          * For S_READ_NOCOW, it's safe not to do a copy on write because the
5064          * caller makes sure no COW will be caused by another thread for a
5065          * softlocked page.
5066          */
5067         if (type == F_SOFTLOCK && svd->vp != NULL && seg->s_szc != 0) {
5068                 int demote = 0;

5070                 if (rw != S_READ_NOCOW) {
5071                         demote = 1;
5072                 }
5073                 if (!demote && len > PAGESIZE) {
5074                         pgsz = page_get_pagesize(seg->s_szc);
5075                         CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr,
5076                             lpgeaddr);
5077                         if (lpgeaddr - lpgaddr > pgsz) {
5078                                 demote = 1;
5079                         }
5080                 }

5082                 ASSERT(demote || AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

5084                 if (demote) {
5085                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5086                         SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
5087                         if (seg->s_szc != 0) {
5088                                 segvn_vmpss_clrszc_cnt++;
5089                                 ASSERT(svd->softlockcnt == 0);
5090                                 err = segvn_clrszc(seg);
5091                                 if (err) {
5092                                         segvn_vmpss_clrszc_err++;
5093                                         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5094                                         return (FC_MAKE_ERR(err));
5095                                 }
5096                         }
5097                         ASSERT(seg->s_szc == 0);
```

```
5098                            SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5099                            goto top;
5100                    }
5101            }

5103            /*
5104             * Check to see if we need to allocate an anon_map structure.
5105             */
5106            if (svd->amp == NULL && (svd->vp == NULL || brkcow)) {
5107                    ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
5108                    /*
5109                     * Drop the "read" lock on the segment and acquire
5110                     * the "write" version since we have to allocate the
5111                     * anon_map.
5112                     */
5113                    SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5114                    SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);

5116                    if (svd->amp == NULL) {
5117                            svd->amp = anonmap_alloc(seg->s_size, 0, ANON_SLEEP);
5118                            svd->amp->a_szc = seg->s_szc;
5119                    }
5120                    SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);

5122                    /*
5123                     * Start all over again since segment protections
5124                     * may have changed after we dropped the "read" lock.
5125                     */
5126                    goto top;
5127            }

5129            /*
5130             * S_READ_NOCOW vs S_READ distinction was
5131             * only needed for the code above. After
5132             * that we treat it as S_READ.
5133             */
5134            if (rw == S_READ_NOCOW) {
5135                    ASSERT(type == F_SOFTLOCK);
5136                    ASSERT(AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
5137                    rw = S_READ;
5138            }

5140            amp = svd->amp;

5142            /*
5143             * MADV_SEQUENTIAL work is ignored for large page segments.
5144             */
5145            if (seg->s_szc != 0) {
5146                    pgsz = page_get_pagesize(seg->s_szc);
5147                    ASSERT(SEGVN_LOCK_HELD(seg->s_as, &svd->lock));
5148                    CALC_LPG_REGION(pgsz, seg, addr, len, lpgaddr, lpgeaddr);
5149                    if (svd->vp == NULL) {
5150                            err = segvn_fault_anonpages(hat, seg, lpgaddr,
5151                                lpgeaddr, type, rw, addr, addr + len, brkcow);
5152                    } else {
5153                            err = segvn_fault_vnodepages(hat, seg, lpgaddr,
5154                                lpgeaddr, type, rw, addr, addr + len, brkcow);
5155                            if (err == IE_RETRY) {
5156                                    ASSERT(seg->s_szc == 0);
5157                                    ASSERT(SEGVN_READ_HELD(seg->s_as, &svd->lock));
5158                                    SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5159                                    goto top;
5160                            }
5161                    }
5162                    SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5163                    return (err);
```



```
5164            }

5166            page = seg_page(seg, addr);
5167            if (amp != NULL) {
5168                    ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE);
5169                    anon_index = svd->anon_index + page;

5171                    if (type == F_PROT && rw == S_READ &&
5172                        svd->tr_state == SEGVN_TR_OFF &&
5173                        svd->type == MAP_PRIVATE && svd->pageprot == 0) {
5174                            size_t index = anon_index;
5175                            struct anon *ap;

5177                            ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
5178                            /*
5179                             * The fast path could apply to S_WRITE also, except
5180                             * that the protection fault could be caused by lazy
5181                             * tlb flush when ro->rw. In this case, the pte is
5182                             * RW already. But RO in the other cpu's tlb causes
5183                             * the fault. Since hat_chgprot won't do anything if
5184                             * pte doesn't change, we may end up faulting
5185                             * indefinitely until the RO tlb entry gets replaced.
5186                             */
5187                            for (a = addr; a < addr + len; a += PAGESIZE, index++) {
5188                                    anon_array_enter(amp, index, &cookie);
5189                                    ap = anon_get_ptr(amp->ahp, index);
5190                                    anon_array_exit(&cookie);
5191                                    if ((ap == NULL) || (ap->an_refcnt != 1)) {
5192                                            ANON_LOCK_EXIT(&amp->a_rwlock);
5193                                            goto slow;
5194                                    }
5195                            }
5196                            hat_chgprot(seg->s_as->a_hat, addr, len, svd->prot);
5197                            ANON_LOCK_EXIT(&amp->a_rwlock);
5198                            SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5199                            return (0);
5200                    }
5201            }
5202    slow:

5204            if (svd->vpage == NULL)
5205                    vpage = NULL;
5206            else
5207                    vpage = &svd->vpage[page];

5209            off = svd->offset + (uintptr_t)(addr - seg->s_base);

5211            /*
5212             * If MADV_SEQUENTIAL has been set for the particular page we
5213             * are faulting on, free behind all pages in the segment and put
5214             * them on the free list.
5215             */

5217            if ((page != 0) && fltadvice && svd->tr_state != SEGVN_TR_ON) {
5218                    struct vpage *vpp;
5219                    ulong_t fanon_index;
5220                    size_t fpage;
5221                    u_offset_t pgoff, fpgoff;
5222                    struct vnode *fvp;
5223                    struct anon *fap = NULL;

5225                    if (svd->advice == MADV_SEQUENTIAL ||
5226                        (svd->pageadvice &&
5227                        VPP_ADVICE(vpage) == MADV_SEQUENTIAL)) {
5228                            pgoff = off - PAGESIZE;
5229                            fpage = page - 1;
```

```
5230                        if (vpage != NULL)
5231                                vpp = &svd->vpage[fpage];
5232                        if (amp != NULL)
5233                                fanon_index = svd->anon_index + fpage;
5234
5235                        while (pgoff > svd->offset) {
5236                                if (svd->advice != MADV_SEQUENTIAL &&
5237                                    (!svd->pageadvice || (vpage &&
5238                                    VPP_ADVICE(vpp) != MADV_SEQUENTIAL)))
5239                                        break;
5240
5241                                /*
5242                                 * If this is an anon page, we must find the
5243                                 * correct <vp, offset> for it
5244                                 */
5245                                fap = NULL;
5246                                if (amp != NULL) {
5247                                        ANON_LOCK_ENTER(&amp->a_rwlock,
5248                                            RW_READER);
5249                                        anon_array_enter(amp, fanon_index,
5250                                            &cookie);
5251                                        fap = anon_get_ptr(amp->ahp,
5252                                            fanon_index);
5253                                        if (fap != NULL) {
5254                                                swap_xlate(fap, &fvp, &fpgoff);
5255                                        } else {
5256                                                fpgoff = pgoff;
5257                                                fvp = svd->vp;
5258                                        }
5259                                        anon_array_exit(&cookie);
5260                                        ANON_LOCK_EXIT(&amp->a_rwlock);
5261                                } else {
5262                                        fpgoff = pgoff;
5263                                        fvp = svd->vp;
5264                                }
5265                                if (fvp == NULL)
5266                                        break;  /* XXX */
5267                                /*
5268                                 * Skip pages that are free or have an
5269                                 * "exclusive" lock.
5270                                 */
5271                                pp = page_lookup_nowait(fvp, fpgoff, SE_SHARED);
5272                                if (pp == NULL)
5273                                        break;
5274                                /*
5275                                 * We don't need the page_struct_lock to test
5276                                 * as this is only advisory; even if we
5277                                 * acquire it someone might race in and lock
5278                                 * the page after we unlock and before the
5279                                 * PUTPAGE, then VOP_PUTPAGE will do nothing.
5280                                 */
5281                                if (pp->p_lckcnt == 0 && pp->p_cowcnt == 0) {
5282                                        /*
5283                                         * Hold the vnode before releasing
5284                                         * the page lock to prevent it from
5285                                         * being freed and re-used by some
5286                                         * other thread.
5287                                         */
5288                                        VN_HOLD(fvp);
5289                                        page_unlock(pp);
5290                                        /*
5291                                         * We should build a page list
5292                                         * to kluster putpages XXX
5293                                         */
5294                                        (void) VOP_PUTPAGE(fvp,
5295                                            (offset_t)fpgoff, PAGESIZE,
```



```
5296                                            (B_DONTNEED|B_FREE|B_ASYNC),
5297                                            svd->cred, NULL);
5298                                        VN_RELE(fvp);
5299                                } else {
5300                                        /*
5301                                         * XXX - Should the loop terminate if
5302                                         * the page is 'locked'?
5303                                         */
5304                                        page_unlock(pp);
5305                                }
5306                                --vpp;
5307                                --fanon_index;
5308                                pgoff -= PAGESIZE;
5309                        }
5310                }
5311        }
5312
5313        plp = pl;
5314        *plp = NULL;
5315        pl_alloc_sz = 0;
5316
5317        /*
5318         * See if we need to call VOP_GETPAGE for
5319         * *any* of the range being faulted on.
5320         * We can skip all of this work if there
5321         * was no original vnode.
5322         */
5323        if (svd->vp != NULL) {
5324                u_offset_t vp_off;
5325                size_t vp_len;
5326                struct anon *ap;
5327                vnode_t *vp;
5328
5329                vp_off = off;
5330                vp_len = len;
5331
5332                if (amp == NULL)
5333                        dogetpage = 1;
5334                else {
5335                        /*
5336                         * Only acquire reader lock to prevent amp->ahp
5337                         * from being changed.  It's ok to miss pages,
5338                         * hence we don't do anon_array_enter
5339                         */
5340                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
5341                        ap = anon_get_ptr(amp->ahp, anon_index);
5342
5343                        if (len <= PAGESIZE)
5344                                /* inline non_anon() */
5345                                dogetpage = (ap == NULL);
5346                        else
5347                                dogetpage = non_anon(amp->ahp, anon_index,
5348                                    &vp_off, &vp_len);
5349                        ANON_LOCK_EXIT(&amp->a_rwlock);
5350                }
5351
5352                if (dogetpage) {
5353                        enum seg_rw arw;
5354                        struct as *as = seg->s_as;
5355
5356                        if (len > FAULT_TMP_PAGES_SZ) {
5392                        if (len > ptob((sizeof (pl) / sizeof (pl[0])) - 1)) {
5357                                /*
5358                                 * Page list won't fit in local array,
5359                                 * allocate one of the needed size.
5360                                 */
```

```
5361                                    pl_alloc_sz =
5362                                        (btop(len) + 1) * sizeof (page_t *);
5363                                    plp = kmem_alloc(pl_alloc_sz, KM_SLEEP);
5364                                    plp[0] = NULL;
5365                                    plsz = len;
5366                            } else if (rw == S_WRITE && svd->type == MAP_PRIVATE ||
5367                                svd->tr_state == SEGVN_TR_ON || rw == S_OTHER ||
5368                                (((size_t)(addr + PAGESIZE) <
5369                                (size_t)(seg->s_base + seg->s_size)) &&
5370                                hat_probe(as->a_hat, addr + PAGESIZE))) {
5371                                    /*
5372                                     * Ask VOP_GETPAGE to return the exact number
5373                                     * of pages if
5374                                     * (a) this is a COW fault, or
5375                                     * (b) this is a software fault, or
5376                                     * (c) next page is already mapped.
5377                                     */
5378                                    plsz = len;
5379                            } else {
5380                                    /*
5381                                     * Ask VOP_GETPAGE to return adjacent pages
5382                                     * within the segment.
5383                                     */
5384                                    plsz = MIN((size_t)FAULT_TMP_PAGES_SZ, (size_t)
5420                                    plsz = MIN((size_t)PVN_GETPAGE_SZ, (size_t)
5385                                        ((seg->s_base + seg->s_size) - addr));
5386                                    ASSERT((addr + plsz) <=
5387                                        (seg->s_base + seg->s_size));
5388                            }

5390                            /*
5391                             * Need to get some non-anonymous pages.
5392                             * We need to make only one call to GETPAGE to do
5393                             * this to prevent certain deadlocking conditions
5394                             * when we are doing locking.  In this case
5395                             * non_anon() should have picked up the smallest
5396                             * range which includes all the non-anonymous
5397                             * pages in the requested range.  We have to
5398                             * be careful regarding which rw flag to pass in
5399                             * because on a private mapping, the underlying
5400                             * object is never allowed to be written.
5401                             */
5402                            if (rw == S_WRITE && svd->type == MAP_PRIVATE) {
5403                                    arw = S_READ;
5404                            } else {
5405                                    arw = rw;
5406                            }
5407                            vp = svd->vp;
5408                            TRACE_3(TR_FAC_VM, TR_SEGVN_GETPAGE,
5409                                "segvn_getpage:seg %p addr %p vp %p",
5410                                seg, addr, vp);
5411                            err = VOP_GETPAGE(vp, (offset_t)vp_off, vp_len,
5412                                &vpprot, plp, plsz, seg, addr + (vp_off - off), arw,
5413                                svd->cred, NULL);
5414                            if (err) {
5415                                    SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5416                                    segvn_pagelist_rele(plp);
5417                                    if (pl_alloc_sz)
5418                                            kmem_free(plp, pl_alloc_sz);
5419                                    return (FC_MAKE_ERR(err));
5420                            }
5421                            if (svd->type == MAP_PRIVATE)
5422                                    vpprot &= ~PROT_WRITE;
5423                    }
5424            }
```



```
5426            /*
5427             * N.B. at this time the plp array has all the needed non-anon
5428             * pages in addition to (possibly) having some adjacent pages.
5429             */

5431            /*
5432             * Always acquire the anon_array_lock to prevent
5433             * 2 threads from allocating separate anon slots for
5434             * the same "addr".
5435             *
5436             * If this is a copy-on-write fault and we don't already
5437             * have the anon_array_lock, acquire it to prevent the
5438             * fault routine from handling multiple copy-on-write faults
5439             * on the same "addr" in the same address space.
5440             *
5441             * Only one thread should deal with the fault since after
5442             * it is handled, the other threads can acquire a translation
5443             * to the newly created private page.  This prevents two or
5444             * more threads from creating different private pages for the
5445             * same fault.
5446             *
5447             * We grab "serialization" lock here if this is a MAP_PRIVATE segment
5448             * to prevent deadlock between this thread and another thread
5449             * which has soft-locked this page and wants to acquire serial_lock.
5450             * ( bug 4026339 )
5451             *
5452             * The fix for bug 4026339 becomes unnecessary when using the
5453             * locking scheme with per amp rwlock and a global set of hash
5454             * lock, anon_array_lock.  If we steal a vnode page when low
5455             * on memory and upgrad the page lock through page_rename,
5456             * then the page is PAGE_HANDLED, nothing needs to be done
5457             * for this page after returning from segvn_faultpage.
5458             *
5459             * But really, the page lock should be downgraded after
5460             * the stolen page is page_rename'd.
5461             */

5463            if (amp != NULL)
5464                    ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);

5466            /*
5467             * Ok, now loop over the address range and handle faults
5468             */
5469            for (a = addr; a < addr + len; a += PAGESIZE, off += PAGESIZE) {
5470                    err = segvn_faultpage(hat, seg, a, off, vpage, plp, vpprot,
5471                        type, rw, brkcow);
5472                    if (err) {
5473                            if (amp != NULL)
5474                                    ANON_LOCK_EXIT(&amp->a_rwlock);
5475                            if (type == F_SOFTLOCK && a > addr) {
5476                                    segvn_softunlock(seg, addr, (a - addr),
5477                                        S_OTHER);
5478                            }
5479                            SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5480                            segvn_pagelist_rele(plp);
5481                            if (pl_alloc_sz)
5482                                    kmem_free(plp, pl_alloc_sz);
5483                            return (err);
5484                    }
5485                    if (vpage) {
5486                            vpage++;
5487                    } else if (svd->vpage) {
5488                            page = seg_page(seg, addr);
5489                            vpage = &svd->vpage[++page];
5490                    }
5491            }
```

```
5493         /* Didn't get pages from the underlying fs so we're done */
5494         if (!dogetpage)
5495                 goto done;

5497         /*
5498          * Now handle any other pages in the list returned.
5499          * If the page can be used, load up the translations now.
5500          * Note that the for loop will only be entered if "plp"
5501          * is pointing to a non-NULL page pointer which means that
5502          * VOP_GETPAGE() was called and vpprot has been initialized.
5503          */
5504         if (svd->pageprot == 0)
5505                 prot = svd->prot & vpprot;


5508         /*
5509          * Large Files: diff should be unsigned value because we started
5510          * supporting > 2GB segment sizes from 2.5.1 and when a
5511          * large file of size > 2GB gets mapped to address space
5512          * the diff value can be > 2GB.
5513          */

5515         for (ppp = plp; (pp = *ppp) != NULL; ppp++) {
5516                 size_t diff;
5517                 struct anon *ap;
5518                 int anon_index;
5519                 anon_sync_obj_t cookie;
5520                 int hat_flag = HAT_LOAD_ADV;

5522                 if (svd->flags & MAP_TEXT) {
5523                         hat_flag |= HAT_LOAD_TEXT;
5524                 }

5526                 if (pp == PAGE_HANDLED)
5527                         continue;

5529                 if (svd->tr_state != SEGVN_TR_ON &&
5530                     pp->p_offset >=  svd->offset &&
5531                     pp->p_offset < svd->offset + seg->s_size) {

5533                         diff = pp->p_offset - svd->offset;

5535                         /*
5536                          * Large Files: Following is the assertion
5537                          * validating the above cast.
5538                          */
5539                         ASSERT(svd->vp == pp->p_vnode);

5541                         page = btop(diff);
5542                         if (svd->pageprot)
5543                                 prot = VPP_PROT(&svd->vpage[page]) & vpprot;

5545                         /*
5546                          * Prevent other threads in the address space from
5547                          * creating private pages (i.e., allocating anon slots)
5548                          * while we are in the process of loading translations
5549                          * to additional pages returned by the underlying
5550                          * object.
5551                          */
5552                         if (amp != NULL) {
5553                                 anon_index = svd->anon_index + page;
5554                                 anon_array_enter(amp, anon_index, &cookie);
5555                                 ap = anon_get_ptr(amp->ahp, anon_index);
5556                         }
5557                         if ((amp == NULL) || (ap == NULL)) {
```



```
5558                                 if (IS_VMODSORT(pp->p_vnode) ||
5559                                     enable_mbit_wa) {
5560                                         if (rw == S_WRITE)
5561                                                 hat_setmod(pp);
5562                                         else if (rw != S_OTHER &&
5563                                             !hat_ismod(pp))
5564                                                 prot &= ~PROT_WRITE;
5565                                 }
5566                                 /*
5567                                  * Skip mapping read ahead pages marked
5568                                  * for migration, so they will get migrated
5569                                  * properly on fault
5570                                  */
5571                                 ASSERT(amp == NULL ||
5572                                     svd->rcookie == HAT_INVALID_REGION_COOKIE);
5573                                 if ((prot & PROT_READ) && !PP_ISMIGRATE(pp)) {
5574                                         hat_memload_region(hat,
5575                                             seg->s_base + diff,
5576                                             pp, prot, hat_flag,
5577                                             svd->rcookie);
5578                                 }
5579                         }
5580                         if (amp != NULL)
5581                                 anon_array_exit(&cookie);
5582                 }
5583                 page_unlock(pp);
5584         }
5585 done:
5586         if (amp != NULL)
5587                 ANON_LOCK_EXIT(&amp->a_rwlock);
5588         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
5589         if (pl_alloc_sz)
5590                 kmem_free(plp, pl_alloc_sz);
5591         return (0);
5592 }
```

*unchanged_portion_omitted_*

```
6050 /*
6051  * segvn_setpagesize is called via segop_setpagesize from as_setpagesize,
6087  * segvn_setpagesize is called via SEGOP_SETPAGESIZE from as_setpagesize,
6052  * to determine if the seg is capable of mapping the requested szc.
6053  */
6054 static int
6055 segvn_setpagesize(struct seg *seg, caddr_t addr, size_t len, uint_t szc)
6056 {
6057         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6058         struct segvn_data *nsvd;
6059         struct anon_map *amp = svd->amp;
6060         struct seg *nseg;
6061         caddr_t eaddr = addr + len, a;
6062         size_t pgsz = page_get_pagesize(szc);
6063         pgcnt_t pgcnt = page_get_pagecnt(szc);
6064         int err;
6065         u_offset_t off = svd->offset + (uintptr_t)(addr - seg->s_base);

6067         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
6068         ASSERT(addr >= seg->s_base && eaddr <= seg->s_base + seg->s_size);

6070         if (seg->s_szc == szc || segvn_lpg_disable != 0) {
6071                 return (0);
6072         }

6074         /*
6075          * addr should always be pgsz aligned but eaddr may be misaligned if
6076          * it's at the end of the segment.
6077          *
```

```
6078          * XXX we should assert this condition since as_setpagesize() logic
6079          * guarantees it.
6080          */
6081         if (!IS_P2ALIGNED(addr, pgsz) ||
6082             (!IS_P2ALIGNED(eaddr, pgsz) &&
6083             eaddr != seg->s_base + seg->s_size)) {

6085                 segvn_setpgsz_align_err++;
6086                 return (EINVAL);
6087         }

6089         if (amp != NULL && svd->type == MAP_SHARED) {
6090                 ulong_t an_idx = svd->anon_index + seg_page(seg, addr);
6091                 if (!IS_P2ALIGNED(an_idx, pgcnt)) {

6093                         segvn_setpgsz_anon_align_err++;
6094                         return (EINVAL);
6095                 }
6096         }

6098         if ((svd->flags & MAP_NORESERVE) || seg->s_as == &kas ||
6099             szc > segvn_maxpgszc) {
6100                 return (EINVAL);
6101         }

6103         /* paranoid check */
6104         if (svd->vp != NULL &&
6105             (IS_SWAPFSVP(svd->vp) || VN_ISKAS(svd->vp))) {
6106                 return (EINVAL);
6107         }

6109         if (seg->s_szc == 0 && svd->vp != NULL &&
6110             map_addr_vacalign_check(addr, off)) {
6111                 return (EINVAL);
6112         }

6114         /*
6115          * Check that protections are the same within new page
6116          * size boundaries.
6117          */
6118         if (svd->pageprot) {
6119                 for (a = addr; a < eaddr; a += pgsz) {
6120                         if ((a + pgsz) > eaddr) {
6121                                 if (!sameprot(seg, a, eaddr - a)) {
6122                                         return (EINVAL);
6123                                 }
6124                         } else {
6125                                 if (!sameprot(seg, a, pgsz)) {
6126                                         return (EINVAL);
6127                                 }
6128                         }
6129                 }
6130         }

6132         /*
6133          * Since we are changing page size we first have to flush
6134          * the cache. This makes sure all the pagelock calls have
6135          * to recheck protections.
6136          */
6137         if (svd->softlockcnt > 0) {
6138                 ASSERT(svd->tr_state == SEGVN_TR_OFF);

6140                 /*
6141                  * If this is shared segment non 0 softlockcnt
6142                  * means locked pages are still in use.
6143                  */
```



```
6144                 if (svd->type == MAP_SHARED) {
6145                         return (EAGAIN);
6146                 }

6148                 /*
6149                  * Since we do have the segvn writers lock nobody can fill
6150                  * the cache with entries belonging to this seg during
6151                  * the purge. The flush either succeeds or we still have
6152                  * pending I/Os.
6153                  */
6154                 segvn_purge(seg);
6155                 if (svd->softlockcnt > 0) {
6156                         return (EAGAIN);
6157                 }
6158         }

6160         if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
6161                 ASSERT(svd->amp == NULL);
6162                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
6163                 hat_leave_region(seg->s_as->a_hat, svd->rcookie,
6164                     HAT_REGION_TEXT);
6165                 svd->rcookie = HAT_INVALID_REGION_COOKIE;
6166         } else if (svd->tr_state == SEGVN_TR_INIT) {
6167                 svd->tr_state = SEGVN_TR_OFF;
6168         } else if (svd->tr_state == SEGVN_TR_ON) {
6169                 ASSERT(svd->amp != NULL);
6170                 segvn_textunrepl(seg, 1);
6171                 ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
6172                 amp = NULL;
6173         }

6175         /*
6176          * Operation for sub range of existing segment.
6177          */
6178         if (addr != seg->s_base || eaddr != (seg->s_base + seg->s_size)) {
6179                 if (szc < seg->s_szc) {
6180                         VM_STAT_ADD(segvnvmstats.demoterange[2]);
6181                         err = segvn_demote_range(seg, addr, len, SDR_RANGE, 0);
6182                         if (err == 0) {
6183                                 return (IE_RETRY);
6184                         }
6185                         if (err == ENOMEM) {
6186                                 return (IE_NOMEM);
6187                         }
6188                         return (err);
6189                 }
6190                 if (addr != seg->s_base) {
6191                         nseg = segvn_split_seg(seg, addr);
6192                         if (eaddr != (nseg->s_base + nseg->s_size)) {
6193                                 /* eaddr is szc aligned */
6194                                 (void) segvn_split_seg(nseg, eaddr);
6195                         }
6196                         return (IE_RETRY);
6197                 }
6198                 if (eaddr != (seg->s_base + seg->s_size)) {
6199                         /* eaddr is szc aligned */
6200                         (void) segvn_split_seg(seg, eaddr);
6201                 }
6202                 return (IE_RETRY);
6203         }

6205         /*
6206          * Break any low level sharing and reset seg->s_szc to 0.
6207          */
6208         if ((err = segvn_clrszc(seg)) != 0) {
6209                 if (err == ENOMEM) {
```

```
6210                        err = IE_NOMEM;
6211                }
6212                return (err);
6213        }
6214        ASSERT(seg->s_szc == 0);

6216        /*
6217         * If the end of the current segment is not pgsz aligned
6218         * then attempt to concatenate with the next segment.
6219         */
6220        if (!IS_P2ALIGNED(eaddr, pgsz)) {
6221                nseg = AS_SEGNEXT(seg->s_as, seg);
6222                if (nseg == NULL || nseg == seg || eaddr != nseg->s_base) {
6223                        return (ENOMEM);
6224                }
6225                if (nseg->s_ops != &segvn_ops) {
6226                        return (EINVAL);
6227                }
6228                nsvd = (struct segvn_data *)nseg->s_data;
6229                if (nsvd->softlockcnt > 0) {
6230                        /*
6231                         * If this is shared segment non 0 softlockcnt
6232                         * means locked pages are still in use.
6233                         */
6234                        if (nsvd->type == MAP_SHARED) {
6235                                return (EAGAIN);
6236                        }
6237                        segvn_purge(nseg);
6238                        if (nsvd->softlockcnt > 0) {
6239                                return (EAGAIN);
6240                        }
6241                }
6242                err = segvn_clrszc(nseg);
6243                if (err == ENOMEM) {
6244                        err = IE_NOMEM;
6245                }
6246                if (err != 0) {
6247                        return (err);
6248                }
6249                ASSERT(nsvd->rcookie == HAT_INVALID_REGION_COOKIE);
6250                err = segvn_concat(seg, nseg, 1);
6251                if (err == -1) {
6252                        return (EINVAL);
6253                }
6254                if (err == -2) {
6255                        return (IE_NOMEM);
6256                }
6257                return (IE_RETRY);
6258        }

6260        /*
6261         * May need to re-align anon array to
6262         * new szc.
6263         */
6264        if (amp != NULL) {
6265                if (!IS_P2ALIGNED(svd->anon_index, pgcnt)) {
6266                        struct anon_hdr *nahp;

6268                        ASSERT(svd->type == MAP_PRIVATE);

6270                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
6271                        ASSERT(amp->refcnt == 1);
6272                        nahp = anon_create(btop(amp->size), ANON_NOSLEEP);
6273                        if (nahp == NULL) {
6274                                ANON_LOCK_EXIT(&amp->a_rwlock);
6275                                return (IE_NOMEM);
```



```
6276                        }
6277                        if (anon_copy_ptr(amp->ahp, svd->anon_index,
6278                            nahp, 0, btop(seg->s_size), ANON_NOSLEEP)) {
6279                                anon_release(nahp, btop(amp->size));
6280                                ANON_LOCK_EXIT(&amp->a_rwlock);
6281                                return (IE_NOMEM);
6282                        }
6283                        anon_release(amp->ahp, btop(amp->size));
6284                        amp->ahp = nahp;
6285                        svd->anon_index = 0;
6286                        ANON_LOCK_EXIT(&amp->a_rwlock);
6287                }
6288        }
6289        if (svd->vp != NULL && szc != 0) {
6290                struct vattr va;
6291                u_offset_t eoffpage = svd->offset;
6292                va.va_mask = AT_SIZE;
6293                eoffpage += seg->s_size;
6294                eoffpage = btopr(eoffpage);
6295                if (VOP_GETATTR(svd->vp, &va, 0, svd->cred, NULL) != 0) {
6296                        segvn_setpgsz_getattr_err++;
6297                        return (EINVAL);
6298                }
6299                if (btopr(va.va_size) < eoffpage) {
6300                        segvn_setpgsz_eof_err++;
6301                        return (EINVAL);
6302                }
6303                if (amp != NULL) {
6304                        /*
6305                         * anon_fill_cow_holes() may call VOP_GETPAGE().
6306                         * don't take anon map lock here to avoid holding it
6307                         * across VOP_GETPAGE() calls that may call back into
6308                         * segvn for klsutering checks. We don't really need
6309                         * anon map lock here since it's a private segment and
6310                         * we hold as level lock as writers.
6311                         */
6312                        if ((err = anon_fill_cow_holes(seg, seg->s_base,
6313                            amp->ahp, svd->anon_index, svd->vp, svd->offset,
6314                            seg->s_size, szc, svd->prot, svd->vpage,
6315                            svd->cred)) != 0) {
6316                                return (EINVAL);
6317                        }
6318                }
6319                segvn_setvnode_mpss(svd->vp);
6320        }

6322        if (amp != NULL) {
6323                ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
6324                if (svd->type == MAP_PRIVATE) {
6325                        amp->a_szc = szc;
6326                } else if (szc > amp->a_szc) {
6327                        amp->a_szc = szc;
6328                }
6329                ANON_LOCK_EXIT(&amp->a_rwlock);
6330        }

6332        seg->s_szc = szc;

6334        return (0);
6335 }
```

_____unchanged_portion_omitted_

```
7038 /*
7075  * Swap the pages of seg out to secondary storage, returning the
7076  * number of bytes of storage freed.
7077  *
```

```
 * The basic idea is first to unload all translations and then to call
 * VOP_PUTPAGE() for all newly-unmapped pages, to push them out to the
 * swap device.  Pages to which other segments have mappings will remain
 * mapped and won't be swapped.  Our caller (as_swapout) has already
 * performed the unloading step.
 *
 * The value returned is intended to correlate well with the process's
 * memory requirements.  However, there are some caveats:
 * 1)   When given a shared segment as argument, this routine will
 *      only succeed in swapping out pages for the last sharer of the
 *      segment.  (Previous callers will only have decremented mapping
 *      reference counts.)
 * 2)   We assume that the hat layer maintains a large enough translation
 *      cache to capture process reference patterns.
 */
static size_t
segvn_swapout(struct seg *seg)
{
        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
        struct anon_map *amp;
        pgcnt_t pgcnt = 0;
        pgcnt_t npages;
        pgcnt_t page;
        ulong_t anon_index;

        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
        /*
         * Find pages unmapped by our caller and force them
         * out to the virtual swap device.
         */
        if ((amp = svd->amp) != NULL)
                anon_index = svd->anon_index;
        npages = seg->s_size >> PAGESHIFT;
        for (page = 0; page < npages; page++) {
                page_t *pp;
                struct anon *ap;
                struct vnode *vp;
                u_offset_t off;
                anon_sync_obj_t cookie;

                /*
                 * Obtain <vp, off> pair for the page, then look it up.
                 *
                 * Note that this code is willing to consider regular
                 * pages as well as anon pages.  Is this appropriate here?
                 */
                ap = NULL;
                if (amp != NULL) {
                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
                        if (anon_array_try_enter(amp, anon_index + page,
                            &cookie)) {
                                ANON_LOCK_EXIT(&amp->a_rwlock);
                                continue;
                        }
                        ap = anon_get_ptr(amp->ahp, anon_index + page);
                        if (ap != NULL) {
                                swap_xlate(ap, &vp, &off);
                        } else {
                                vp = svd->vp;
                                off = svd->offset + ptob(page);
                        }
                        anon_array_exit(&cookie);
                        ANON_LOCK_EXIT(&amp->a_rwlock);
                } else {
```



```
                        vp = svd->vp;
                        off = svd->offset + ptob(page);
                }
                if (vp == NULL) {               /* untouched zfod page */
                        ASSERT(ap == NULL);
                        continue;
                }

                pp = page_lookup_nowait(vp, off, SE_SHARED);
                if (pp == NULL)
                        continue;


                /*
                 * Examine the page to see whether it can be tossed out,
                 * keeping track of how many we've found.
                 */
                if (!page_tryupgrade(pp)) {
                        /*
                         * If the page has an i/o lock and no mappings,
                         * it's very likely that the page is being
                         * written out as a result of klustering.
                         * Assume this is so and take credit for it here.
                         */
                        if (!page_io_trylock(pp)) {
                                if (!hat_page_is_mapped(pp))
                                        pgcnt++;
                        } else {
                                page_io_unlock(pp);
                        }
                        page_unlock(pp);
                        continue;
                }
                ASSERT(!page_iolock_assert(pp));


                /*
                 * Skip if page is locked or has mappings.
                 * We don't need the page_struct_lock to look at lckcnt
                 * and cowcnt because the page is exclusive locked.
                 */
                if (pp->p_lckcnt != 0 || pp->p_cowcnt != 0 ||
                    hat_page_is_mapped(pp)) {
                        page_unlock(pp);
                        continue;
                }

                /*
                 * dispose skips large pages so try to demote first.
                 */
                if (pp->p_szc != 0 && !page_try_demote_pages(pp)) {
                        page_unlock(pp);
                        /*
                         * XXX should skip the remaining page_t's of this
                         * large page.
                         */
                        continue;
                }

                ASSERT(pp->p_szc == 0);

                /*
                 * No longer mapped -- we can toss it out.  How
                 * we do so depends on whether or not it's dirty.
                 */
                if (hat_ismod(pp) && pp->p_vnode) {
```

```
7210                        /*
7211                         * We must clean the page before it can be
7212                         * freed.  Setting B_FREE will cause pvn_done
7213                         * to free the page when the i/o completes.
7214                         * XXX: This also causes it to be accounted
7215                         *      as a pageout instead of a swap: need
7216                         *      B_SWAPOUT bit to use instead of B_FREE.
7217                         *
7218                         * Hold the vnode before releasing the page lock
7219                         * to prevent it from being freed and re-used by
7220                         * some other thread.
7221                         */
7222                        VN_HOLD(vp);
7223                        page_unlock(pp);

7225                        /*
7226                         * Queue all i/o requests for the pageout thread
7227                         * to avoid saturating the pageout devices.
7228                         */
7229                        if (!queue_io_request(vp, off))
7230                                VN_RELE(vp);
7231                } else {
7232                        /*
7233                         * The page was clean, free it.
7234                         *
7235                         * XXX: Can we ever encounter modified pages
7236                         *      with no associated vnode here?
7237                         */
7238                        ASSERT(pp->p_vnode != NULL);
7239                        /*LINTED: constant in conditional context*/
7240                        VN_DISPOSE(pp, B_FREE, 0, kcred);
7241                }

7243                /*
7244                 * Credit now even if i/o is in progress.
7245                 */
7246                pgcnt++;
7247        }
7248        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);

7250        /*
7251         * Wakeup pageout to initiate i/o on all queued requests.
7252         */
7253        cv_signal_pageout();
7254        return (ptob(pgcnt));
7255 }

7257 /*
7039  * Synchronize primary storage cache with real object in virtual memory.
7040  *
7041  * XXX - Anonymous pages should not be sync'ed out at all.
7042  */
7043 static int
7044 segvn_sync(struct seg *seg, caddr_t addr, size_t len, int attr, uint_t flags)
7045 {
7046        struct segvn_data *svd = (struct segvn_data *)seg->s_data;
7047        struct vpage *vpp;
7048        page_t *pp;
7049        u_offset_t offset;
7050        struct vnode *vp;
7051        u_offset_t off;
7052        caddr_t eaddr;
7053        int bflags;
7054        int err = 0;
7055        int segtype;
7056        int pageprot;
```



```
7057        int prot;
7058        ulong_t anon_index;
7059        struct anon_map *amp;
7060        struct anon *ap;
7061        anon_sync_obj_t cookie;

7063        ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

7065        SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);

7067        if (svd->softlockcnt > 0) {
7068                /*
7069                 * If this is shared segment non 0 softlockcnt
7070                 * means locked pages are still in use.
7071                 */
7072                if (svd->type == MAP_SHARED) {
7073                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7074                        return (EAGAIN);
7075                }

7077                /*
7078                 * flush all pages from seg cache
7079                 * otherwise we may deadlock in swap_putpage
7080                 * for B_INVAL page (4175402).
7081                 *
7082                 * Even if we grab segvn WRITER's lock
7083                 * here, there might be another thread which could've
7084                 * successfully performed lookup/insert just before
7085                 * we acquired the lock here.  So, grabbing either
7086                 * lock here is of not much use.  Until we devise
7087                 * a strategy at upper layers to solve the
7088                 * synchronization issues completely, we expect
7089                 * applications to handle this appropriately.
7090                 */
7091                segvn_purge(seg);
7092                if (svd->softlockcnt > 0) {
7093                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7094                        return (EAGAIN);
7095                }
7096        } else if (svd->type == MAP_SHARED && svd->amp != NULL &&
7097            svd->amp->a_softlockcnt > 0) {
7098                /*
7099                 * Try to purge this amp's entries from pcache. It will
7100                 * succeed only if other segments that share the amp have no
7101                 * outstanding softlock's.
7102                 */
7103                segvn_purge(seg);
7104                if (svd->amp->a_softlockcnt > 0 || svd->softlockcnt > 0) {
7105                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7106                        return (EAGAIN);
7107                }
7108        }

7110        vpp = svd->vpage;
7111        offset = svd->offset + (uintptr_t)(addr - seg->s_base);
7112        bflags = ((flags & MS_ASYNC) ? B_ASYNC : 0) |
7113            ((flags & MS_INVALIDATE) ? B_INVAL : 0);

7115        if (attr) {
7116                pageprot = attr & ~(SHARED|PRIVATE);
7117                segtype = (attr & SHARED) ? MAP_SHARED : MAP_PRIVATE;

7119                /*
7120                 * We are done if the segment types don't match
7121                 * or if we have segment level protections and
7122                 * they don't match.
```

```
7123                 */
7124                if (svd->type != segtype) {
7125                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7126                        return (0);
7127                }
7128                if (vpp == NULL) {
7129                        if (svd->prot != pageprot) {
7130                                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7131                                return (0);
7132                        }
7133                        prot = svd->prot;
7134                } else
7135                        vpp = &svd->vpage[seg_page(seg, addr)];

7137        } else if (svd->vp && svd->amp == NULL &&
7138            (flags & MS_INVALIDATE) == 0) {

7140                /*
7141                 * No attributes, no anonymous pages and MS_INVALIDATE flag
7142                 * is not on, just use one big request.
7143                 */
7144                err = VOP_PUTPAGE(svd->vp, (offset_t)offset, len,
7145                    bflags, svd->cred, NULL);
7146                SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7147                return (err);
7148        }

7150        if ((amp = svd->amp) != NULL)
7151                anon_index = svd->anon_index + seg_page(seg, addr);

7153        for (eaddr = addr + len; addr < eaddr; addr += PAGESIZE) {
7154                ap = NULL;
7155                if (amp != NULL) {
7156                        ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
7157                        anon_array_enter(amp, anon_index, &cookie);
7158                        ap = anon_get_ptr(amp->ahp, anon_index++);
7159                        if (ap != NULL) {
7160                                swap_xlate(ap, &vp, &off);
7161                        } else {
7162                                vp = svd->vp;
7163                                off = offset;
7164                        }
7165                        anon_array_exit(&cookie);
7166                        ANON_LOCK_EXIT(&amp->a_rwlock);
7167                } else {
7168                        vp = svd->vp;
7169                        off = offset;
7170                }
7171                offset += PAGESIZE;

7173                if (vp == NULL)         /* untouched zfod page */
7174                        continue;

7176                if (attr) {
7177                        if (vpp) {
7178                                prot = VPP_PROT(vpp);
7179                                vpp++;
7180                        }
7181                        if (prot != pageprot) {
7182                                continue;
7183                        }
7184                }

7186                /*
7187                 * See if any of these pages are locked --  if so, then we
7188                 * will have to truncate an invalidate request at the first
```



```
7189                 * locked one. We don't need the page_struct_lock to test
7190                 * as this is only advisory; even if we acquire it someone
7191                 * might race in and lock the page after we unlock and before
7192                 * we do the PUTPAGE, then PUTPAGE simply does nothing.
7193                 */
7194                if (flags & MS_INVALIDATE) {
7195                        if ((pp = page_lookup(vp, off, SE_SHARED)) != NULL) {
7196                                if (pp->p_lckcnt != 0 || pp->p_cowcnt != 0) {
7197                                        page_unlock(pp);
7198                                        SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7199                                        return (EBUSY);
7200                                }
7201                                if (ap != NULL && pp->p_szc != 0 &&
7202                                    page_tryupgrade(pp)) {
7203                                        if (pp->p_lckcnt == 0 &&
7204                                            pp->p_cowcnt == 0) {
7205                                                /*
7206                                                 * swapfs VN_DISPOSE() won't
7207                                                 * invalidate large pages.
7208                                                 * Attempt to demote.
7209                                                 * XXX can't help it if it
7210                                                 * fails. But for swapfs
7211                                                 * pages it is no big deal.
7212                                                 */
7213                                                (void) page_try_demote_pages(
7214                                                    pp);
7215                                        }
7216                                }
7217                                page_unlock(pp);
7218                        }
7219                } else if (svd->type == MAP_SHARED && amp != NULL) {
7220                        /*
7221                         * Avoid writing out to disk ISM's large pages
7222                         * because segspt_free_pages() relies on NULL an_pvp
7223                         * of anon slots of such pages.
7224                         */

7226                        ASSERT(svd->vp == NULL);
7227                        /*
7228                         * swapfs uses page_lookup_nowait if not freeing or
7229                         * invalidating and skips a page if
7230                         * page_lookup_nowait returns NULL.
7231                         */
7232                        pp = page_lookup_nowait(vp, off, SE_SHARED);
7233                        if (pp == NULL) {
7234                                continue;
7235                        }
7236                        if (pp->p_szc != 0) {
7237                                page_unlock(pp);
7238                                continue;
7239                        }

7241                        /*
7242                         * Note ISM pages are created large so (vp, off)'s
7243                         * page cannot suddenly become large after we unlock
7244                         * pp.
7245                         */
7246                        page_unlock(pp);
7247                }
7248                /*
7249                 * XXX - Should ultimately try to kluster
7250                 * calls to VOP_PUTPAGE() for performance.
7251                 */
7252                VN_HOLD(vp);
7253                err = VOP_PUTPAGE(vp, (offset_t)off, PAGESIZE,
7254                    (bflags | (IS_SWAPFSVP(vp) ? B_PAGE_NOWAIT : 0)),
```

```
7255                     svd->cred, NULL);

7257                 VN_RELE(vp);
7258                 if (err)
7259                         break;
7260         }
7261         SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
7262         return (err);
7263 }
______unchanged_portion_omitted_

9434 /*
9435  * Get memory allocation policy info for specified address in given segment
9436  */
9437 static lgrp_mem_policy_info_t *
9438 segvn_getpolicy(struct seg *seg, caddr_t addr)
9439 {
9440         struct anon_map         *amp;
9441         ulong_t                 anon_index;
9442         lgrp_mem_policy_info_t  *policy_info;
9443         struct segvn_data       *svn_data;
9444         u_offset_t              vn_off;
9445         vnode_t                 *vp;

9447         ASSERT(seg != NULL);

9449         svn_data = (struct segvn_data *)seg->s_data;
9450         if (svn_data == NULL)
9451                 return (NULL);

9453         /*
9454          * Get policy info for private or shared memory
9455          */
9456         if (svn_data->type != MAP_SHARED) {
9457                 if (svn_data->tr_state != SEGVN_TR_ON) {
9458                         policy_info = &svn_data->policy_info;
9459                 } else {
9460                         policy_info = &svn_data->tr_policy_info;
9461                         ASSERT(policy_info->mem_policy ==
9462                             LGRP_MEM_POLICY_NEXT_SEG);
9463                 }
9464         } else {
9465                 amp = svn_data->amp;
9466                 anon_index = svn_data->anon_index + seg_page(seg, addr);
9467                 vp = svn_data->vp;
9468                 vn_off = svn_data->offset + (uintptr_t)(addr - seg->s_base);
9469                 policy_info = lgrp_shm_policy_get(amp, anon_index, vp, vn_off);
9470         }

9472         return (policy_info);
9692 }

9694 /*ARGSUSED*/
9695 static int
9696 segvn_capable(struct seg *seg, segcapability_t capability)
9697 {
9698         return (0);
9473 }
______unchanged_portion_omitted_
```

```
**********************************************************
    9263 Fri May  8 18:10:37 2015
new/usr/src/uts/common/vm/seg_vn.h
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
**********************************************************
______unchanged_portion_omitted_

 230 extern void     segvn_init(void);
 231 extern int      segvn_create(struct seg *, void *);

 233 extern const struct seg_ops segvn_ops;
 233 extern  struct seg_ops segvn_ops;

 235 /*
 236  * Provided as shorthand for creating user zfod segments.
 237  */
 238 extern  caddr_t zfod_argsp;
 239 extern  caddr_t kzfod_argsp;
 240 extern  caddr_t stack_exec_argsp;
 241 extern  caddr_t stack_noexec_argsp;

 243 #endif  /* _KERNEL */

 245 #ifdef  __cplusplus
 246 }
______unchanged_portion_omitted_
```

```
**********************************************************
   89741 Fri May  8 18:10:37 2015
new/usr/src/uts/common/vm/vm_anon.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
```

```
/*
 * 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) 1986, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */

/*
 * VM - anonymous pages.
 *
 * This layer sits immediately above the vm_swap layer.  It manages
 * physical pages that have no permanent identity in the file system
 * name space, using the services of the vm_swap layer to allocate
 * backing storage for these pages.  Since these pages have no external
 * identity, they are discarded when the last reference is removed.
 *
 * An important function of this layer is to manage low-level sharing
 * of pages that are logically distinct but that happen to be
 * physically identical (e.g., the corresponding pages of the processes
 * resulting from a fork before one process or the other changes their
 * contents).  This pseudo-sharing is present only as an optimization
 * and is not to be confused with true sharing in which multiple
 * address spaces deliberately contain references to the same object;
 * such sharing is managed at a higher level.
 *
```



```
 * The key data structure here is the anon struct, which contains a
 * reference count for its associated physical page and a hint about
 * the identity of that page.  Anon structs typically live in arrays,
 * with an instance's position in its array determining where the
 * corresponding backing storage is allocated; however, the swap_xlate()
 * routine abstracts away this representation information so that the
 * rest of the anon layer need not know it.  (See the swap layer for
 * more details on anon struct layout.)
 *
 * In the future versions of the system, the association between an
 * anon struct and its position on backing store will change so that
 * we don't require backing store all anonymous pages in the system.
 * This is important for consideration for large memory systems.
 * We can also use this technique to delay binding physical locations
 * to anonymous pages until pageout time where we can make smarter
 * allocation decisions to improve anonymous klustering.
 * to anonymous pages until pageout/swapout time where we can make
 * smarter allocation decisions to improve anonymous klustering.
 *
 * Many of the routines defined here take a (struct anon **) argument,
 * which allows the code at this level to manage anon pages directly,
 * so that callers can regard anon structs as opaque objects and not be
 * concerned with assigning or inspecting their contents.
 *
 * Clients of this layer refer to anon pages indirectly.  That is, they
 * maintain arrays of pointers to anon structs rather than maintaining
 * anon structs themselves.  The (struct anon **) arguments mentioned
 * above are pointers to entries in these arrays.  It is these arrays
 * that capture the mapping between offsets within a given segment and
 * the corresponding anonymous backing storage address.
 */

#ifdef DEBUG
#define ANON_DEBUG
#endif

#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/mman.h>
#include <sys/cred.h>
#include <sys/thread.h>
#include <sys/vnode.h>
#include <sys/cpuvar.h>
#include <sys/swap.h>
#include <sys/cmn_err.h>
#include <sys/vtrace.h>
#include <sys/kmem.h>
#include <sys/sysmacros.h>
#include <sys/bitmap.h>
#include <sys/vmsystm.h>
#include <sys/tuneable.h>
#include <sys/debug.h>
#include <sys/fs/swapnode.h>
#include <sys/tnf_probe.h>
#include <sys/lgrp.h>
#include <sys/policy.h>
#include <sys/condvar_impl.h>
#include <sys/mutex_impl.h>
#include <sys/rctl.h>

#include <vm/as.h>
#include <vm/hat.h>
#include <vm/anon.h>
#include <vm/page.h>
```

```
 120 #include <vm/vpage.h>
 121 #include <vm/seg.h>
 122 #include <vm/rm.h>

 124 #include <fs/fs_subr.h>

 126 struct vnode *anon_vp;

 128 int anon_debug;

 130 kmutex_t        anoninfo_lock;
 131 struct          k_anoninfo k_anoninfo;
 132 ani_free_t      *ani_free_pool;
 133 pad_mutex_t     anon_array_lock[ANON_LOCKSIZE];
 134 kcondvar_t      anon_array_cv[ANON_LOCKSIZE];

 136 /*
 137  * Global hash table for (vp, off) -> anon slot
 138  */
 139 extern  int swap_maxcontig;
 140 size_t  anon_hash_size;
 141 unsigned int anon_hash_shift;
 142 struct anon **anon_hash;

 144 static struct kmem_cache *anon_cache;
 145 static struct kmem_cache *anonmap_cache;

 147 pad_mutex_t     *anonhash_lock;

 149 /*
 150  * Used to make the increment of all refcnts of all anon slots of a large
 151  * page appear to be atomic.  The lock is grabbed for the first anon slot of
 152  * a large page.
 153  */
 154 pad_mutex_t     *anonpages_hash_lock;

 156 #define APH_MUTEX(vp, off)                              \
 157         (&anonpages_hash_lock[(ANON_HASH((vp), (off)) & \
 158             (AH_LOCK_SIZE - 1))].pad_mutex)

 160 #ifdef VM_STATS
 161 static struct anonvmstats_str {
 162         ulong_t getpages[30];
 163         ulong_t privatepages[10];
 164         ulong_t demotepages[9];
 165         ulong_t decrefpages[9];
 166         ulong_t dupfillholes[4];
 167         ulong_t freepages[1];
 168 } anonvmstats;
______unchanged_portion_omitted_

3551 void
3552 anon_array_enter(struct anon_map *amp, ulong_t an_idx, anon_sync_obj_t *sobj)
3553 {
3554         ulong_t         *ap_slot;
3555         kmutex_t        *mtx;
3556         kcondvar_t      *cv;
3557         int             hash;

3559         /*
3560          * Use szc to determine anon slot(s) to appear atomic.
3561          * If szc = 0, then lock the anon slot and mark it busy.
3562          * If szc > 0, then lock the range of slots by getting the
3563          * anon_array_lock for the first anon slot, and mark only the
3564          * first anon slot busy to represent whole range being busy.
3565          */
```



```
3567         ASSERT(RW_READ_HELD(&amp->a_rwlock));
3568         an_idx = P2ALIGN(an_idx, page_get_pagecnt(amp->a_szc));
3569         hash = ANON_ARRAY_HASH(amp, an_idx);
3570         sobj->sync_mutex = mtx = &anon_array_lock[hash].pad_mutex;
3571         sobj->sync_cv = cv = &anon_array_cv[hash];
3572         mutex_enter(mtx);
3573         ap_slot = anon_get_slot(amp->ahp, an_idx);
3574         while (ANON_ISBUSY(ap_slot))
3575                 cv_wait(cv, mtx);
3576         ANON_SETBUSY(ap_slot);
3577         sobj->sync_data = ap_slot;
3578         mutex_exit(mtx);
3579 }

3581 int
3582 anon_array_try_enter(struct anon_map *amp, ulong_t an_idx,
3583                         anon_sync_obj_t *sobj)
3584 {
3585         ulong_t         *ap_slot;
3586         kmutex_t        *mtx;
3587         int             hash;

3589         /*
3590          * Try to lock a range of anon slots.
3591          * Use szc to determine anon slot(s) to appear atomic.
3592          * If szc = 0, then lock the anon slot and mark it busy.
3593          * If szc > 0, then lock the range of slots by getting the
3594          * anon_array_lock for the first anon slot, and mark only the
3595          * first anon slot busy to represent whole range being busy.
3596          * Fail if the mutex or the anon_array are busy.
3597          */

3599         ASSERT(RW_READ_HELD(&amp->a_rwlock));
3600         an_idx = P2ALIGN(an_idx, page_get_pagecnt(amp->a_szc));
3601         hash = ANON_ARRAY_HASH(amp, an_idx);
3602         sobj->sync_mutex = mtx = &anon_array_lock[hash].pad_mutex;
3603         sobj->sync_cv = &anon_array_cv[hash];
3604         if (!mutex_tryenter(mtx)) {
3605                 return (EWOULDBLOCK);
3606         }
3607         ap_slot = anon_get_slot(amp->ahp, an_idx);
3608         if (ANON_ISBUSY(ap_slot)) {
3609                 mutex_exit(mtx);
3610                 return (EWOULDBLOCK);
3611         }
3612         ANON_SETBUSY(ap_slot);
3613         sobj->sync_data = ap_slot;
3614         mutex_exit(mtx);
3615         return (0);
3579 }
______unchanged_portion_omitted_
```

```
**********************************************************
   90474 Fri May  8 18:10:37 2015
new/usr/src/uts/common/vm/vm_as.c
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
use NULL getmemid segop as a shorthand for ENODEV
Instead of forcing every segment driver to implement a dummy function to
return (hopefully) ENODEV, handle NULL getmemid segop function pointer as
"return ENODEV" shorthand.
seg_inherit_notsup is redundant since segop_inherit checks for NULL properly
patch lower-case-segops
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
remove xhat
The xhat infrastructure was added to support hardware such as the zulu
graphics card - hardware which had on-board MMUs.  The VM used the xhat code
to keep the CPU's and Zulu's page tables in-sync.  Since the only xhat user
was zulu (which is gone), we can safely remove it simplifying the whole VM
subsystem.
Assorted notes:
- AS_BUSY flag was used solely by xhat
**********************************************************
   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 2010 Sun Microsystems, Inc.  All rights reserved.
  23  * Use is subject to license terms.
  24  * Copyright 2015, Joyent, Inc.  All rights reserved.
  25  */

  27 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
  28 /*        All Rights Reserved   */

  30 /*
  31  * University Copyright- Copyright (c) 1982, 1986, 1988
  32  * The Regents of the University of California
  33  * All Rights Reserved
  34  *
  35  * University Acknowledgment- Portions of this document are derived from
  36  * software developed by the University of California, Berkeley, and its
  37  * contributors.
  38  */
```



```
  40 /*
  41  * VM - address spaces.
  42  */

  44 #include <sys/types.h>
  45 #include <sys/t_lock.h>
  46 #include <sys/param.h>
  47 #include <sys/errno.h>
  48 #include <sys/systm.h>
  49 #include <sys/mman.h>
  50 #include <sys/sysmacros.h>
  51 #include <sys/cpuvar.h>
  52 #include <sys/sysinfo.h>
  53 #include <sys/kmem.h>
  54 #include <sys/vnode.h>
  55 #include <sys/vmsystm.h>
  56 #include <sys/cmn_err.h>
  57 #include <sys/debug.h>
  58 #include <sys/tnf_probe.h>
  59 #include <sys/vtrace.h>

  61 #include <vm/hat.h>
  62 #include <vm/xhat.h>
  62 #include <vm/as.h>
  63 #include <vm/seg.h>
  64 #include <vm/seg_vn.h>
  65 #include <vm/seg_dev.h>
  66 #include <vm/seg_kmem.h>
  67 #include <vm/seg_map.h>
  68 #include <vm/seg_spt.h>
  69 #include <vm/page.h>

  71 clock_t deadlk_wait = 1; /* number of ticks to wait before retrying */

  73 static struct kmem_cache *as_cache;

  75 static void as_setwatchprot(struct as *, caddr_t, size_t, uint_t);
  76 static void as_clearwatchprot(struct as *, caddr_t, size_t);
  77 int as_map_locked(struct as *, caddr_t, size_t, int ((*)()), void *);


  80 /*
  81  * Verifying the segment lists is very time-consuming; it may not be
  82  * desirable always to define VERIFY_SEGLIST when DEBUG is set.
  83  */
  84 #ifdef DEBUG
  85 #define VERIFY_SEGLIST
  86 int do_as_verify = 0;
  87 #endif

  89 /*
  90  * Allocate a new callback data structure entry and fill in the events of
  91  * interest, the address range of interest, and the callback argument.
  92  * Link the entry on the as->a_callbacks list. A callback entry for the
  93  * entire address space may be specified with vaddr = 0 and size = -1.
  94  *
  95  * CALLERS RESPONSIBILITY: If not calling from within the process context for
  96  * the specified as, the caller must guarantee persistence of the specified as
  97  * for the duration of this function (eg. pages being locked within the as
  98  * will guarantee persistence).
  99  */
 100 int
 101 as_add_callback(struct as *as, void (*cb_func)(), void *arg, uint_t events,
 102                 caddr_t vaddr, size_t size, int sleepflag)
 103 {
 104         struct as_callback      *current_head, *cb;
```

```
105         caddr_t                 saddr;
106         size_t                  rsize;

108         /* callback function and an event are mandatory */
109         if ((cb_func == NULL) || ((events & AS_ALL_EVENT) == 0))
110                 return (EINVAL);

112         /* Adding a callback after as_free has been called is not allowed */
113         if (as == &kas)
114                 return (ENOMEM);

116         /*
117          * vaddr = 0 and size = -1 is used to indicate that the callback range
118          * is the entire address space so no rounding is done in that case.
119          */
120         if (size != -1) {
121                 saddr = (caddr_t)((uintptr_t)vaddr & (uintptr_t)PAGEMASK);
122                 rsize = (((size_t)(vaddr + size) + PAGEOFFSET) & PAGEMASK) -
123                     (size_t)saddr;
124                 /* check for wraparound */
125                 if (saddr + rsize < saddr)
126                         return (ENOMEM);
127         } else {
128                 if (vaddr != 0)
129                         return (EINVAL);
130                 saddr = vaddr;
131                 rsize = size;
132         }

134         /* Allocate and initialize a callback entry */
135         cb = kmem_zalloc(sizeof (struct as_callback), sleepflag);
136         if (cb == NULL)
137                 return (EAGAIN);

139         cb->ascb_func = cb_func;
140         cb->ascb_arg = arg;
141         cb->ascb_events = events;
142         cb->ascb_saddr = saddr;
143         cb->ascb_len = rsize;

145         /* Add the entry to the list */
146         mutex_enter(&as->a_contents);
147         current_head = as->a_callbacks;
148         as->a_callbacks = cb;
149         cb->ascb_next = current_head;

151         /*
152          * The call to this function may lose in a race with
153          * a pertinent event - eg. a thread does long term memory locking
154          * but before the callback is added another thread executes as_unmap.
155          * A broadcast here resolves that.
156          */
157         if ((cb->ascb_events & AS_UNMAPWAIT_EVENT) && AS_ISUNMAPWAIT(as)) {
158                 AS_CLRUNMAPWAIT(as);
159                 cv_broadcast(&as->a_cv);
160         }

162         mutex_exit(&as->a_contents);
163         return (0);
164 }
```

unchanged_portion_omitted_

```
409 #endif /* VERIFY_SEGLIST */

411 /*
412  * Add a new segment to the address space. The avl_find()
413  * may be expensive so we attempt to use last segment accessed
```



```
414  * in as_gap() as an insertion point.
415  */
416 int
417 as_addseg(struct as  *as, struct seg *newseg)
418 {
419         struct seg *seg;
420         caddr_t addr;
421         caddr_t eaddr;
422         avl_index_t where;

424         ASSERT(AS_WRITE_HELD(as, &as->a_lock));

426         as->a_updatedir = 1;    /* inform /proc */
427         gethrestime(&as->a_updatetime);

429         if (as->a_lastgaphl != NULL) {
430                 struct seg *hseg = NULL;
431                 struct seg *lseg = NULL;

433                 if (as->a_lastgaphl->s_base > newseg->s_base) {
434                         hseg = as->a_lastgaphl;
435                         lseg = AVL_PREV(&as->a_segtree, hseg);
436                 } else {
437                         lseg = as->a_lastgaphl;
438                         hseg = AVL_NEXT(&as->a_segtree, lseg);
439                 }

441                 if (hseg && lseg && lseg->s_base < newseg->s_base &&
442                     hseg->s_base > newseg->s_base) {
443                         avl_insert_here(&as->a_segtree, newseg, lseg,
444                             AVL_AFTER);
445                         as->a_lastgaphl = NULL;
446                         as->a_seglast = newseg;
447                         return (0);
448                 }
449                 as->a_lastgaphl = NULL;
450         }

452         addr = newseg->s_base;
453         eaddr = addr + newseg->s_size;
454 again:

456         seg = avl_find(&as->a_segtree, &addr, &where);

458         if (seg == NULL)
459                 seg = avl_nearest(&as->a_segtree, where, AVL_AFTER);

461         if (seg == NULL)
462                 seg = avl_last(&as->a_segtree);

464         if (seg != NULL) {
465                 caddr_t base = seg->s_base;

467                 /*
468                  * If top of seg is below the requested address, then
469                  * the insertion point is at the end of the linked list,
470                  * and seg points to the tail of the list.  Otherwise,
471                  * the insertion point is immediately before seg.
472                  */
473                 if (base + seg->s_size > addr) {
474                         if (addr >= base || eaddr > base) {
475 #ifdef __sparc
476                                 extern const struct seg_ops segnf_ops;
477                                 extern struct seg_ops segnf_ops;

478                                 /*
```

```
                                 * no-fault segs must disappear if overlaid.
                                 * XXX need new segment type so
                                 * we don't have to check s_ops
                                 */
                                if (seg->s_ops == &segnf_ops) {
                                        seg_unmap(seg);
                                        goto again;
                                }
#endif
                                return (-1);    /* overlapping segment */
                        }
                }
        }
        as->a_seglast = newseg;
        avl_insert(&as->a_segtree, newseg, where);

#ifdef VERIFY_SEGLIST
        as_verify(as);
#endif
        return (0);
}
```

*unchanged_portion_omitted_*

```
/*
 * Allocate and initialize an address space data structure.
 * We call hat_alloc to allow any machine dependent
 * information in the hat structure to be initialized.
 */
struct as *
as_alloc(void)
{
        struct as *as;

        as = kmem_cache_alloc(as_cache, KM_SLEEP);

        as->a_flags             = 0;
        as->a_vbits             = 0;
        as->a_hrm               = NULL;
        as->a_seglast           = NULL;
        as->a_size              = 0;
        as->a_resvsize          = 0;
        as->a_updatedir         = 0;
        gethrestime(&as->a_updatetime);
        as->a_objectdir         = NULL;
        as->a_sizedir           = 0;
        as->a_userlimit         = (caddr_t)USERLIMIT;
        as->a_lastgap           = NULL;
        as->a_lastgaphl         = NULL;
        as->a_callbacks         = NULL;

        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
        as->a_hat = hat_alloc(as);      /* create hat for default system mmu */
        AS_LOCK_EXIT(as, &as->a_lock);

        as->a_xhat = NULL;

        return (as);
}

/*
 * Free an address space data structure.
 * Need to free the hat first and then
 * all the segments on this as and finally
 * the space for the as struct itself.
 */
void
```



```
as_free(struct as *as)
{
        struct hat *hat = as->a_hat;
        struct seg *seg, *next;
        boolean_t free_started = B_FALSE;
        int called = 0;

top:
        /*
         * Invoke ALL callbacks. as_do_callbacks will do one callback
         * per call, and not return (-1) until the callback has completed.
         * When as_do_callbacks returns zero, all callbacks have completed.
         */
        mutex_enter(&as->a_contents);
        while (as->a_callbacks && as_do_callbacks(as, AS_ALL_EVENT, 0, 0))
                ;

        /* This will prevent new XHATs from attaching to as */
        if (!called)
                AS_SETBUSY(as);
        mutex_exit(&as->a_contents);
        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

        if (!free_started) {
                free_started = B_TRUE;
        if (!called) {
                called = 1;
                hat_free_start(hat);
                if (as->a_xhat != NULL)
                        xhat_free_start_all(as);
        }
        for (seg = AS_SEGFIRST(as); seg != NULL; seg = next) {
                int err;

                next = AS_SEGNEXT(as, seg);
retry:
                err = segop_unmap(seg, seg->s_base, seg->s_size);
                err = SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
                if (err == EAGAIN) {
                        mutex_enter(&as->a_contents);
                        if (as->a_callbacks) {
                                AS_LOCK_EXIT(as, &as->a_lock);
                        } else if (!AS_ISNOUNMAPWAIT(as)) {
                                /*
                                 * Memory is currently locked. Wait for a
                                 * cv_signal that it has been unlocked, then
                                 * try the operation again.
                                 */
                                if (AS_ISUNMAPWAIT(as) == 0)
                                        cv_broadcast(&as->a_cv);
                                AS_SETUNMAPWAIT(as);
                                AS_LOCK_EXIT(as, &as->a_lock);
                                while (AS_ISUNMAPWAIT(as))
                                        cv_wait(&as->a_cv, &as->a_contents);
                        } else {
                                /*
                                 * We may have raced with
                                 * segvn_reclaim()/segspt_reclaim(). In this
                                 * case clean nounmapwait flag and retry since
                                 * softlockcnt in this segment may be already
                                 * 0.  We don't drop as writer lock so our
                                 * number of retries without sleeping should
                                 * be very small. See segvn_reclaim() for
                                 * more comments.
                                 */
                                AS_CLRNOUNMAPWAIT(as);
```

```
740                                 mutex_exit(&as->a_contents);
741                                 goto retry;
742                         }
743                         mutex_exit(&as->a_contents);
744                         goto top;
745                 } else {
746                         /*
747                          * We do not expect any other error return at this
748                          * time. This is similar to an ASSERT in seg_unmap()
749                          */
750                         ASSERT(err == 0);
751                 }
752         }
753         hat_free_end(hat);
762         if (as->a_xhat != NULL)
763                 xhat_free_end_all(as);
754         AS_LOCK_EXIT(as, &as->a_lock);

756         /* /proc stuff */
757         ASSERT(avl_numnodes(&as->a_wpage) == 0);
758         if (as->a_objectdir) {
759                 kmem_free(as->a_objectdir, as->a_sizedir * sizeof (vnode_t *));
760                 as->a_objectdir = NULL;
761                 as->a_sizedir = 0;
762         }

764         /*
765          * Free the struct as back to kmem.  Assert it has no segments.
766          */
767         ASSERT(avl_numnodes(&as->a_segtree) == 0);
768         kmem_cache_free(as_cache, as);
769 }

771 int
772 as_dup(struct as *as, struct proc *forkedproc)
773 {
774         struct as *newas;
775         struct seg *seg, *newseg;
776         size_t  purgesize = 0;
777         int error;

779         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
780         as_clearwatch(as);
781         newas = as_alloc();
782         newas->a_userlimit = as->a_userlimit;
783         newas->a_proc = forkedproc;

785         AS_LOCK_ENTER(newas, &newas->a_lock, RW_WRITER);

797         /* This will prevent new XHATs from attaching */
798         mutex_enter(&as->a_contents);
799         AS_SETBUSY(as);
800         mutex_exit(&as->a_contents);
801         mutex_enter(&newas->a_contents);
802         AS_SETBUSY(newas);
803         mutex_exit(&newas->a_contents);

787         (void) hat_dup(as->a_hat, newas->a_hat, NULL, 0, HAT_DUP_SRD);

789         for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {

791                 if (seg->s_flags & S_PURGE) {
792                         purgesize += seg->s_size;
793                         continue;
794                 }
```



```
796                 newseg = seg_alloc(newas, seg->s_base, seg->s_size);
797                 if (newseg == NULL) {
798                         AS_LOCK_EXIT(newas, &newas->a_lock);
799                         as_setwatch(as);
818                         mutex_enter(&as->a_contents);
819                         AS_CLRBUSY(as);
820                         mutex_exit(&as->a_contents);
800                         AS_LOCK_EXIT(as, &as->a_lock);
801                         as_free(newas);
802                         return (-1);
803                 }
804                 if ((error = segop_dup(seg, newseg)) != 0) {
825                 if ((error = SEGOP_DUP(seg, newseg)) != 0) {
805                         /*
806                          * We call seg_free() on the new seg
807                          * because the segment is not set up
808                          * completely; i.e. it has no ops.
809                          */
810                         as_setwatch(as);
832                         mutex_enter(&as->a_contents);
833                         AS_CLRBUSY(as);
834                         mutex_exit(&as->a_contents);
811                         AS_LOCK_EXIT(as, &as->a_lock);
812                         seg_free(newseg);
813                         AS_LOCK_EXIT(newas, &newas->a_lock);
814                         as_free(newas);
815                         return (error);
816                 }
817                 newas->a_size += seg->s_size;
818         }
819         newas->a_resvsize = as->a_resvsize - purgesize;

821         error = hat_dup(as->a_hat, newas->a_hat, NULL, 0, HAT_DUP_ALL);
846         if (as->a_xhat != NULL)
847                 error |= xhat_dup_all(as, newas, NULL, 0, HAT_DUP_ALL);

849         mutex_enter(&newas->a_contents);
850         AS_CLRBUSY(newas);
851         mutex_exit(&newas->a_contents);
823         AS_LOCK_EXIT(newas, &newas->a_lock);

825         as_setwatch(as);
855         mutex_enter(&as->a_contents);
856         AS_CLRBUSY(as);
857         mutex_exit(&as->a_contents);
826         AS_LOCK_EXIT(as, &as->a_lock);
827         if (error != 0) {
828                 as_free(newas);
829                 return (error);
830         }
831         forkedproc->p_as = newas;
832         return (0);
833 }

835 /*
836  * Handle a ''fault'' at addr for size bytes.
837  */
838 faultcode_t
839 as_fault(struct hat *hat, struct as *as, caddr_t addr, size_t size,
840         enum fault_type type, enum seg_rw rw)
841 {
842         struct seg *seg;
843         caddr_t raddr;                  /* rounded down addr */
844         size_t rsize;                   /* rounded up size */
845         size_t ssize;
846         faultcode_t res = 0;
```

```
 847         caddr_t addrsav;
 848         struct seg *segsav;
 849         int as_lock_held;
 850         klwp_t *lwp = ttolwp(curthread);
 883         int is_xhat = 0;
 851         int holding_wpage = 0;
 885         extern struct seg_ops   segdev_ops;



 889         if (as->a_hat != hat) {
 890                 /* This must be an XHAT then */
 891                 is_xhat = 1;

 893                 if ((type != F_INVAL) || (as == &kas))
 894                         return (FC_NOSUPPORT);
 895         }

 855 retry:
 898         if (!is_xhat) {
 856         /*
 857          * Indicate that the lwp is not to be stopped while waiting for a
 858          * pagefault.  This is to avoid deadlock while debugging a process
 859          * via /proc over NFS (in particular).
 900                  * Indicate that the lwp is not to be stopped while waiting
 901                  * for a pagefault.  This is to avoid deadlock while debugging
 902                  * a process via /proc over NFS (in particular).
 860          */
 861         if (lwp != NULL)
 862                 lwp->lwp_nostop++;

 864         /*
 865          * same length must be used when we softlock and softunlock.  We
 866          * don't support softunlocking lengths less than the original length
 867          * when there is largepage support.  See seg_dev.c for more
 868          * comments.
 908                  * same length must be used when we softlock and softunlock.
 909                  * We don't support softunlocking lengths less than
 910                  * the original length when there is largepage support.
 911                  * See seg_dev.c for more comments.
 869          */
 870         switch (type) {

 872         case F_SOFTLOCK:
 873                 CPU_STATS_ADD_K(vm, softlock, 1);
 874                 break;

 876         case F_SOFTUNLOCK:
 877                 break;

 879         case F_PROT:
 880                 CPU_STATS_ADD_K(vm, prot_fault, 1);
 881                 break;

 883         case F_INVAL:
 884                 CPU_STATS_ENTER_K();
 885                 CPU_STATS_ADDQ(CPU, vm, as_fault, 1);
 886                 if (as == &kas)
 887                         CPU_STATS_ADDQ(CPU, vm, kernel_asflt, 1);
 888                 CPU_STATS_EXIT_K();
 889                 break;
 890         }
 934         }

 892         /* Kernel probe */
 893         TNF_PROBE_3(address_fault, "vm pagefault", /* CSTYLED */,
```



```
 894             tnf_opaque, address,        addr,
 895             tnf_fault_type,     fault_type,     type,
 896             tnf_seg_access,     access,         rw);

 898         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
 899         rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
 900             (size_t)raddr;

 902         /*
 903          * XXX -- Don't grab the as lock for segkmap. We should grab it for
 904          * correctness, but then we could be stuck holding this lock for
 905          * a LONG time if the fault needs to be resolved on a slow
 906          * filesystem, and then no-one will be able to exec new commands,
 907          * as exec'ing requires the write lock on the as.
 908          */
 909         if (as == &kas && segkmap && segkmap->s_base <= raddr &&
 910             raddr + size < segkmap->s_base + segkmap->s_size) {
 955                 /*
 956                  * if (as==&kas), this can't be XHAT: we've already returned
 957                  * FC_NOSUPPORT.
 958                  */
 911                 seg = segkmap;
 912                 as_lock_held = 0;
 913         } else {
 914                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
 963                 if (is_xhat && avl_numnodes(&as->a_wpage) != 0) {
 964                         /*
 965                          * Grab and hold the writers' lock on the as
 966                          * if the fault is to a watched page.
 967                          * This will keep CPUs from "peeking" at the
 968                          * address range while we're temporarily boosting
 969                          * the permissions for the XHAT device to
 970                          * resolve the fault in the segment layer.
 971                          *
 972                          * We could check whether faulted address
 973                          * is within a watched page and only then grab
 974                          * the writer lock, but this is simpler.
 975                          */
 976                         AS_LOCK_EXIT(as, &as->a_lock);
 977                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
 978                 }

 916                 seg = as_segat(as, raddr);
 917                 if (seg == NULL) {
 918                         AS_LOCK_EXIT(as, &as->a_lock);
 919                         if (lwp != NULL)
 983                         if ((lwp != NULL) && (!is_xhat))
 920                                 lwp->lwp_nostop--;
 921                         return (FC_NOMAP);
 922                 }

 924                 as_lock_held = 1;
 925         }

 927         addrsav = raddr;
 928         segsav = seg;

 930         for (; rsize != 0; rsize -= ssize, raddr += ssize) {
 931                 if (raddr >= seg->s_base + seg->s_size) {
 932                         seg = AS_SEGNEXT(as, seg);
 933                         if (seg == NULL || raddr != seg->s_base) {
 934                                 res = FC_NOMAP;
 935                                 break;
 936                         }
 937                 }
 938                 if (raddr + rsize > seg->s_base + seg->s_size)
```

```
 939                         ssize = seg->s_base + seg->s_size - raddr;
 940                 else
 941                         ssize = rsize;

 943                 res = segop_fault(hat, seg, raddr, ssize, type, rw);
1007                 if (!is_xhat || (seg->s_ops != &segdev_ops)) {

1009                         if (is_xhat && avl_numnodes(&as->a_wpage) != 0 &&
1010                             pr_is_watchpage_as(raddr, rw, as)) {
1011                                 /*
1012                                  * Handle watch pages.  If we're faulting on a
1013                                  * watched page from an X-hat, we have to
1014                                  * restore the original permissions while we
1015                                  * handle the fault.
1016                                  */
1017                                 as_clearwatch(as);
1018                                 holding_wpage = 1;
1019                         }

1021                         res = SEGOP_FAULT(hat, seg, raddr, ssize, type, rw);

 945                 /* Restore watchpoints */
 946                 if (holding_wpage) {
 947                         as_setwatch(as);
 948                         holding_wpage = 0;
 949                 }

 951                 if (res != 0)
 952                         break;
1031                 } else {
1032                         /* XHAT does not support seg_dev */
1033                         res = FC_NOSUPPORT;
1034                         break;
1035                 }
 953         }

 955         /*
 956          * If we were SOFTLOCKing and encountered a failure,
 957          * we must SOFTUNLOCK the range we already did. (Maybe we
 958          * should just panic if we are SOFTLOCKing or even SOFTUNLOCKing
 959          * right here...)
 960          */
 961         if (res != 0 && type == F_SOFTLOCK) {
 962                 for (seg = segsav; addrsav < raddr; addrsav += ssize) {
 963                         if (addrsav >= seg->s_base + seg->s_size)
 964                                 seg = AS_SEGNEXT(as, seg);
 965                         ASSERT(seg != NULL);
 966                         /*
 967                          * Now call the fault routine again to perform the
 968                          * unlock using S_OTHER instead of the rw variable
 969                          * since we never got a chance to touch the pages.
 970                          */
 971                         if (raddr > seg->s_base + seg->s_size)
 972                                 ssize = seg->s_base + seg->s_size - addrsav;
 973                         else
 974                                 ssize = raddr - addrsav;
 975                         (void) segop_fault(hat, seg, addrsav, ssize,
1058                         (void) SEGOP_FAULT(hat, seg, addrsav, ssize,
 976                             F_SOFTUNLOCK, S_OTHER);
 977                 }
 978         }
 979         if (as_lock_held)
 980                 AS_LOCK_EXIT(as, &as->a_lock);
 981         if (lwp != NULL)
1064         if ((lwp != NULL) && (!is_xhat))
 982                 lwp->lwp_nostop--;
```



```
 984         /*
 985          * If the lower levels returned EDEADLK for a fault,
 986          * It means that we should retry the fault.  Let's wait
 987          * a bit also to let the deadlock causing condition clear.
 988          * This is part of a gross hack to work around a design flaw
 989          * in the ufs/sds logging code and should go away when the
 990          * logging code is re-designed to fix the problem. See bug
 991          * 4125102 for details of the problem.
 992          */
 993         if (FC_ERRNO(res) == EDEADLK) {
 994                 delay(deadlk_wait);
 995                 res = 0;
 996                 goto retry;
 997         }
 998         return (res);
 999 }



1003 /*
1004  * Asynchronous ''fault'' at addr for size bytes.
1005  */
1006 faultcode_t
1007 as_faulta(struct as *as, caddr_t addr, size_t size)
1008 {
1009         struct seg *seg;
1010         caddr_t raddr;                  /* rounded down addr */
1011         size_t rsize;                   /* rounded up size */
1012         faultcode_t res = 0;
1013         klwp_t *lwp = ttolwp(curthread);

1015 retry:
1016         /*
1017          * Indicate that the lwp is not to be stopped while waiting
1018          * for a pagefault.  This is to avoid deadlock while debugging
1019          * a process via /proc over NFS (in particular).
1020          */
1021         if (lwp != NULL)
1022                 lwp->lwp_nostop++;

1024         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1025         rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
1026             (size_t)raddr;

1028         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1029         seg = as_segat(as, raddr);
1030         if (seg == NULL) {
1031                 AS_LOCK_EXIT(as, &as->a_lock);
1032                 if (lwp != NULL)
1033                         lwp->lwp_nostop--;
1034                 return (FC_NOMAP);
1035         }

1037         for (; rsize != 0; rsize -= PAGESIZE, raddr += PAGESIZE) {
1038                 if (raddr >= seg->s_base + seg->s_size) {
1039                         seg = AS_SEGNEXT(as, seg);
1040                         if (seg == NULL || raddr != seg->s_base) {
1041                                 res = FC_NOMAP;
1042                                 break;
1043                         }
1044                 }
1045                 res = segop_faulta(seg, raddr);
1128                 res = SEGOP_FAULTA(seg, raddr);
1046                 if (res != 0)
1047                         break;
```

```
1048         }
1049         AS_LOCK_EXIT(as, &as->a_lock);
1050         if (lwp != NULL)
1051                 lwp->lwp_nostop--;
1052         /*
1053          * If the lower levels returned EDEADLK for a fault,
1054          * It means that we should retry the fault.  Let's wait
1055          * a bit also to let the deadlock causing condition clear.
1056          * This is part of a gross hack to work around a design flaw
1057          * in the ufs/sds logging code and should go away when the
1058          * logging code is re-designed to fix the problem. See bug
1059          * 4125102 for details of the problem.
1060          */
1061         if (FC_ERRNO(res) == EDEADLK) {
1062                 delay(deadlk_wait);
1063                 res = 0;
1064                 goto retry;
1065         }
1066         return (res);
1067 }

1069 /*
1070  * Set the virtual mapping for the interval from [addr : addr + size)
1071  * in address space 'as' to have the specified protection.
1072  * It is ok for the range to cross over several segments,
1073  * as long as they are contiguous.
1074  */
1075 int
1076 as_setprot(struct as *as, caddr_t addr, size_t size, uint_t prot)
1077 {
1078         struct seg *seg;
1079         struct as_callback *cb;
1080         size_t ssize;
1081         caddr_t raddr;                  /* rounded down addr */
1082         size_t rsize;                   /* rounded up size */
1083         int error = 0, writer = 0;
1084         caddr_t saveraddr;
1085         size_t saversize;

1087 setprot_top:
1088         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1089         rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
1090             (size_t)raddr;

1092         if (raddr + rsize < raddr)              /* check for wraparound */
1093                 return (ENOMEM);

1095         saveraddr = raddr;
1096         saversize = rsize;

1098         /*
1099          * Normally we only lock the as as a reader. But
1100          * if due to setprot the segment driver needs to split
1101          * a segment it will return IE_RETRY. Therefore we re-acquire
1102          * the as lock as a writer so the segment driver can change
1103          * the seg list. Also the segment driver will return IE_RETRY
1104          * after it has changed the segment list so we therefore keep
1105          * locking as a writer. Since these opeartions should be rare
1106          * want to only lock as a writer when necessary.
1107          */
1108         if (writer || avl_numnodes(&as->a_wpage) != 0) {
1109                 AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1110         } else {
1111                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1112         }
```



```
1114         as_clearwatchprot(as, raddr, rsize);
1115         seg = as_segat(as, raddr);
1116         if (seg == NULL) {
1117                 as_setwatch(as);
1118                 AS_LOCK_EXIT(as, &as->a_lock);
1119                 return (ENOMEM);
1120         }

1122         for (; rsize != 0; rsize -= ssize, raddr += ssize) {
1123                 if (raddr >= seg->s_base + seg->s_size) {
1124                         seg = AS_SEGNEXT(as, seg);
1125                         if (seg == NULL || raddr != seg->s_base) {
1126                                 error = ENOMEM;
1127                                 break;
1128                         }
1129                 }
1130                 if ((raddr + rsize) > (seg->s_base + seg->s_size))
1131                         ssize = seg->s_base + seg->s_size - raddr;
1132                 else
1133                         ssize = rsize;
1134 retry:
1135                 error = segop_setprot(seg, raddr, ssize, prot);
1218                 error = SEGOP_SETPROT(seg, raddr, ssize, prot);

1137                 if (error == IE_NOMEM) {
1138                         error = EAGAIN;
1139                         break;
1140                 }

1142                 if (error == IE_RETRY) {
1143                         AS_LOCK_EXIT(as, &as->a_lock);
1144                         writer = 1;
1145                         goto setprot_top;
1146                 }

1148                 if (error == EAGAIN) {
1149                         /*
1150                          * Make sure we have a_lock as writer.
1151                          */
1152                         if (writer == 0) {
1153                                 AS_LOCK_EXIT(as, &as->a_lock);
1154                                 writer = 1;
1155                                 goto setprot_top;
1156                         }

1158                         /*
1159                          * Memory is currently locked.  It must be unlocked
1160                          * before this operation can succeed through a retry.
1161                          * The possible reasons for locked memory and
1162                          * corresponding strategies for unlocking are:
1163                          * (1) Normal I/O
1164                          *      wait for a signal that the I/O operation
1165                          *      has completed and the memory is unlocked.
1166                          * (2) Asynchronous I/O
1167                          *      The aio subsystem does not unlock pages when
1168                          *      the I/O is completed. Those pages are unlocked
1169                          *      when the application calls aiowait/aioerror.
1170                          *      So, to prevent blocking forever, cv_broadcast()
1171                          *      is done to wake up aio_cleanup_thread.
1172                          *      Subsequently, segvn_reclaim will be called, and
1173                          *      that will do AS_CLRUNMAPWAIT() and wake us up.
1174                          * (3) Long term page locking:
1175                          *      Drivers intending to have pages locked for a
1176                          *      period considerably longer than for normal I/O
1177                          *      (essentially forever) may have registered for a
1178                          *      callback so they may unlock these pages on
```

```
1179                          *      request. This is needed to allow this operation
1180                          *      to succeed. Each entry on the callback list is
1181                          *      examined. If the event or address range pertains
1182                          *      the callback is invoked (unless it already is in
1183                          *      progress). The a_contents lock must be dropped
1184                          *      before the callback, so only one callback can
1185                          *      be done at a time. Go to the top and do more
1186                          *      until zero is returned. If zero is returned,
1187                          *      either there were no callbacks for this event
1188                          *      or they were already in progress.
1189                          */
1190                         mutex_enter(&as->a_contents);
1191                         if (as->a_callbacks &&
1192                             (cb = as_find_callback(as, AS_SETPROT_EVENT,
1193                             seg->s_base, seg->s_size))) {
1194                                 AS_LOCK_EXIT(as, &as->a_lock);
1195                                 as_execute_callback(as, cb, AS_SETPROT_EVENT);
1196                         } else if (!AS_ISNOUNMAPWAIT(as)) {
1197                                 if (AS_ISUNMAPWAIT(as) == 0)
1198                                         cv_broadcast(&as->a_cv);
1199                                 AS_SETUNMAPWAIT(as);
1200                                 AS_LOCK_EXIT(as, &as->a_lock);
1201                                 while (AS_ISUNMAPWAIT(as))
1202                                         cv_wait(&as->a_cv, &as->a_contents);
1203                         } else {
1204                                 /*
1205                                  * We may have raced with
1206                                  * segvn_reclaim()/segspt_reclaim(). In this
1207                                  * case clean nounmapwait flag and retry since
1208                                  * softlockcnt in this segment may be already
1209                                  * 0.  We don't drop as writer lock so our
1210                                  * number of retries without sleeping should
1211                                  * be very small. See segvn_reclaim() for
1212                                  * more comments.
1213                                  */
1214                                 AS_CLRNOUNMAPWAIT(as);
1215                                 mutex_exit(&as->a_contents);
1216                                 goto retry;
1217                         }
1218                         mutex_exit(&as->a_contents);
1219                         goto setprot_top;
1220                 } else if (error != 0)
1221                         break;
1222         }
1223         if (error != 0) {
1224                 as_setwatch(as);
1225         } else {
1226                 as_setwatchprot(as, saveraddr, saversize, prot);
1227         }
1228         AS_LOCK_EXIT(as, &as->a_lock);
1229         return (error);
1230 }

1232 /*
1233  * Check to make sure that the interval [addr, addr + size)
1234  * in address space 'as' has at least the specified protection.
1235  * It is ok for the range to cross over several segments, as long
1236  * as they are contiguous.
1237  */
1238 int
1239 as_checkprot(struct as *as, caddr_t addr, size_t size, uint_t prot)
1240 {
1241         struct seg *seg;
1242         size_t ssize;
1243         caddr_t raddr;                  /* rounded down addr */
1244         size_t rsize;                   /* rounded up size */
```



```
1245         int error = 0;

1247         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1248         rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
1249             (size_t)raddr;

1251         if (raddr + rsize < raddr)              /* check for wraparound */
1252                 return (ENOMEM);

1254         /*
1255          * This is ugly as sin...
1256          * Normally, we only acquire the address space readers lock.
1257          * However, if the address space has watchpoints present,
1258          * we must acquire the writer lock on the address space for
1259          * the benefit of as_clearwatchprot() and as_setwatchprot().
1260          */
1261         if (avl_numnodes(&as->a_wpage) != 0)
1262                 AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1263         else
1264                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1265         as_clearwatchprot(as, raddr, rsize);
1266         seg = as_segat(as, raddr);
1267         if (seg == NULL) {
1268                 as_setwatch(as);
1269                 AS_LOCK_EXIT(as, &as->a_lock);
1270                 return (ENOMEM);
1271         }

1273         for (; rsize != 0; rsize -= ssize, raddr += ssize) {
1274                 if (raddr >= seg->s_base + seg->s_size) {
1275                         seg = AS_SEGNEXT(as, seg);
1276                         if (seg == NULL || raddr != seg->s_base) {
1277                                 error = ENOMEM;
1278                                 break;
1279                         }
1280                 }
1281                 if ((raddr + rsize) > (seg->s_base + seg->s_size))
1282                         ssize = seg->s_base + seg->s_size - raddr;
1283                 else
1284                         ssize = rsize;

1286                 error = segop_checkprot(seg, raddr, ssize, prot);
1369                 error = SEGOP_CHECKPROT(seg, raddr, ssize, prot);
1287                 if (error != 0)
1288                         break;
1289         }
1290         as_setwatch(as);
1291         AS_LOCK_EXIT(as, &as->a_lock);
1292         return (error);
1293 }

1295 int
1296 as_unmap(struct as *as, caddr_t addr, size_t size)
1297 {
1298         struct seg *seg, *seg_next;
1299         struct as_callback *cb;
1300         caddr_t raddr, eaddr;
1301         size_t ssize, rsize = 0;
1302         int err;

1304 top:
1305         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1306         eaddr = (caddr_t)(((uintptr_t)(addr + size) + PAGEOFFSET) &
1307             (uintptr_t)PAGEMASK);

1309         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
```

```
1311         as->a_updatedir = 1;    /* inform /proc */
1312         gethrestime(&as->a_updatetime);
1313
1314         /*
1315          * Use as_findseg to find the first segment in the range, then
1316          * step through the segments in order, following s_next.
1317          */
1318         as_clearwatchprot(as, raddr, eaddr - raddr);
1319
1320         for (seg = as_findseg(as, raddr, 0); seg != NULL; seg = seg_next) {
1321                 if (eaddr <= seg->s_base)
1322                         break;          /* eaddr was in a gap; all done */
1323
1324                 /* this is implied by the test above */
1325                 ASSERT(raddr < eaddr);
1326
1327                 if (raddr < seg->s_base)
1328                         raddr = seg->s_base;    /* raddr was in a gap */
1329
1330                 if (eaddr > (seg->s_base + seg->s_size))
1331                         ssize = seg->s_base + seg->s_size - raddr;
1332                 else
1333                         ssize = eaddr - raddr;
1334
1335                 /*
1336                  * Save next segment pointer since seg can be
1337                  * destroyed during the segment unmap operation.
1338                  */
1339                 seg_next = AS_SEGNEXT(as, seg);
1340
1341                 /*
1342                  * We didn't count /dev/null mappings, so ignore them here.
1343                  * We'll handle MAP_NORESERVE cases in segvn_unmap(). (Again,
1344                  * we have to do this check here while we have seg.)
1345                  */
1346                 rsize = 0;
1347                 if (!SEG_IS_DEVNULL_MAPPING(seg) &&
1348                     !SEG_IS_PARTIAL_RESV(seg))
1349                         rsize = ssize;
1350
1351 retry:
1352                 err = segop_unmap(seg, raddr, ssize);
1435                 err = SEGOP_UNMAP(seg, raddr, ssize);
1353                 if (err == EAGAIN) {
1354                         /*
1355                          * Memory is currently locked.  It must be unlocked
1356                          * before this operation can succeed through a retry.
1357                          * The possible reasons for locked memory and
1358                          * corresponding strategies for unlocking are:
1359                          * (1) Normal I/O
1360                          *      wait for a signal that the I/O operation
1361                          *      has completed and the memory is unlocked.
1362                          * (2) Asynchronous I/O
1363                          *      The aio subsystem does not unlock pages when
1364                          *      the I/O is completed. Those pages are unlocked
1365                          *      when the application calls aiowait/aioerror.
1366                          *      So, to prevent blocking forever, cv_broadcast()
1367                          *      is done to wake up aio_cleanup_thread.
1368                          *      Subsequently, segvn_reclaim will be called, and
1369                          *      that will do AS_CLRUNMAPWAIT() and wake us up.
1370                          * (3) Long term page locking:
1371                          *      Drivers intending to have pages locked for a
1372                          *      period considerably longer than for normal I/O
1373                          *      (essentially forever) may have registered for a
1374                          *      callback so they may unlock these pages on
```



```
1375                          *      request. This is needed to allow this operation
1376                          *      to succeed. Each entry on the callback list is
1377                          *      examined. If the event or address range pertains
1378                          *      the callback is invoked (unless it already is in
1379                          *      progress). The a_contents lock must be dropped
1380                          *      before the callback, so only one callback can
1381                          *      be done at a time. Go to the top and do more
1382                          *      until zero is returned. If zero is returned,
1383                          *      either there were no callbacks for this event
1384                          *      or they were already in progress.
1385                          */
1386                         mutex_enter(&as->a_contents);
1387                         if (as->a_callbacks &&
1388                             (cb = as_find_callback(as, AS_UNMAP_EVENT,
1389                             seg->s_base, seg->s_size))) {
1390                                 AS_LOCK_EXIT(as, &as->a_lock);
1391                                 as_execute_callback(as, cb, AS_UNMAP_EVENT);
1392                         } else if (!AS_ISNOUNMAPWAIT(as)) {
1393                                 if (AS_ISUNMAPWAIT(as) == 0)
1394                                         cv_broadcast(&as->a_cv);
1395                                 AS_SETUNMAPWAIT(as);
1396                                 AS_LOCK_EXIT(as, &as->a_lock);
1397                                 while (AS_ISUNMAPWAIT(as))
1398                                         cv_wait(&as->a_cv, &as->a_contents);
1399                         } else {
1400                                 /*
1401                                  * We may have raced with
1402                                  * segvn_reclaim()/segspt_reclaim(). In this
1403                                  * case clean nounmapwait flag and retry since
1404                                  * softlockcnt in this segment may be already
1405                                  * 0.  We don't drop as writer lock so our
1406                                  * number of retries without sleeping should
1407                                  * be very small. See segvn_reclaim() for
1408                                  * more comments.
1409                                  */
1410                                 AS_CLRNOUNMAPWAIT(as);
1411                                 mutex_exit(&as->a_contents);
1412                                 goto retry;
1413                         }
1414                         mutex_exit(&as->a_contents);
1415                         goto top;
1416                 } else if (err == IE_RETRY) {
1417                         AS_LOCK_EXIT(as, &as->a_lock);
1418                         goto top;
1419                 } else if (err) {
1420                         as_setwatch(as);
1421                         AS_LOCK_EXIT(as, &as->a_lock);
1422                         return (-1);
1423                 }
1424
1425                 as->a_size -= ssize;
1426                 if (rsize)
1427                         as->a_resvsize -= rsize;
1428                 raddr += ssize;
1429         }
1430         AS_LOCK_EXIT(as, &as->a_lock);
1431         return (0);
1432 }
_____unchanged_portion_omitted_


1765 /*
1766  * Delete all segments in the address space marked with S_PURGE.
1767  * This is currently used for Sparc V9 nofault ASI segments (seg_nf.c).
1768  * These segments are deleted as a first step before calls to as_gap(), so
1769  * that they don't affect mmap() or shmat().
```

```
1770  */
1771 void
1772 as_purge(struct as *as)
1773 {
1774         struct seg *seg;
1775         struct seg *next_seg;

1777         /*
1778          * the setting of NEEDSPURGE is protect by as_rangelock(), so
1779          * no need to grab a_contents mutex for this check
1780          */
1781         if ((as->a_flags & AS_NEEDSPURGE) == 0)
1782                 return;

1784         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1785         next_seg = NULL;
1786         seg = AS_SEGFIRST(as);
1787         while (seg != NULL) {
1788                 next_seg = AS_SEGNEXT(as, seg);
1789                 if (seg->s_flags & S_PURGE)
1790                         segop_unmap(seg, seg->s_base, seg->s_size);
1873                         SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
1791                 seg = next_seg;
1792         }
1793         AS_LOCK_EXIT(as, &as->a_lock);

1795         mutex_enter(&as->a_contents);
1796         as->a_flags &= ~AS_NEEDSPURGE;
1797         mutex_exit(&as->a_contents);
1798 }
_____unchanged_portion_omitted_

2000 /*
2001  * Return the next range within [base, base + len) that is backed
2002  * with "real memory".  Skip holes and non-seg_vn segments.
2003  * We're lazy and only return one segment at a time.
2004  */
2005 int
2006 as_memory(struct as *as, caddr_t *basep, size_t *lenp)
2007 {
2008         extern const struct seg_ops segspt_shmops;      /* needs a header file *
2091         extern struct seg_ops segspt_shmops;    /* needs a header file */
2009         struct seg *seg;
2010         caddr_t addr, eaddr;
2011         caddr_t segend;

2013         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

2015         addr = *basep;
2016         eaddr = addr + *lenp;

2018         seg = as_findseg(as, addr, 0);
2019         if (seg != NULL)
2020                 addr = MAX(seg->s_base, addr);

2022         for (;;) {
2023                 if (seg == NULL || addr >= eaddr || eaddr <= seg->s_base) {
2024                         AS_LOCK_EXIT(as, &as->a_lock);
2025                         return (EINVAL);
2026                 }

2028                 if (seg->s_ops == &segvn_ops) {
2029                         segend = seg->s_base + seg->s_size;
2030                         break;
2031                 }
```



```
2033                 /*
2034                  * We do ISM by looking into the private data
2035                  * to determine the real size of the segment.
2036                  */
2037                 if (seg->s_ops == &segspt_shmops) {
2038                         segend = seg->s_base + spt_realsize(seg);
2039                         if (addr < segend)
2040                                 break;
2041                 }

2043                 seg = AS_SEGNEXT(as, seg);

2045                 if (seg != NULL)
2046                         addr = seg->s_base;
2047         }

2049         *basep = addr;

2051         if (segend > eaddr)
2052                 *lenp = eaddr - addr;
2053         else
2054                 *lenp = segend - addr;

2056         AS_LOCK_EXIT(as, &as->a_lock);
2057         return (0);
2058 }

2060 /*
2144  * Swap the pages associated with the address space as out to
2145  * secondary storage, returning the number of bytes actually
2146  * swapped.
2147  *
2148  * The value returned is intended to correlate well with the process's
2149  * memory requirements.  Its usefulness for this purpose depends on
2150  * how well the segment-level routines do at returning accurate
2151  * information.
2152  */
2153 size_t
2154 as_swapout(struct as *as)
2155 {
2156         struct seg *seg;
2157         size_t swpcnt = 0;

2159         /*
2160          * Kernel-only processes have given up their address
2161          * spaces.  Of course, we shouldn't be attempting to
2162          * swap out such processes in the first place...
2163          */
2164         if (as == NULL)
2165                 return (0);

2167         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

2169         /* Prevent XHATs from attaching */
2170         mutex_enter(&as->a_contents);
2171         AS_SETBUSY(as);
2172         mutex_exit(&as->a_contents);


2175         /*
2176          * Free all mapping resources associated with the address
2177          * space.  The segment-level swapout routines capitalize
2178          * on this unmapping by scavanging pages that have become
2179          * unmapped here.
2180          */
2181         hat_swapout(as->a_hat);
```

```
2182         if (as->a_xhat != NULL)
2183                 xhat_swapout_all(as);

2185         mutex_enter(&as->a_contents);
2186         AS_CLRBUSY(as);
2187         mutex_exit(&as->a_contents);

2189         /*
2190          * Call the swapout routines of all segments in the address
2191          * space to do the actual work, accumulating the amount of
2192          * space reclaimed.
2193          */
2194         for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {
2195                 struct seg_ops *ov = seg->s_ops;

2197                 /*
2198                  * We have to check to see if the seg has
2199                  * an ops vector because the seg may have
2200                  * been in the middle of being set up when
2201                  * the process was picked for swapout.
2202                  */
2203                 if ((ov != NULL) && (ov->swapout != NULL))
2204                         swpcnt += SEGOP_SWAPOUT(seg);
2205         }
2206         AS_LOCK_EXIT(as, &as->a_lock);
2207         return (swpcnt);
2208 }

2210 /*
2061  * Determine whether data from the mappings in interval [addr, addr + size)
2062  * are in the primary memory (core) cache.
2063  */
2064 int
2065 as_incore(struct as *as, caddr_t addr,
2066     size_t size, char *vec, size_t *sizep)
2067 {
2068         struct seg *seg;
2069         size_t ssize;
2070         caddr_t raddr;          /* rounded down addr */
2071         size_t rsize;           /* rounded up size */
2072         size_t isize;                   /* iteration size */
2073         int error = 0;          /* result, assume success */

2075         *sizep = 0;
2076         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2077         rsize = ((((size_t)addr + size) + PAGEOFFSET) & PAGEMASK) -
2078             (size_t)raddr;

2080         if (raddr + rsize < raddr)              /* check for wraparound */
2081                 return (ENOMEM);

2083         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
2084         seg = as_segat(as, raddr);
2085         if (seg == NULL) {
2086                 AS_LOCK_EXIT(as, &as->a_lock);
2087                 return (-1);
2088         }

2090         for (; rsize != 0; rsize -= ssize, raddr += ssize) {
2091                 if (raddr >= seg->s_base + seg->s_size) {
2092                         seg = AS_SEGNEXT(as, seg);
2093                         if (seg == NULL || raddr != seg->s_base) {
2094                                 error = -1;
2095                                 break;
2096                         }
2097                 }
```



```
2098                 if ((raddr + rsize) > (seg->s_base + seg->s_size))
2099                         ssize = seg->s_base + seg->s_size - raddr;
2100                 else
2101                         ssize = rsize;
2102                 *sizep += isize = segop_incore(seg, raddr, ssize, vec);
2252                 *sizep += isize = SEGOP_INCORE(seg, raddr, ssize, vec);
2103                 if (isize != ssize) {
2104                         error = -1;
2105                         break;
2106                 }
2107                 vec += btopr(ssize);
2108         }
2109         AS_LOCK_EXIT(as, &as->a_lock);
2110         return (error);
2111 }

2113 static void
2114 as_segunlock(struct seg *seg, caddr_t addr, int attr,
2115         ulong_t *bitmap, size_t position, size_t npages)
2116 {
2117         caddr_t range_start;
2118         size_t  pos1 = position;
2119         size_t  pos2;
2120         size_t  size;
2121         size_t  end_pos = npages + position;

2123         while (bt_range(bitmap, &pos1, &pos2, end_pos)) {
2124                 size = ptob((pos2 - pos1));
2125                 range_start = (caddr_t)((uintptr_t)addr +
2126                     ptob(pos1 - position));

2128                 (void) segop_lockop(seg, range_start, size, attr, MC_UNLOCK,
2278                 (void) SEGOP_LOCKOP(seg, range_start, size, attr, MC_UNLOCK,
2129                     (ulong_t *)NULL, (size_t)NULL);
2130                 pos1 = pos2;
2131         }
2132 }
______unchanged_portion_omitted_

2157 /*
2158  * Cache control operations over the interval [addr, addr + size) in
2159  * address space "as".
2160  */
2161 /*ARGSUSED*/
2162 int
2163 as_ctl(struct as *as, caddr_t addr, size_t size, int func, int attr,
2164     uintptr_t arg, ulong_t *lock_map, size_t pos)
2165 {
2166         struct seg *seg;        /* working segment */
2167         caddr_t raddr;          /* rounded down addr */
2168         caddr_t initraddr;      /* saved initial rounded down addr */
2169         size_t rsize;           /* rounded up size */
2170         size_t initrsize;       /* saved initial rounded up size */
2171         size_t ssize;           /* size of seg */
2172         int error = 0;                  /* result */
2173         size_t mlock_size;      /* size of bitmap */
2174         ulong_t *mlock_map;     /* pointer to bitmap used */
2175                                 /* to represent the locked */
2176                                 /* pages. */
2177 retry:
2178         if (error == IE_RETRY)
2179                 AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
2180         else
2181                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

2183         /*
```

```
2184          * If these are address space lock/unlock operations, loop over
2185          * all segments in the address space, as appropriate.
2186          */
2187         if (func == MC_LOCKAS) {
2188                 size_t npages, idx;
2189                 size_t rlen = 0;        /* rounded as length */
2191                 idx = pos;
2193                 if (arg & MCL_FUTURE) {
2194                         mutex_enter(&as->a_contents);
2195                         AS_SETPGLCK(as);
2196                         mutex_exit(&as->a_contents);
2197                 }
2198                 if ((arg & MCL_CURRENT) == 0) {
2199                         AS_LOCK_EXIT(as, &as->a_lock);
2200                         return (0);
2201                 }
2203                 seg = AS_SEGFIRST(as);
2204                 if (seg == NULL) {
2205                         AS_LOCK_EXIT(as, &as->a_lock);
2206                         return (0);
2207                 }
2209                 do {
2210                         raddr = (caddr_t)((uintptr_t)seg->s_base &
2211                             (uintptr_t)PAGEMASK);
2212                         rlen += (((uintptr_t)(seg->s_base + seg->s_size) +
2213                             PAGEOFFSET) & PAGEMASK) - (uintptr_t)raddr;
2214                 } while ((seg = AS_SEGNEXT(as, seg)) != NULL);
2216                 mlock_size = BT_BITOUL(btopr(rlen));
2217                 if ((mlock_map = (ulong_t *)kmem_zalloc(mlock_size *
2218                     sizeof (ulong_t), KM_NOSLEEP)) == NULL) {
2219                                 AS_LOCK_EXIT(as, &as->a_lock);
2220                                 return (EAGAIN);
2221                 }
2223                 for (seg = AS_SEGFIRST(as); seg; seg = AS_SEGNEXT(as, seg)) {
2224                         error = segop_lockop(seg, seg->s_base,
2374                         error = SEGOP_LOCKOP(seg, seg->s_base,
2225                             seg->s_size, attr, MC_LOCK, mlock_map, pos);
2226                         if (error != 0)
2227                                 break;
2228                         pos += seg_pages(seg);
2229                 }
2231                 if (error) {
2232                         for (seg = AS_SEGFIRST(as); seg != NULL;
2233                             seg = AS_SEGNEXT(as, seg)) {
2235                                 raddr = (caddr_t)((uintptr_t)seg->s_base &
2236                                     (uintptr_t)PAGEMASK);
2237                                 npages = seg_pages(seg);
2238                                 as_segunlock(seg, raddr, attr, mlock_map,
2239                                     idx, npages);
2240                                 idx += npages;
2241                         }
2242                 }
2244                 kmem_free(mlock_map, mlock_size * sizeof (ulong_t));
2245                 AS_LOCK_EXIT(as, &as->a_lock);
2246                 goto lockerr;
2247         } else if (func == MC_UNLOCKAS) {
2248                 mutex_enter(&as->a_contents);
```



```
2249                 AS_CLRPGLCK(as);
2250                 mutex_exit(&as->a_contents);
2252                 for (seg = AS_SEGFIRST(as); seg; seg = AS_SEGNEXT(as, seg)) {
2253                         error = segop_lockop(seg, seg->s_base,
2403                         error = SEGOP_LOCKOP(seg, seg->s_base,
2254                             seg->s_size, attr, MC_UNLOCK, NULL, 0);
2255                         if (error != 0)
2256                                 break;
2257                 }
2259                 AS_LOCK_EXIT(as, &as->a_lock);
2260                 goto lockerr;
2261         }
2263         /*
2264          * Normalize addresses and sizes.
2265          */
2266         initraddr = raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2267         initrsize = rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
2268             (size_t)raddr;
2270         if (raddr + rsize < raddr) {            /* check for wraparound */
2271                 AS_LOCK_EXIT(as, &as->a_lock);
2272                 return (ENOMEM);
2273         }
2275         /*
2276          * Get initial segment.
2277          */
2278         if ((seg = as_segat(as, raddr)) == NULL) {
2279                 AS_LOCK_EXIT(as, &as->a_lock);
2280                 return (ENOMEM);
2281         }
2283         if (func == MC_LOCK) {
2284                 mlock_size = BT_BITOUL(btopr(rsize));
2285                 if ((mlock_map = (ulong_t *)kmem_zalloc(mlock_size *
2286                     sizeof (ulong_t), KM_NOSLEEP)) == NULL) {
2287                                 AS_LOCK_EXIT(as, &as->a_lock);
2288                                 return (EAGAIN);
2289                 }
2290         }
2292         /*
2293          * Loop over all segments.  If a hole in the address range is
2294          * discovered, then fail.  For each segment, perform the appropriate
2295          * control operation.
2296          */
2297         while (rsize != 0) {
2299                 /*
2300                  * Make sure there's no hole, calculate the portion
2301                  * of the next segment to be operated over.
2302                  */
2303                 if (raddr >= seg->s_base + seg->s_size) {
2304                         seg = AS_SEGNEXT(as, seg);
2305                         if (seg == NULL || raddr != seg->s_base) {
2306                                 if (func == MC_LOCK) {
2307                                         as_unlockerr(as, attr, mlock_map,
2308                                             initraddr, initrsize - rsize);
2309                                         kmem_free(mlock_map,
2310                                             mlock_size * sizeof (ulong_t));
2311                                 }
2312                                 AS_LOCK_EXIT(as, &as->a_lock);
2313                                 return (ENOMEM);
```

```
2314                    }
2315            }
2316            if ((raddr + rsize) > (seg->s_base + seg->s_size))
2317                    ssize = seg->s_base + seg->s_size - raddr;
2318            else
2319                    ssize = rsize;

2321            /*
2322             * Dispatch on specific function.
2323             */
2324            switch (func) {

2326            /*
2327             * Synchronize cached data from mappings with backing
2328             * objects.
2329             */
2330            case MC_SYNC:
2331                    if (error = segop_sync(seg, raddr, ssize,
2481                    if (error = SEGOP_SYNC(seg, raddr, ssize,
2332                        attr, (uint_t)arg)) {
2333                            AS_LOCK_EXIT(as, &as->a_lock);
2334                            return (error);
2335                    }
2336                    break;

2338            /*
2339             * Lock pages in memory.
2340             */
2341            case MC_LOCK:
2342                    if (error = segop_lockop(seg, raddr, ssize,
2492                    if (error = SEGOP_LOCKOP(seg, raddr, ssize,
2343                        attr, func, mlock_map, pos)) {
2344                            as_unlockerr(as, attr, mlock_map, initraddr,
2345                                initrsize - rsize + ssize);
2346                            kmem_free(mlock_map, mlock_size *
2347                                sizeof (ulong_t));
2348                            AS_LOCK_EXIT(as, &as->a_lock);
2349                            goto lockerr;
2350                    }
2351                    break;

2353            /*
2354             * Unlock mapped pages.
2355             */
2356            case MC_UNLOCK:
2357                    (void) segop_lockop(seg, raddr, ssize, attr, func,
2507                    (void) SEGOP_LOCKOP(seg, raddr, ssize, attr, func,
2358                        (ulong_t *)NULL, (size_t)NULL);
2359                    break;

2361            /*
2362             * Store VM advise for mapped pages in segment layer.
2363             */
2364            case MC_ADVISE:
2365                    error = segop_advise(seg, raddr, ssize, (uint_t)arg);
2515                    error = SEGOP_ADVISE(seg, raddr, ssize, (uint_t)arg);

2367                    /*
2368                     * Check for regular errors and special retry error
2369                     */
2370                    if (error) {
2371                            if (error == IE_RETRY) {
2372                                    /*
2373                                     * Need to acquire writers lock, so
2374                                     * have to drop readers lock and start
2375                                     * all over again
```



```
2376                                     */
2377                                    AS_LOCK_EXIT(as, &as->a_lock);
2378                                    goto retry;
2379                            } else if (error == IE_REATTACH) {
2380                                    /*
2381                                     * Find segment for current address
2382                                     * because current segment just got
2383                                     * split or concatenated
2384                                     */
2385                                    seg = as_segat(as, raddr);
2386                                    if (seg == NULL) {
2387                                            AS_LOCK_EXIT(as, &as->a_lock);
2388                                            return (ENOMEM);
2389                                    }
2390                            } else {
2391                                    /*
2392                                     * Regular error
2393                                     */
2394                                    AS_LOCK_EXIT(as, &as->a_lock);
2395                                    return (error);
2396                            }
2397                    }
2398                    break;

2400            case MC_INHERIT_ZERO:
2401                    error = segop_inherit(seg, raddr, ssize, SEGP_INH_ZERO);
2551                    if (seg->s_ops->inherit == NULL) {
2552                            error = ENOTSUP;
2553                    } else {
2554                            error = SEGOP_INHERIT(seg, raddr, ssize,
2555                                SEGP_INH_ZERO);
2556                    }
2402                    if (error != 0) {
2403                            AS_LOCK_EXIT(as, &as->a_lock);
2404                            return (error);
2405                    }
2406                    break;

2408            /*
2409             * Can't happen.
2410             */
2411            default:
2412                    panic("as_ctl: bad operation %d", func);
2413                    /*NOTREACHED*/
2414            }

2416            rsize -= ssize;
2417            raddr += ssize;
2418    }

2420    if (func == MC_LOCK)
2421            kmem_free(mlock_map, mlock_size * sizeof (ulong_t));
2422    AS_LOCK_EXIT(as, &as->a_lock);
2423    return (0);
2424 lockerr:

2426    /*
2427     * If the lower levels returned EDEADLK for a segment lockop,
2428     * it means that we should retry the operation.  Let's wait
2429     * a bit also to let the deadlock causing condition clear.
2430     * This is part of a gross hack to work around a design flaw
2431     * in the ufs/sds logging code and should go away when the
2432     * logging code is re-designed to fix the problem. See bug
2433     * 4125102 for details of the problem.
2434     */
2435    if (error == EDEADLK) {
```

```
2436                 delay(deadlk_wait);
2437                 error = 0;
2438                 goto retry;
2439         }
2440         return (error);
2441 }
_____unchanged_portion_omitted_

2462 /*
2463  * Pagelock pages from a range that spans more than 1 segment.  Obtain shadow
2464  * lists from each segment and copy them to one contiguous shadow list (plist)
2465  * as expected by the caller.  Save pointers to per segment shadow lists at
2466  * the tail of plist so that they can be used during as_pageunlock().
2467  */
2468 static int
2469 as_pagelock_segs(struct as *as, struct seg *seg, struct page ***ppp,
2470     caddr_t addr, size_t size, enum seg_rw rw)
2471 {
2472         caddr_t sv_addr = addr;
2473         size_t sv_size = size;
2474         struct seg *sv_seg = seg;
2475         ulong_t segcnt = 1;
2476         ulong_t cnt;
2477         size_t ssize;
2478         pgcnt_t npages = btop(size);
2479         page_t **plist;
2480         page_t **pl;
2481         int error;
2482         caddr_t eaddr;
2483         faultcode_t fault_err = 0;
2484         pgcnt_t pl_off;
2485         extern const struct seg_ops segspt_shmops;
2640         extern struct seg_ops segspt_shmops;

2487         ASSERT(AS_LOCK_HELD(as, &as->a_lock));
2488         ASSERT(seg != NULL);
2489         ASSERT(addr >= seg->s_base && addr < seg->s_base + seg->s_size);
2490         ASSERT(addr + size > seg->s_base + seg->s_size);
2491         ASSERT(IS_P2ALIGNED(size, PAGESIZE));
2492         ASSERT(IS_P2ALIGNED(addr, PAGESIZE));

2494         /*
2495          * Count the number of segments covered by the range we are about to
2496          * lock. The segment count is used to size the shadow list we return
2497          * back to the caller.
2498          */
2499         for (; size != 0; size -= ssize, addr += ssize) {
2500                 if (addr >= seg->s_base + seg->s_size) {

2502                         seg = AS_SEGNEXT(as, seg);
2503                         if (seg == NULL || addr != seg->s_base) {
2504                                 AS_LOCK_EXIT(as, &as->a_lock);
2505                                 return (EFAULT);
2506                         }
2507                         /*
2508                          * Do a quick check if subsequent segments
2509                          * will most likely support pagelock.
2510                          */
2511                         if (seg->s_ops == &segvn_ops) {
2512                                 vnode_t *vp;

2514                                 if (segop_getvp(seg, addr, &vp) != 0 ||
2669                                 if (SEGOP_GETVP(seg, addr, &vp) != 0 ||
2515                                     vp != NULL) {
2516                                         AS_LOCK_EXIT(as, &as->a_lock);
2517                                         goto slow;
```



```
2518                                 }
2519                         } else if (seg->s_ops != &segspt_shmops) {
2520                                 AS_LOCK_EXIT(as, &as->a_lock);
2521                                 goto slow;
2522                         }
2523                         segcnt++;
2524                 }
2525                 if (addr + size > seg->s_base + seg->s_size) {
2526                         ssize = seg->s_base + seg->s_size - addr;
2527                 } else {
2528                         ssize = size;
2529                 }
2530         }
2531         ASSERT(segcnt > 1);

2533         plist = kmem_zalloc((npages + segcnt) * sizeof (page_t *), KM_SLEEP);

2535         addr = sv_addr;
2536         size = sv_size;
2537         seg = sv_seg;

2539         for (cnt = 0, pl_off = 0; size != 0; size -= ssize, addr += ssize) {
2540                 if (addr >= seg->s_base + seg->s_size) {
2541                         seg = AS_SEGNEXT(as, seg);
2542                         ASSERT(seg != NULL && addr == seg->s_base);
2543                         cnt++;
2544                         ASSERT(cnt < segcnt);
2545                 }
2546                 if (addr + size > seg->s_base + seg->s_size) {
2547                         ssize = seg->s_base + seg->s_size - addr;
2548                 } else {
2549                         ssize = size;
2550                 }
2551                 pl = &plist[npages + cnt];
2552                 error = segop_pagelock(seg, addr, ssize, (page_t ***)pl,
2707                 error = SEGOP_PAGELOCK(seg, addr, ssize, (page_t ***)pl,
2553                     L_PAGELOCK, rw);
2554                 if (error) {
2555                         break;
2556                 }
2557                 ASSERT(plist[npages + cnt] != NULL);
2558                 ASSERT(pl_off + btop(ssize) <= npages);
2559                 bcopy(plist[npages + cnt], &plist[pl_off],
2560                     btop(ssize) * sizeof (page_t *));
2561                 pl_off += btop(ssize);
2562         }

2564         if (size == 0) {
2565                 AS_LOCK_EXIT(as, &as->a_lock);
2566                 ASSERT(cnt == segcnt - 1);
2567                 *ppp = plist;
2568                 return (0);
2569         }

2571         /*
2572          * one of pagelock calls failed. The error type is in error variable.
2573          * Unlock what we've locked so far and retry with F_SOFTLOCK if error
2574          * type is either EFAULT or ENOTSUP. Otherwise just return the error
2575          * back to the caller.
2576          */

2578         eaddr = addr;
2579         seg = sv_seg;

2581         for (cnt = 0, addr = sv_addr; addr < eaddr; addr += ssize) {
2582                 if (addr >= seg->s_base + seg->s_size) {
```

```
2583                        seg = AS_SEGNEXT(as, seg);
2584                        ASSERT(seg != NULL && addr == seg->s_base);
2585                        cnt++;
2586                        ASSERT(cnt < segcnt);
2587                }
2588                if (eaddr > seg->s_base + seg->s_size) {
2589                        ssize = seg->s_base + seg->s_size - addr;
2590                } else {
2591                        ssize = eaddr - addr;
2592                }
2593                pl = &plist[npages + cnt];
2594                ASSERT(*pl != NULL);
2595                (void) segop_pagelock(seg, addr, ssize, (page_t ***)pl,
2750                (void) SEGOP_PAGELOCK(seg, addr, ssize, (page_t ***)pl,
2596                    L_PAGEUNLOCK, rw);
2597        }

2599        AS_LOCK_EXIT(as, &as->a_lock);

2601        kmem_free(plist, (npages + segcnt) * sizeof (page_t *));

2603        if (error != ENOTSUP && error != EFAULT) {
2604                return (error);
2605        }

2607 slow:
2608        /*
2609         * If we are here because pagelock failed due to the need to cow fault
2610         * in the pages we want to lock F_SOFTLOCK will do this job and in
2611         * next as_pagelock() call for this address range pagelock will
2612         * hopefully succeed.
2613         */
2614        fault_err = as_fault(as->a_hat, as, sv_addr, sv_size, F_SOFTLOCK, rw);
2615        if (fault_err != 0) {
2616                return (fc_decode(fault_err));
2617        }
2618        *ppp = NULL;

2620        return (0);
2621 }

2623 /*
2624  * lock pages in a given address space. Return shadow list. If
2625  * the list is NULL, the MMU mapping is also locked.
2626  */
2627 int
2628 as_pagelock(struct as *as, struct page ***ppp, caddr_t addr,
2629     size_t size, enum seg_rw rw)
2630 {
2631        size_t rsize;
2632        caddr_t raddr;
2633        faultcode_t fault_err;
2634        struct seg *seg;
2635        int err;

2637        TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_AS_LOCK_START,
2638            "as_pagelock_start: addr %p size %ld", addr, size);

2640        raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2641        rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
2642            (size_t)raddr;

2644        /*
2645         * if the request crosses two segments let
2646         * as_fault handle it.
2647         */
```



```
2648        AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

2650        seg = as_segat(as, raddr);
2651        if (seg == NULL) {
2652                AS_LOCK_EXIT(as, &as->a_lock);
2653                return (EFAULT);
2654        }
2655        ASSERT(raddr >= seg->s_base && raddr < seg->s_base + seg->s_size);
2656        if (raddr + rsize > seg->s_base + seg->s_size) {
2657                return (as_pagelock_segs(as, seg, ppp, raddr, rsize, rw));
2658        }
2659        if (raddr + rsize <= raddr) {
2660                AS_LOCK_EXIT(as, &as->a_lock);
2661                return (EFAULT);
2662        }

2664        TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEG_LOCK_START,
2665            "seg_lock_1_start: raddr %p rsize %ld", raddr, rsize);

2667        /*
2668         * try to lock pages and pass back shadow list
2669         */
2670        err = segop_pagelock(seg, raddr, rsize, ppp, L_PAGELOCK, rw);
2825        err = SEGOP_PAGELOCK(seg, raddr, rsize, ppp, L_PAGELOCK, rw);

2672        TRACE_0(TR_FAC_PHYSIO, TR_PHYSIO_SEG_LOCK_END, "seg_lock_1_end");

2674        AS_LOCK_EXIT(as, &as->a_lock);

2676        if (err == 0 || (err != ENOTSUP && err != EFAULT)) {
2677                return (err);
2678        }

2680        /*
2681         * Use F_SOFTLOCK to lock the pages because pagelock failed either due
2682         * to no pagelock support for this segment or pages need to be cow
2683         * faulted in. If fault is needed F_SOFTLOCK will do this job for
2684         * this as_pagelock() call and in the next as_pagelock() call for the
2685         * same address range pagelock call will hopefull succeed.
2686         */
2687        fault_err = as_fault(as->a_hat, as, addr, size, F_SOFTLOCK, rw);
2688        if (fault_err != 0) {
2689                return (fc_decode(fault_err));
2690        }
2691        *ppp = NULL;

2693        TRACE_0(TR_FAC_PHYSIO, TR_PHYSIO_AS_LOCK_END, "as_pagelock_end");
2694        return (0);
2695 }

2697 /*
2698  * unlock pages locked by as_pagelock_segs().  Retrieve per segment shadow
2699  * lists from the end of plist and call pageunlock interface for each segment.
2700  * Drop as lock and free plist.
2701  */
2702 static void
2703 as_pageunlock_segs(struct as *as, struct seg *seg, caddr_t addr, size_t size,
2704     struct page **plist, enum seg_rw rw)
2705 {
2706        ulong_t cnt;
2707        caddr_t eaddr = addr + size;
2708        pgcnt_t npages = btop(size);
2709        size_t ssize;
2710        page_t **pl;

2712        ASSERT(AS_LOCK_HELD(as, &as->a_lock));
```

```
2713         ASSERT(seg != NULL);
2714         ASSERT(addr >= seg->s_base && addr < seg->s_base + seg->s_size);
2715         ASSERT(addr + size > seg->s_base + seg->s_size);
2716         ASSERT(IS_P2ALIGNED(size, PAGESIZE));
2717         ASSERT(IS_P2ALIGNED(addr, PAGESIZE));
2718         ASSERT(plist != NULL);

2720         for (cnt = 0; addr < eaddr; addr += ssize) {
2721                 if (addr >= seg->s_base + seg->s_size) {
2722                         seg = AS_SEGNEXT(as, seg);
2723                         ASSERT(seg != NULL && addr == seg->s_base);
2724                         cnt++;
2725                 }
2726                 if (eaddr > seg->s_base + seg->s_size) {
2727                         ssize = seg->s_base + seg->s_size - addr;
2728                 } else {
2729                         ssize = eaddr - addr;
2730                 }
2731                 pl = &plist[npages + cnt];
2732                 ASSERT(*pl != NULL);
2733                 (void) segop_pagelock(seg, addr, ssize, (page_t ***)pl,
2888                 (void) SEGOP_PAGELOCK(seg, addr, ssize, (page_t ***)pl,
2734                     L_PAGEUNLOCK, rw);
2735         }
2736         ASSERT(cnt > 0);
2737         AS_LOCK_EXIT(as, &as->a_lock);

2739         cnt++;
2740         kmem_free(plist, (npages + cnt) * sizeof (page_t *));
2741 }

2743 /*
2744  * unlock pages in a given address range
2745  */
2746 void
2747 as_pageunlock(struct as *as, struct page **pp, caddr_t addr, size_t size,
2748     enum seg_rw rw)
2749 {
2750         struct seg *seg;
2751         size_t rsize;
2752         caddr_t raddr;

2754         TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_AS_UNLOCK_START,
2755             "as_pageunlock_start: addr %p size %ld", addr, size);

2757         /*
2758          * if the shadow list is NULL, as_pagelock was
2759          * falling back to as_fault
2760          */
2761         if (pp == NULL) {
2762                 (void) as_fault(as->a_hat, as, addr, size, F_SOFTUNLOCK, rw);
2763                 return;
2764         }

2766         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2767         rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
2768             (size_t)raddr;

2770         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
2771         seg = as_segat(as, raddr);
2772         ASSERT(seg != NULL);

2774         TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEG_UNLOCK_START,
2775             "seg_unlock_start: raddr %p rsize %ld", raddr, rsize);

2777         ASSERT(raddr >= seg->s_base && raddr < seg->s_base + seg->s_size);
```



```
2778         if (raddr + rsize <= seg->s_base + seg->s_size) {
2779                 segop_pagelock(seg, raddr, rsize, &pp, L_PAGEUNLOCK, rw);
2934                 SEGOP_PAGELOCK(seg, raddr, rsize, &pp, L_PAGEUNLOCK, rw);
2780         } else {
2781                 as_pageunlock_segs(as, seg, raddr, rsize, pp, rw);
2782                 return;
2783         }
2784         AS_LOCK_EXIT(as, &as->a_lock);
2785         TRACE_0(TR_FAC_PHYSIO, TR_PHYSIO_AS_UNLOCK_END, "as_pageunlock_end");
2786 }

2788 int
2789 as_setpagesize(struct as *as, caddr_t addr, size_t size, uint_t szc,
2790     boolean_t wait)
2791 {
2792         struct seg *seg;
2793         size_t ssize;
2794         caddr_t raddr;                  /* rounded down addr */
2795         size_t rsize;                   /* rounded up size */
2796         int error = 0;
2797         size_t pgsz = page_get_pagesize(szc);

2799 setpgsz_top:
2800         if (!IS_P2ALIGNED(addr, pgsz) || !IS_P2ALIGNED(size, pgsz)) {
2801                 return (EINVAL);
2802         }

2804         raddr = addr;
2805         rsize = size;

2807         if (raddr + rsize < raddr)              /* check for wraparound */
2808                 return (ENOMEM);

2810         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
2811         as_clearwatchprot(as, raddr, rsize);
2812         seg = as_segat(as, raddr);
2813         if (seg == NULL) {
2814                 as_setwatch(as);
2815                 AS_LOCK_EXIT(as, &as->a_lock);
2816                 return (ENOMEM);
2817         }

2819         for (; rsize != 0; rsize -= ssize, raddr += ssize) {
2820                 if (raddr >= seg->s_base + seg->s_size) {
2821                         seg = AS_SEGNEXT(as, seg);
2822                         if (seg == NULL || raddr != seg->s_base) {
2823                                 error = ENOMEM;
2824                                 break;
2825                         }
2826                 }
2827                 if ((raddr + rsize) > (seg->s_base + seg->s_size)) {
2828                         ssize = seg->s_base + seg->s_size - raddr;
2829                 } else {
2830                         ssize = rsize;
2831                 }

2833 retry:
2834                 error = segop_setpagesize(seg, raddr, ssize, szc);
2989                 error = SEGOP_SETPAGESIZE(seg, raddr, ssize, szc);

2836                 if (error == IE_NOMEM) {
2837                         error = EAGAIN;
2838                         break;
2839                 }

2841                 if (error == IE_RETRY) {
```

```
2842                        AS_LOCK_EXIT(as, &as->a_lock);
2843                        goto setpgsz_top;
2844                }

2846                if (error == ENOTSUP) {
2847                        error = EINVAL;
2848                        break;
2849                }

2851                if (wait && (error == EAGAIN)) {
2852                        /*
2853                         * Memory is currently locked.  It must be unlocked
2854                         * before this operation can succeed through a retry.
2855                         * The possible reasons for locked memory and
2856                         * corresponding strategies for unlocking are:
2857                         * (1) Normal I/O
2858                         *      wait for a signal that the I/O operation
2859                         *      has completed and the memory is unlocked.
2860                         * (2) Asynchronous I/O
2861                         *      The aio subsystem does not unlock pages when
2862                         *      the I/O is completed. Those pages are unlocked
2863                         *      when the application calls aiowait/aioerror.
2864                         *      So, to prevent blocking forever, cv_broadcast()
2865                         *      is done to wake up aio_cleanup_thread.
2866                         *      Subsequently, segvn_reclaim will be called, and
2867                         *      that will do AS_CLRUNMAPWAIT() and wake us up.
2868                         * (3) Long term page locking:
2869                         *      This is not relevant for as_setpagesize()
2870                         *      because we cannot change the page size for
2871                         *      driver memory. The attempt to do so will
2872                         *      fail with a different error than EAGAIN so
2873                         *      there's no need to trigger as callbacks like
2874                         *      as_unmap, as_setprot or as_free would do.
2875                         */
2876                        mutex_enter(&as->a_contents);
2877                        if (!AS_ISNOUNMAPWAIT(as)) {
2878                                if (AS_ISUNMAPWAIT(as) == 0) {
2879                                        cv_broadcast(&as->a_cv);
2880                                }
2881                                AS_SETUNMAPWAIT(as);
2882                                AS_LOCK_EXIT(as, &as->a_lock);
2883                                while (AS_ISUNMAPWAIT(as)) {
2884                                        cv_wait(&as->a_cv, &as->a_contents);
2885                                }
2886                        } else {
2887                                /*
2888                                 * We may have raced with
2889                                 * segvn_reclaim()/segspt_reclaim(). In this
2890                                 * case clean nounmapwait flag and retry since
2891                                 * softlockcnt in this segment may be already
2892                                 * 0.  We don't drop as writer lock so our
2893                                 * number of retries without sleeping should
2894                                 * be very small. See segvn_reclaim() for
2895                                 * more comments.
2896                                 */
2897                                AS_CLRNOUNMAPWAIT(as);
2898                                mutex_exit(&as->a_contents);
2899                                goto retry;
2900                        }
2901                        mutex_exit(&as->a_contents);
2902                        goto setpgsz_top;
2903                } else if (error != 0) {
2904                        break;
2905                }
2906        }
2907        as_setwatch(as);
```



```
2908        AS_LOCK_EXIT(as, &as->a_lock);
2909        return (error);
2910 }

2912 /*
2913  * as_iset3_default_lpsize() just calls segop_setpagesize() on all segments
3068  * as_iset3_default_lpsize() just calls SEGOP_SETPAGESIZE() on all segments
2914  * in its chunk where s_szc is less than the szc we want to set.
2915  */
2916 static int
2917 as_iset3_default_lpsize(struct as *as, caddr_t raddr, size_t rsize, uint_t szc,
2918     int *retry)
2919 {
2920        struct seg *seg;
2921        size_t ssize;
2922        int error;

2924        ASSERT(AS_WRITE_HELD(as, &as->a_lock));

2926        seg = as_segat(as, raddr);
2927        if (seg == NULL) {
2928                panic("as_iset3_default_lpsize: no seg");
2929        }

2931        for (; rsize != 0; rsize -= ssize, raddr += ssize) {
2932                if (raddr >= seg->s_base + seg->s_size) {
2933                        seg = AS_SEGNEXT(as, seg);
2934                        if (seg == NULL || raddr != seg->s_base) {
2935                                panic("as_iset3_default_lpsize: as changed");
2936                        }
2937                }
2938                if ((raddr + rsize) > (seg->s_base + seg->s_size)) {
2939                        ssize = seg->s_base + seg->s_size - raddr;
2940                } else {
2941                        ssize = rsize;
2942                }

2944                if (szc > seg->s_szc) {
2945                        error = segop_setpagesize(seg, raddr, ssize, szc);
3100                        error = SEGOP_SETPAGESIZE(seg, raddr, ssize, szc);
2946                        /* Only retry on EINVAL segments that have no vnode. */
2947                        if (error == EINVAL) {
2948                                vnode_t *vp = NULL;
2949                                if ((segop_gettype(seg, raddr) & MAP_SHARED) &&
2950                                    (segop_getvp(seg, raddr, &vp) != 0 ||
3104                                if ((SEGOP_GETTYPE(seg, raddr) & MAP_SHARED) &&
3105                                    (SEGOP_GETVP(seg, raddr, &vp) != 0 ||
2951                                    vp == NULL)) {
2952                                        *retry = 1;
2953                                } else {
2954                                        *retry = 0;
2955                                }
2956                        }
2957                        if (error) {
2958                                return (error);
2959                        }
2960                }
2961        }
2962        return (0);
2963 }
```

_____*unchanged_portion_omitted_*

```
3150 /*
3151  * Set the default large page size for the range. Called via memcntl with
3152  * page size set to 0. as_set_default_lpsize breaks the range down into
3153  * chunks with the same type/flags, ignores-non segvn segments, and passes
```

```
3154  * each chunk to as_iset_default_lpsize().
3155  */
3156 int
3157 as_set_default_lpsize(struct as *as, caddr_t addr, size_t size)
3158 {
3159         struct seg *seg;
3160         caddr_t raddr;
3161         size_t rsize;
3162         size_t ssize;
3163         int rtype, rflags;
3164         int stype, sflags;
3165         int error;
3166         caddr_t setaddr;
3167         size_t setsize;
3168         int segvn;

3170         if (size == 0)
3171                 return (0);

3173         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
3174 again:
3175         error = 0;

3177         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
3178         rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
3179             (size_t)raddr;

3181         if (raddr + rsize < raddr) {            /* check for wraparound */
3182                 AS_LOCK_EXIT(as, &as->a_lock);
3183                 return (ENOMEM);
3184         }
3185         as_clearwatchprot(as, raddr, rsize);
3186         seg = as_segat(as, raddr);
3187         if (seg == NULL) {
3188                 as_setwatch(as);
3189                 AS_LOCK_EXIT(as, &as->a_lock);
3190                 return (ENOMEM);
3191         }
3192         if (seg->s_ops == &segvn_ops) {
3193                 rtype = segop_gettype(seg, addr);
3348                 rtype = SEGOP_GETTYPE(seg, addr);
3194                 rflags = rtype & (MAP_TEXT | MAP_INITDATA);
3195                 rtype = rtype & (MAP_SHARED | MAP_PRIVATE);
3196                 segvn = 1;
3197         } else {
3198                 segvn = 0;
3199         }
3200         setaddr = raddr;
3201         setsize = 0;

3203         for (; rsize != 0; rsize -= ssize, raddr += ssize, setsize += ssize) {
3204                 if (raddr >= (seg->s_base + seg->s_size)) {
3205                         seg = AS_SEGNEXT(as, seg);
3206                         if (seg == NULL || raddr != seg->s_base) {
3207                                 error = ENOMEM;
3208                                 break;
3209                         }
3210                         if (seg->s_ops == &segvn_ops) {
3211                                 stype = segop_gettype(seg, raddr);
3366                                 stype = SEGOP_GETTYPE(seg, raddr);
3212                                 sflags = stype & (MAP_TEXT | MAP_INITDATA);
3213                                 stype &= (MAP_SHARED | MAP_PRIVATE);
3214                                 if (segvn && (rflags != sflags ||
3215                                     rtype != stype)) {
3216                                         /*
3217                                          * The next segment is also segvn but
```



```
3218                                          * has different flags and/or type.
3219                                          */
3220                                         ASSERT(setsize != 0);
3221                                         error = as_iset_default_lpsize(as,
3222                                             setaddr, setsize, rflags, rtype);
3223                                         if (error) {
3224                                                 break;
3225                                         }
3226                                         rflags = sflags;
3227                                         rtype = stype;
3228                                         setaddr = raddr;
3229                                         setsize = 0;
3230                                 } else if (!segvn) {
3231                                         rflags = sflags;
3232                                         rtype = stype;
3233                                         setaddr = raddr;
3234                                         setsize = 0;
3235                                         segvn = 1;
3236                                 }
3237                         } else if (segvn) {
3238                                 /* The next segment is not segvn. */
3239                                 ASSERT(setsize != 0);
3240                                 error = as_iset_default_lpsize(as,
3241                                     setaddr, setsize, rflags, rtype);
3242                                 if (error) {
3243                                         break;
3244                                 }
3245                                 segvn = 0;
3246                         }
3247                 }
3248                 if ((raddr + rsize) > (seg->s_base + seg->s_size)) {
3249                         ssize = seg->s_base + seg->s_size - raddr;
3250                 } else {
3251                         ssize = rsize;
3252                 }
3253         }
3254         if (error == 0 && segvn) {
3255                 /* The last chunk when rsize == 0. */
3256                 ASSERT(setsize != 0);
3257                 error = as_iset_default_lpsize(as, setaddr, setsize,
3258                     rflags, rtype);
3259         }

3261         if (error == IE_RETRY) {
3262                 goto again;
3263         } else if (error == IE_NOMEM) {
3264                 error = EAGAIN;
3265         } else if (error == ENOTSUP) {
3266                 error = EINVAL;
3267         } else if (error == EAGAIN) {
3268                 mutex_enter(&as->a_contents);
3269                 if (!AS_ISNOUNMAPWAIT(as)) {
3270                         if (AS_ISUNMAPWAIT(as) == 0) {
3271                                 cv_broadcast(&as->a_cv);
3272                         }
3273                         AS_SETUNMAPWAIT(as);
3274                         AS_LOCK_EXIT(as, &as->a_lock);
3275                         while (AS_ISUNMAPWAIT(as)) {
3276                                 cv_wait(&as->a_cv, &as->a_contents);
3277                         }
3278                         mutex_exit(&as->a_contents);
3279                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
3280                 } else {
3281                         /*
3282                          * We may have raced with
3283                          * segvn_reclaim()/segspt_reclaim(). In this case
```

```
3284                         * clean nounmapwait flag and retry since softlockcnt
3285                         * in this segment may be already 0.  We don't drop as
3286                         * writer lock so our number of retries without
3287                         * sleeping should be very small. See segvn_reclaim()
3288                         * for more comments.
3289                         */
3290                        AS_CLRNOUNMAPWAIT(as);
3291                        mutex_exit(&as->a_contents);
3292                }
3293                goto again;
3294        }

3296        as_setwatch(as);
3297        AS_LOCK_EXIT(as, &as->a_lock);
3298        return (error);
3299 }

3301 /*
3302  * Setup all of the uninitialized watched pages that we can.
3303  */
3304 void
3305 as_setwatch(struct as *as)
3306 {
3307        struct watched_page *pwp;
3308        struct seg *seg;
3309        caddr_t vaddr;
3310        uint_t prot;
3311        int  err, retrycnt;

3313        if (avl_numnodes(&as->a_wpage) == 0)
3314                return;

3316        ASSERT(AS_WRITE_HELD(as, &as->a_lock));

3318        for (pwp = avl_first(&as->a_wpage); pwp != NULL;
3319            pwp = AVL_NEXT(&as->a_wpage, pwp)) {
3320                retrycnt = 0;
3321        retry:
3322                vaddr = pwp->wp_vaddr;
3323                if (pwp->wp_oprot != 0 ||       /* already set up */
3324                    (seg = as_segat(as, vaddr)) == NULL ||
3325                    segop_getprot(seg, vaddr, 0, &prot) != 0)
3480                    SEGOP_GETPROT(seg, vaddr, 0, &prot) != 0)
3326                        continue;

3328                pwp->wp_oprot = prot;
3329                if (pwp->wp_read)
3330                        prot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3331                if (pwp->wp_write)
3332                        prot &= ~PROT_WRITE;
3333                if (pwp->wp_exec)
3334                        prot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3335                if (!(pwp->wp_flags & WP_NOWATCH) && prot != pwp->wp_oprot) {
3336                        err = segop_setprot(seg, vaddr, PAGESIZE, prot);
3491                        err = SEGOP_SETPROT(seg, vaddr, PAGESIZE, prot);
3337                        if (err == IE_RETRY) {
3338                                pwp->wp_oprot = 0;
3339                                ASSERT(retrycnt == 0);
3340                                retrycnt++;
3341                                goto retry;
3342                        }
3343                }
3344                pwp->wp_prot = prot;
3345        }
3346 }
```



```
3348 /*
3349  * Clear all of the watched pages in the address space.
3350  */
3351 void
3352 as_clearwatch(struct as *as)
3353 {
3354        struct watched_page *pwp;
3355        struct seg *seg;
3356        caddr_t vaddr;
3357        uint_t prot;
3358        int err, retrycnt;

3360        if (avl_numnodes(&as->a_wpage) == 0)
3361                return;

3363        ASSERT(AS_WRITE_HELD(as, &as->a_lock));

3365        for (pwp = avl_first(&as->a_wpage); pwp != NULL;
3366            pwp = AVL_NEXT(&as->a_wpage, pwp)) {
3367                retrycnt = 0;
3368        retry:
3369                vaddr = pwp->wp_vaddr;
3370                if (pwp->wp_oprot == 0 ||       /* not set up */
3371                    (seg = as_segat(as, vaddr)) == NULL)
3372                        continue;

3374                if ((prot = pwp->wp_oprot) != pwp->wp_prot) {
3375                        err = segop_setprot(seg, vaddr, PAGESIZE, prot);
3530                        err = SEGOP_SETPROT(seg, vaddr, PAGESIZE, prot);
3376                        if (err == IE_RETRY) {
3377                                ASSERT(retrycnt == 0);
3378                                retrycnt++;
3379                                goto retry;
3380                        }
3381                }
3382                pwp->wp_oprot = 0;
3383                pwp->wp_prot = 0;
3384        }
3385 }

3387 /*
3388  * Force a new setup for all the watched pages in the range.
3389  */
3390 static void
3391 as_setwatchprot(struct as *as, caddr_t addr, size_t size, uint_t prot)
3392 {
3393        struct watched_page *pwp;
3394        struct watched_page tpw;
3395        caddr_t eaddr = addr + size;
3396        caddr_t vaddr;
3397        struct seg *seg;
3398        int err, retrycnt;
3399        uint_t  wprot;
3400        avl_index_t where;

3402        if (avl_numnodes(&as->a_wpage) == 0)
3403                return;

3405        ASSERT(AS_WRITE_HELD(as, &as->a_lock));

3407        tpw.wp_vaddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
3408        if ((pwp = avl_find(&as->a_wpage, &tpw, &where)) == NULL)
3409                pwp = avl_nearest(&as->a_wpage, where, AVL_AFTER);

3411        while (pwp != NULL && pwp->wp_vaddr < eaddr) {
3412                retrycnt = 0;
```

```
3413                 vaddr = pwp->wp_vaddr;

3415                 wprot = prot;
3416                 if (pwp->wp_read)
3417                         wprot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3418                 if (pwp->wp_write)
3419                         wprot &= ~PROT_WRITE;
3420                 if (pwp->wp_exec)
3421                         wprot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3422                 if (!(pwp->wp_flags & WP_NOWATCH) && wprot != pwp->wp_oprot) {
3423                 retry:
3424                         seg = as_segat(as, vaddr);
3425                         if (seg == NULL) {
3426                                 panic("as_setwatchprot: no seg");
3427                                 /*NOTREACHED*/
3428                         }
3429                         err = segop_setprot(seg, vaddr, PAGESIZE, wprot);
3584                         err = SEGOP_SETPROT(seg, vaddr, PAGESIZE, wprot);
3430                         if (err == IE_RETRY) {
3431                                 ASSERT(retrycnt == 0);
3432                                 retrycnt++;
3433                                 goto retry;
3434                         }
3435                 }
3436                 pwp->wp_oprot = prot;
3437                 pwp->wp_prot = wprot;

3439                 pwp = AVL_NEXT(&as->a_wpage, pwp);
3440         }
3441 }

3443 /*
3444  * Clear all of the watched pages in the range.
3445  */
3446 static void
3447 as_clearwatchprot(struct as *as, caddr_t addr, size_t size)
3448 {
3449         caddr_t eaddr = addr + size;
3450         struct watched_page *pwp;
3451         struct watched_page tpw;
3452         uint_t prot;
3453         struct seg *seg;
3454         int err, retrycnt;
3455         avl_index_t where;

3457         if (avl_numnodes(&as->a_wpage) == 0)
3458                 return;

3460         tpw.wp_vaddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
3461         if ((pwp = avl_find(&as->a_wpage, &tpw, &where)) == NULL)
3462                 pwp = avl_nearest(&as->a_wpage, where, AVL_AFTER);

3464         ASSERT(AS_WRITE_HELD(as, &as->a_lock));

3466         while (pwp != NULL && pwp->wp_vaddr < eaddr) {

3468                 if ((prot = pwp->wp_oprot) != 0) {
3469                         retrycnt = 0;

3471                         if (prot != pwp->wp_prot) {
3472                         retry:
3473                                 seg = as_segat(as, pwp->wp_vaddr);
3474                                 if (seg == NULL)
3475                                         continue;
3476                                 err = segop_setprot(seg, pwp->wp_vaddr,
3631                                 err = SEGOP_SETPROT(seg, pwp->wp_vaddr,
```



```
3477                                     PAGESIZE, prot);
3478                                 if (err == IE_RETRY) {
3479                                         ASSERT(retrycnt == 0);
3480                                         retrycnt++;
3481                                         goto retry;

3483                                 }
3484                         }
3485                         pwp->wp_oprot = 0;
3486                         pwp->wp_prot = 0;
3487                 }

3489                 pwp = AVL_NEXT(&as->a_wpage, pwp);
3490         }
3491 }
______unchanged_portion_omitted_

3510 /*
3511  * return memory object ID
3512  */
3513 int
3514 as_getmemid(struct as *as, caddr_t addr, memid_t *memidp)
3515 {
3516         struct seg      *seg;
3517         int             sts;

3519         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
3520         seg = as_segat(as, addr);
3521         if (seg == NULL) {
3522                 AS_LOCK_EXIT(as, &as->a_lock);
3523                 return (EFAULT);
3524         }
3680         /*
3681          * catch old drivers which may not support getmemid
3682          */
3683         if (seg->s_ops->getmemid == NULL) {
3684                 AS_LOCK_EXIT(as, &as->a_lock);
3685                 return (ENODEV);
3686         }

3526         sts = segop_getmemid(seg, addr, memidp);
3688         sts = SEGOP_GETMEMID(seg, addr, memidp);

3528         AS_LOCK_EXIT(as, &as->a_lock);
3529         return (sts);
3530 }
______unchanged_portion_omitted_
```

```
**********************************************************
   31976 Fri May  8 18:10:38 2015
new/usr/src/uts/common/vm/vm_pvn.c
patch lower-case-segops
**********************************************************
```

```
/*
 * 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) 1986, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */

/*
 * VM - paged vnode.
 *
 * This file supplies vm support for the vnode operations that deal with pages.
 */
#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/systm.h>
#include <sys/time.h>
#include <sys/buf.h>
#include <sys/vnode.h>
#include <sys/uio.h>
#include <sys/vmsystm.h>
#include <sys/mman.h>
#include <sys/vfs.h>
#include <sys/cred.h>
#include <sys/user.h>
#include <sys/kmem.h>
#include <sys/cmn_err.h>
#include <sys/debug.h>
#include <sys/cpuvar.h>
```



```
#include <sys/vtrace.h>
#include <sys/tnf_probe.h>

#include <vm/hat.h>
#include <vm/as.h>
#include <vm/seg.h>
#include <vm/rm.h>
#include <vm/pvn.h>
#include <vm/page.h>
#include <vm/seg_map.h>
#include <vm/seg_kmem.h>
#include <sys/fs/swapnode.h>

int pvn_nofodklust = 0;
int pvn_write_noklust = 0;

uint_t pvn_vmodsort_supported = 0;      /* set if HAT supports VMODSORT */
uint_t pvn_vmodsort_disable = 0;        /* set in /etc/system to disable HAT */
                                        /* support for vmodsort for testing */

static struct kmem_cache *marker_cache = NULL;

/*
 * Find the largest contiguous block which contains ‘addr’ for file offset
 * ‘offset’ in it while living within the file system block sizes (‘vp_off’
 * and ‘vp_len’) and the address space limits for which no pages currently
 * exist and which map to consecutive file offsets.
 */
page_t *
pvn_read_kluster(
        struct vnode *vp,
        u_offset_t off,
        struct seg *seg,
        caddr_t addr,
        u_offset_t *offp,                       /* return values */
        size_t *lenp,                           /* return values */
        u_offset_t vp_off,
        size_t vp_len,
        int isra)
{
        ssize_t deltaf, deltab;
        page_t *pp;
        page_t *plist = NULL;
        spgcnt_t pagesavail;
        u_offset_t vp_end;

        ASSERT(off >= vp_off && off < vp_off + vp_len);

        /*
         * We only want to do klustering/read ahead if there
         * is more than minfree pages currently available.
         */
        pagesavail = freemem - minfree;

        if (pagesavail <= 0)
                if (isra)
                        return ((page_t *)NULL);    /* ra case - give up */
                else
                        pagesavail = 1;             /* must return a page */

        /* We calculate in pages instead of bytes due to 32-bit overflows */
        if (pagesavail < (spgcnt_t)btopr(vp_len)) {
                /*
                 * Don’t have enough free memory for the
                 * max request, try sizing down vp request.
                 */
```

```
128                 deltab = (ssize_t)(off - vp_off);
129                 vp_len -= deltab;
130                 vp_off += deltab;
131                 if (pagesavail < btopr(vp_len)) {
132                         /*
133                          * Still not enough memory, just settle for
134                          * pagesavail which is at least 1.
135                          */
136                         vp_len = ptob(pagesavail);
137                 }
138         }

140         vp_end = vp_off + vp_len;
141         ASSERT(off >= vp_off && off < vp_end);

143         if (isra && segop_kluster(seg, addr, 0))
143         if (isra && SEGOP_KLUSTER(seg, addr, 0))
144                 return ((page_t *)NULL);        /* segment driver says no */

146         if ((plist = page_create_va(vp, off,
147             PAGESIZE, PG_EXCL | PG_WAIT, seg, addr)) == NULL)
148                 return ((page_t *)NULL);

150         if (vp_len <= PAGESIZE || pvn_nofodklust) {
151                 *offp = off;
152                 *lenp = MIN(vp_len, PAGESIZE);
153         } else {
154                 /*
155                  * Scan back from front by incrementing "deltab" and
156                  * comparing "off" with "vp_off + deltab" to avoid
157                  * "signed" versus "unsigned" conversion problems.
158                  */
159                 for (deltab = PAGESIZE; off >= vp_off + deltab;
160                     deltab += PAGESIZE) {
161                         /*
162                          * Call back to the segment driver to verify that
163                          * the klustering/read ahead operation makes sense.
164                          */
165                         if (segop_kluster(seg, addr, -deltab))
165                         if (SEGOP_KLUSTER(seg, addr, -deltab))
166                                 break;          /* page not eligible */
167                         if ((pp = page_create_va(vp, off - deltab,
168                             PAGESIZE, PG_EXCL, seg, addr - deltab))
169                             == NULL)
170                                 break;          /* already have the page */
171                         /*
172                          * Add page to front of page list.
173                          */
174                         page_add(&plist, pp);
175                 }
176                 deltab -= PAGESIZE;

178                 /* scan forward from front */
179                 for (deltaf = PAGESIZE; off + deltaf < vp_end;
180                     deltaf += PAGESIZE) {
181                         /*
182                          * Call back to the segment driver to verify that
183                          * the klustering/read ahead operation makes sense.
184                          */
185                         if (segop_kluster(seg, addr, deltaf))
185                         if (SEGOP_KLUSTER(seg, addr, deltaf))
186                                 break;          /* page not file extension */
187                         if ((pp = page_create_va(vp, off + deltaf,
188                             PAGESIZE, PG_EXCL, seg, addr + deltaf))
189                             == NULL)
190                                 break;          /* already have page */
```



```
192                         /*
193                          * Add page to end of page list.
194                          */
195                         page_add(&plist, pp);
196                         plist = plist->p_next;
197                 }
198                 *offp = off = off - deltab;
199                 *lenp = deltab + deltaf;
200                 ASSERT(off >= vp_off);

202                 /*
203                  * If we ended up getting more than was actually
204                  * requested, retract the returned length to only
205                  * reflect what was requested.  This might happen
206                  * if we were allowed to kluster pages across a
207                  * span of (say) 5 frags, and frag size is less
208                  * than PAGESIZE.  We need a whole number of
209                  * pages to contain those frags, but the returned
210                  * size should only allow the returned range to
211                  * extend as far as the end of the frags.
212                  */
213                 if ((vp_off + vp_len) < (off + *lenp)) {
214                         ASSERT(vp_end > off);
215                         *lenp = vp_end - off;
216                 }
217         }
218         TRACE_3(TR_FAC_VM, TR_PVN_READ_KLUSTER,
219             "pvn_read_kluster:seg %p addr %x isra %x",
220             seg, addr, isra);
221         return (plist);
222 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    3093 Fri May  8 18:10:38 2015
new/usr/src/uts/common/vm/vm_rm.c
remove xhat
The xhat infrastructure was added to support hardware such as the zulu
graphics card - hardware which had on-board MMUs.  The VM used the xhat code
to keep the CPU's and Zulu's page tables in-sync.  Since the only xhat user
was zulu (which is gone), we can safely remove it simplifying the whole VM
subsystem.
Assorted notes:
- AS_BUSY flag was used solely by xhat
**********************************************************
```

```
/*
 * 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.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */

#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/mman.h>
#include <sys/sysmacros.h>
#include <sys/errno.h>
#include <sys/signal.h>
#include <sys/user.h>
#include <sys/proc.h>
#include <sys/cmn_err.h>
#include <sys/debug.h>

#include <vm/hat.h>
#include <vm/as.h>
#include <vm/seg_vn.h>
```



```
#include <vm/rm.h>
#include <vm/seg.h>
#include <vm/page.h>

/*
 * Yield the memory claim requirement for an address space.
 *
 * This is currently implemented as the number of active hardware
 * translations that have page structures.  Therefore, it can
 * underestimate the traditional resident set size, eg, if the
 * physical page is present and the hardware translation is missing;
 * and it can overestimate the rss, eg, if there are active
 * translations to a frame buffer with page structs.
 * Also, it does not take sharing into account.
 * Also, it does not take sharing and XHATs into account.
 */
size_t
rm_asrss(as)
        register struct as *as;
{
        if (as != (struct as *)NULL && as != &kas)
                return ((size_t)btop(hat_get_mapped_size(as->a_hat)));
        else
                return (0);
}
_____unchanged_portion_omitted_
```

```
**********************************************************
   54157 Fri May  8 18:10:38 2015
new/usr/src/uts/common/vm/vm_seg.c
patch segpcache-maxwindow-is-useless
use NULL dump segop as a shorthand for no-op
Instead of forcing every segment driver to implement a dummy function that
does nothing, handle NULL dump segop function pointer as a no-op shorthand.
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
use NULL setpagesize segop as a shorthand for ENOTSUP
Instead of forcing every segment driver to implement a dummp function to
return (hopefully) ENOTSUP, handle NULL setpagesize segop function pointer
as "return ENOTSUP" shorthand.
use NULL getmemid segop as a shorthand for ENODEV
Instead of forcing every segment driver to implement a dummy function to
return (hopefully) ENODEV, handle NULL getmemid segop function pointer as
"return ENODEV" shorthand.
use NULL capable segop as a shorthand for no-capabilities
Instead of forcing every segment driver to implement a dummy "return 0"
function, handle NULL capable segop function pointer as "no copabilities
supported" shorthand.
seg_inherit_notsup is redundant since segop_inherit checks for NULL properly
patch lower-case-segops
instead using SEGOP_* macros, define full-fledged segop_* functions
This will allow us to do some sanity checking or even implement stub
functionality in one place instead of duplicating it wherever these wrappers
are used.
**********************************************************
______unchanged_portion_omitted_

 113 /*
 114  * A parameter to control a maximum number of bytes that can be
 115  * purged from pcache at a time.
 116  */
 117 #define P_MAX_APURGE_BYTES      (1024 * 1024 * 1024)

 119 /*
 120  * log2(fraction of pcache to reclaim at a time).
 121  */
 122 #define P_SHRINK_SHFT           (5)

 124 /*
 125  * The following variables can be tuned via /etc/system.
 126  */

 128 int     segpcache_enabled = 1;          /* if 1, shadow lists are cached */
 129 pgcnt_t segpcache_maxwindow = 0;        /* max # of pages that can be cached */
 129 ulong_t segpcache_hashsize_win = 0;     /* # of non wired buckets */
 130 ulong_t segpcache_hashsize_wired = 0;   /* # of wired buckets */
 131 int     segpcache_reap_sec = 1;         /* reap check rate in secs */
 132 clock_t segpcache_reap_ticks = 0;       /* reap interval in ticks */
 133 int     segpcache_pcp_maxage_sec = 1;   /* pcp max age in secs */
 134 clock_t segpcache_pcp_maxage_ticks = 0; /* pcp max age in ticks */
 135 int     segpcache_shrink_shift = P_SHRINK_SHFT; /* log2 reap fraction */
 136 pgcnt_t segpcache_maxapurge_bytes = P_MAX_APURGE_BYTES; /* max purge bytes */

 138 static kmutex_t seg_pcache_mtx; /* protects seg_pdisabled counter */
 139 static kmutex_t seg_pasync_mtx; /* protects async thread scheduling */
 140 static kcondvar_t seg_pasync_cv;

 142 #pragma align 64(pctrl1)
 143 #pragma align 64(pctrl2)
 144 #pragma align 64(pctrl3)

 146 /*
 147  * Keep frequently used variables together in one cache line.
```



```
 148  */
 149 static struct p_ctrl1 {
 150         uint_t p_disabled;              /* if not 0, caching temporarily off */
 152         pgcnt_t p_maxwin;               /* max # of pages that can be cached */
 151         size_t p_hashwin_sz;            /* # of non wired buckets */
 152         struct seg_phash *p_htabwin;    /* hash table for non wired entries */
 153         size_t p_hashwired_sz;          /* # of wired buckets */
 154         struct seg_phash_wired *p_htabwired; /* hash table for wired entries */
 155         kmem_cache_t *p_kmcache;        /* kmem cache for seg_pcache structs */
 156 #ifdef _LP64
 157         ulong_t pad[2];
 159         ulong_t pad[1];
 158 #endif /* _LP64 */
 159 } pctrl1;
______unchanged_portion_omitted_

 181 #define seg_pdisabled                   pctrl1.p_disabled
 184 #define seg_pmaxwindow                  pctrl1.p_maxwin
 182 #define seg_phashsize_win               pctrl1.p_hashwin_sz
 183 #define seg_phashtab_win                pctrl1.p_htabwin
 184 #define seg_phashsize_wired             pctrl1.p_hashwired_sz
 185 #define seg_phashtab_wired              pctrl1.p_htabwired
 186 #define seg_pkmcache                    pctrl1.p_kmcache
 187 #define seg_pmem_mtx                    pctrl2.p_mem_mtx
 188 #define seg_plocked_window              pctrl2.p_locked_win
 189 #define seg_plocked                     pctrl2.p_locked
 190 #define seg_pahcur                      pctrl2.p_ahcur
 191 #define seg_pathr_on                    pctrl2.p_athr_on
 192 #define seg_pahhead                     pctrl2.p_ahhead
 193 #define seg_pmax_pcpage                 pctrl3.p_pcp_maxage
 194 #define seg_pathr_empty_ahb             pctrl3.p_athr_empty_ahb
 195 #define seg_pathr_full_ahb              pctrl3.p_athr_full_ahb
 196 #define seg_pshrink_shift               pctrl3.p_shrink_shft
 197 #define seg_pmaxapurge_npages           pctrl3.p_maxapurge_npages

 199 #define P_HASHWIN_MASK                  (seg_phashsize_win - 1)
 200 #define P_HASHWIRED_MASK                (seg_phashsize_wired - 1)
 201 #define P_BASESHIFT                     (6)

 203 kthread_t *seg_pasync_thr;

 205 extern const struct seg_ops segvn_ops;
 206 extern const struct seg_ops segspt_shmops;
 208 extern struct seg_ops segvn_ops;
 209 extern struct seg_ops segspt_shmops;

 208 #define IS_PFLAGS_WIRED(flags) ((flags) & SEGP_FORCE_WIRED)
 209 #define IS_PCP_WIRED(pcp) IS_PFLAGS_WIRED((pcp)->p_flags)

 211 #define LBOLT_DELTA(t)  ((ulong_t)(ddi_get_lbolt() - (t)))

 213 #define PCP_AGE(pcp)    LBOLT_DELTA((pcp)->p_lbolt)

 215 /*
 216  * htag0 argument can be a seg or amp pointer.
 217  */
 218 #define P_HASHBP(seg, htag0, addr, flags)                               \
 219         (IS_PFLAGS_WIRED((flags)) ?                                     \
 220             ((struct seg_phash *)&seg_phashtab_wired[P_HASHWIRED_MASK & \
 221             ((uintptr_t)(htag0) >> P_BASESHIFT)]) :                     \
 222             (&seg_phashtab_win[P_HASHWIN_MASK &                         \
 223             (((uintptr_t)(htag0) >> 3) ^                                \
 224             ((uintptr_t)(addr) >> ((flags & SEGP_PSHIFT) ?              \
 225             (flags >> 16) : page_get_shift((seg)->s_szc))))]))

 227 /*
```

```
 228  * htag0 argument can be a seg or amp pointer.
 229  */
 230 #define P_MATCH(pcp, htag0, addr, len)                          \
 231         ((pcp)->p_htag0 == (htag0) &&                           \
 232         (pcp)->p_addr == (addr) &&                              \
 233         (pcp)->p_len >= (len))

 235 #define P_MATCH_PP(pcp, htag0, addr, len, pp)                   \
 236         ((pcp)->p_pp == (pp) &&                                 \
 237         (pcp)->p_htag0 == (htag0) &&                            \
 238         (pcp)->p_addr == (addr) &&                              \
 239         (pcp)->p_len >= (len))

 241 #define plink2pcache(pl)        ((struct seg_pcache *)((uintptr_t)(pl) - \
 242     offsetof(struct seg_pcache, p_plink)))

 244 #define hlink2phash(hl, l)      ((struct seg_phash *)((uintptr_t)(hl) - \
 245     offsetof(struct seg_phash, p_halink[l])))

 247 /*
 248  * seg_padd_abuck()/seg_premove_abuck() link and unlink hash buckets from
 249  * active hash bucket lists. We maintain active bucket lists to reduce the
 250  * overhead of finding active buckets during asynchronous purging since there
 251  * can be 10s of millions of buckets on a large system but only a small subset
 252  * of them in actual use.
 253  *
 254  * There're 2 active bucket lists. Current active list (as per seg_pahcur) is
 255  * used by seg_pinsert()/seg_pinactive()/seg_ppurge() to add and delete
 256  * buckets. The other list is used by asynchronous purge thread. This allows
 257  * the purge thread to walk its active list without holding seg_pmem_mtx for a
 258  * long time. When asynchronous thread is done with its list it switches to
 259  * current active list and makes the list it just finished processing as
 260  * current active list.
 261  *
 262  * seg_padd_abuck() only adds the bucket to current list if the bucket is not
 263  * yet on any list.  seg_premove_abuck() may remove the bucket from either
 264  * list. If the bucket is on current list it will be always removed. Otherwise
 265  * the bucket is only removed if asynchronous purge thread is not currently
 266  * running or seg_premove_abuck() is called by asynchronous purge thread
 267  * itself. A given bucket can only be on one of active lists at a time. These
 268  * routines should be called with per bucket lock held.  The routines use
 269  * seg_pmem_mtx to protect list updates. seg_padd_abuck() must be called after
 270  * the first entry is added to the bucket chain and seg_premove_abuck() must
 271  * be called after the last pcp entry is deleted from its chain. Per bucket
 272  * lock should be held by the callers.  This avoids a potential race condition
 273  * when seg_premove_abuck() removes a bucket after pcp entries are added to
 274  * its list after the caller checked that the bucket has no entries. (this
 275  * race would cause a loss of an active bucket from the active lists).
 276  *
 277  * Both lists are circular doubly linked lists anchored at seg_pahhead heads.
 278  * New entries are added to the end of the list since LRU is used as the
 279  * purging policy.
 280  */
 281 static void
 282 seg_padd_abuck(struct seg_phash *hp)
 283 {
 284         int lix;

 286         ASSERT(MUTEX_HELD(&hp->p_hmutex));
 287         ASSERT((struct seg_phash *)hp->p_hnext != hp);
 288         ASSERT((struct seg_phash *)hp->p_hprev != hp);
 289         ASSERT(hp->p_hnext == hp->p_hprev);
 290         ASSERT(!IS_PCP_WIRED(hp->p_hnext));
 291         ASSERT(hp->p_hnext->p_hnext == (struct seg_pcache *)hp);
 292         ASSERT(hp->p_hprev->p_hprev == (struct seg_pcache *)hp);
 293         ASSERT(hp >= seg_phashtab_win &&
```



```
 294             hp < &seg_phashtab_win[seg_phashsize_win]);

 296         /*
 297          * This bucket can already be on one of active lists
 298          * since seg_premove_abuck() may have failed to remove it
 299          * before.
 300          */
 301         mutex_enter(&seg_pmem_mtx);
 302         lix = seg_pahcur;
 303         ASSERT(lix >= 0 && lix <= 1);
 304         if (hp->p_halink[lix].p_lnext != NULL) {
 305                 ASSERT(hp->p_halink[lix].p_lprev != NULL);
 306                 ASSERT(hp->p_halink[!lix].p_lnext == NULL);
 307                 ASSERT(hp->p_halink[!lix].p_lprev == NULL);
 308                 mutex_exit(&seg_pmem_mtx);
 309                 return;
 310         }
 311         ASSERT(hp->p_halink[lix].p_lprev == NULL);

 313         /*
 314          * If this bucket is still on list !lix async thread can't yet remove
 315          * it since we hold here per bucket lock. In this case just return
 316          * since async thread will eventually find and process this bucket.
 317          */
 318         if (hp->p_halink[!lix].p_lnext != NULL) {
 319                 ASSERT(hp->p_halink[!lix].p_lprev != NULL);
 320                 mutex_exit(&seg_pmem_mtx);
 321                 return;
 322         }
 323         ASSERT(hp->p_halink[!lix].p_lprev == NULL);
 324         /*
 325          * This bucket is not on any active bucket list yet.
 326          * Add the bucket to the tail of current active list.
 327          */
 328         hp->p_halink[lix].p_lnext = &seg_pahhead[lix];
 329         hp->p_halink[lix].p_lprev = seg_pahhead[lix].p_lprev;
 330         seg_pahhead[lix].p_lprev->p_lnext = &hp->p_halink[lix];
 331         seg_pahhead[lix].p_lprev = &hp->p_halink[lix];
 332         mutex_exit(&seg_pmem_mtx);
 333 }
_____unchanged_portion_omitted_

 726 #ifdef DEBUG
 727 static uint32_t p_insert_chk_mtbf = 0;
 728 #endif

 730 /*
 731  * The seg_pinsert_check() is used by segment drivers to predict whether
 732  * a call to seg_pinsert will fail and thereby avoid wasteful pre-processing.
 733  */
 734 /*ARGSUSED*/
 735 int
 736 seg_pinsert_check(struct seg *seg, struct anon_map *amp, caddr_t addr,
 737     size_t len, uint_t flags)
 738 {
 739         ASSERT(seg != NULL);

 741 #ifdef DEBUG
 742         if (p_insert_chk_mtbf && !(gethrtime() % p_insert_chk_mtbf)) {
 743                 return (SEGP_FAIL);
 744         }
 745 #endif

 747         if (seg_pdisabled) {
 748                 return (SEGP_FAIL);
 749         }
```

```
750         ASSERT(seg_phashsize_win != 0);

752         if (IS_PFLAGS_WIRED(flags)) {
753                 return (SEGP_SUCCESS);
754         }

759         if (seg_plocked_window + btop(len) > seg_pmaxwindow) {
760                 return (SEGP_FAIL);
761         }

756         if (freemem < desfree) {
757                 return (SEGP_FAIL);
758         }

760         return (SEGP_SUCCESS);
761 }

763 #ifdef DEBUG
764 static uint32_t p_insert_mtbf = 0;
765 #endif

767 /*
768  * Insert address range with shadow list into pagelock cache if there's no
769  * shadow list already cached for this address range. If the cache is off or
770  * caching is temporarily disabled or the allowed 'window' is exceeded return
771  * SEGP_FAIL. Otherwise return SEGP_SUCCESS.
772  *
773  * For non wired shadow lists (segvn case) include address in the hashing
774  * function to avoid linking all the entries from the same segment or amp on
775  * the same bucket.  amp is used instead of seg if amp is not NULL. Non wired
776  * pcache entries are also linked on a per segment/amp list so that all
777  * entries can be found quickly during seg/amp purge without walking the
778  * entire pcache hash table.  For wired shadow lists (segspt case) we
779  * don't use address hashing and per segment linking because the caller
780  * currently inserts only one entry per segment that covers the entire
781  * segment. If we used per segment linking even for segspt it would complicate
782  * seg_ppurge_wiredpp() locking.
783  *
784  * Both hash bucket and per seg/amp locks need to be held before adding a non
785  * wired entry to hash and per seg/amp lists. per seg/amp lock should be taken
786  * first.
787  *
788  * This function will also remove from pcache old inactive shadow lists that
789  * overlap with this request but cover smaller range for the same start
790  * address.
791  */
792 int
793 seg_pinsert(struct seg *seg, struct anon_map *amp, caddr_t addr, size_t len,
794     size_t wlen, struct page **pp, enum seg_rw rw, uint_t flags,
795     seg_preclaim_cbfunc_t callback)
796 {
797         struct seg_pcache *pcp;
798         struct seg_phash *hp;
799         pgcnt_t npages;
800         pcache_link_t *pheadp;
801         kmutex_t *pmtx;
802         struct seg_pcache *delcallb_list = NULL;

804         ASSERT(seg != NULL);
805         ASSERT(rw == S_READ || rw == S_WRITE);
806         ASSERT(rw == S_READ || wlen == len);
807         ASSERT(rw == S_WRITE || wlen <= len);
808         ASSERT(amp == NULL || wlen == len);

810 #ifdef DEBUG
811         if (p_insert_mtbf && !(gethrtime() % p_insert_mtbf)) {
```



```
812                 return (SEGP_FAIL);
813         }
814 #endif

816         if (seg_pdisabled) {
817                 return (SEGP_FAIL);
818         }
819         ASSERT(seg_phashsize_win != 0);

821         ASSERT((len & PAGEOFFSET) == 0);
822         npages = btop(len);
823         mutex_enter(&seg_pmem_mtx);
824         if (!IS_PFLAGS_WIRED(flags)) {
832                 if (seg_plocked_window + npages > seg_pmaxwindow) {
833                         mutex_exit(&seg_pmem_mtx);
834                         return (SEGP_FAIL);
835                 }
825                 seg_plocked_window += npages;
826         }
827         seg_plocked += npages;
828         mutex_exit(&seg_pmem_mtx);

830         pcp = kmem_cache_alloc(seg_pkmcache, KM_SLEEP);
831         /*
832          * If amp is not NULL set htag0 to amp otherwise set it to seg.
833          */
834         if (amp == NULL) {
835                 pcp->p_htag0 = (void *)seg;
836                 pcp->p_flags = flags & 0xffff;
837         } else {
838                 pcp->p_htag0 = (void *)amp;
839                 pcp->p_flags = (flags & 0xffff) | SEGP_AMP;
840         }
841         pcp->p_addr = addr;
842         pcp->p_len = len;
843         pcp->p_wlen = wlen;
844         pcp->p_pp = pp;
845         pcp->p_write = (rw == S_WRITE);
846         pcp->p_callback = callback;
847         pcp->p_active = 1;

849         hp = P_HASHBP(seg, pcp->p_htag0, addr, flags);
850         if (!IS_PFLAGS_WIRED(flags)) {
851                 int found;
852                 void *htag0;
853                 if (amp == NULL) {
854                         pheadp = &seg->s_phead;
855                         pmtx = &seg->s_pmtx;
856                         htag0 = (void *)seg;
857                 } else {
858                         pheadp = &amp->a_phead;
859                         pmtx = &amp->a_pmtx;
860                         htag0 = (void *)amp;
861                 }
862                 mutex_enter(pmtx);
863                 mutex_enter(&hp->p_hmutex);
864                 delcallb_list = seg_plookup_checkdup(hp, htag0, addr,
865                     len, &found);
866                 if (found) {
867                         mutex_exit(&hp->p_hmutex);
868                         mutex_exit(pmtx);
869                         mutex_enter(&seg_pmem_mtx);
870                         seg_plocked -= npages;
871                         seg_plocked_window -= npages;
872                         mutex_exit(&seg_pmem_mtx);
873                         kmem_cache_free(seg_pkmcache, pcp);
```

```
874                         goto out;
875                 }
876                 pcp->p_plink.p_lnext = pheadp->p_lnext;
877                 pcp->p_plink.p_lprev = pheadp;
878                 pheadp->p_lnext->p_lprev = &pcp->p_plink;
879                 pheadp->p_lnext = &pcp->p_plink;
880         } else {
881                 mutex_enter(&hp->p_hmutex);
882         }
883         pcp->p_hashp = hp;
884         pcp->p_hnext = hp->p_hnext;
885         pcp->p_hprev = (struct seg_pcache *)hp;
886         hp->p_hnext->p_hprev = pcp;
887         hp->p_hnext = pcp;
888         if (!IS_PFLAGS_WIRED(flags) &&
889             hp->p_hprev == pcp) {
890                 seg_padd_abuck(hp);
891         }
892         mutex_exit(&hp->p_hmutex);
893         if (!IS_PFLAGS_WIRED(flags)) {
894                 mutex_exit(pmtx);
895         }

897 out:
898         npages = 0;
899         while (delcallb_list != NULL) {
900                 pcp = delcallb_list;
901                 delcallb_list = pcp->p_hprev;
902                 ASSERT(!IS_PCP_WIRED(pcp) && !pcp->p_active);
903                 (void) (*pcp->p_callback)(pcp->p_htag0, pcp->p_addr,
904                     pcp->p_len, pcp->p_pp, pcp->p_write ? S_WRITE : S_READ, 0);
905                 npages += btop(pcp->p_len);
906                 kmem_cache_free(seg_pkmcache, pcp);
907         }
908         if (npages) {
909                 ASSERT(!IS_PFLAGS_WIRED(flags));
910                 mutex_enter(&seg_pmem_mtx);
911                 ASSERT(seg_plocked >= npages);
912                 ASSERT(seg_plocked_window >= npages);
913                 seg_plocked -= npages;
914                 seg_plocked_window -= npages;
915                 mutex_exit(&seg_pmem_mtx);
916         }

918         return (SEGP_SUCCESS);
919 }

921 /*
922  * purge entries from the pagelock cache if not active
923  * and not recently used.
924  */
925 static void
926 seg_ppurge_async(int force)
927 {
928         struct seg_pcache *delcallb_list = NULL;
929         struct seg_pcache *pcp;
930         struct seg_phash *hp;
931         pgcnt_t npages = 0;
932         pgcnt_t npages_window = 0;
933         pgcnt_t npgs_to_purge;
934         pgcnt_t npgs_purged = 0;
935         int hlinks = 0;
936         int hlix;
937         pcache_link_t *hlinkp;
938         pcache_link_t *hlnextp = NULL;
939         int lowmem;
```



```
951         int trim;

941         ASSERT(seg_phashsize_win != 0);

943         /*
944          * if the cache is off or empty, return
945          */
946         if (seg_plocked == 0 || (!force && seg_plocked_window == 0)) {
947                 return;
948         }

950         if (!force) {
951                 lowmem = 0;
964                 trim = 0;
952                 if (freemem < lotsfree + needfree) {
953                         spgcnt_t fmem = MAX((spgcnt_t)(freemem - needfree), 0);
954                         if (fmem <= 5 * (desfree >> 2)) {
955                                 lowmem = 1;
956                         } else if (fmem <= 7 * (lotsfree >> 3)) {
957                                 if (seg_plocked_window >=
958                                     (availrmem_initial >> 1)) {
959                                         lowmem = 1;
960                                 }
961                         } else if (fmem < lotsfree) {
962                                 if (seg_plocked_window >=
963                                     3 * (availrmem_initial >> 2)) {
964                                         lowmem = 1;
965                                 }
966                         }
967                 }
968                 if (!lowmem) {
981                 if (seg_plocked_window >= 7 * (seg_pmaxwindow >> 3)) {
982                         trim = 1;
983                 }
984                 if (!lowmem && !trim) {
969                         return;
970                 }
971                 npgs_to_purge = seg_plocked_window >>
972                     seg_pshrink_shift;
973                 if (lowmem) {
974                         npgs_to_purge = MIN(npgs_to_purge,
975                             MAX(seg_pmaxapurge_npages, desfree));
976                 } else {
977                         npgs_to_purge = MIN(npgs_to_purge,
978                             seg_pmaxapurge_npages);
979                 }
980                 if (npgs_to_purge == 0) {
981                         return;
982                 }
983         } else {
984                 struct seg_phash_wired *hpw;

986                 ASSERT(seg_phashsize_wired != 0);

988                 for (hpw = seg_phashtab_wired;
989                     hpw < &seg_phashtab_wired[seg_phashsize_wired]; hpw++) {

991                         if (hpw->p_hnext == (struct seg_pcache *)hpw) {
992                                 continue;
993                         }

995                         mutex_enter(&hpw->p_hmutex);

997                         for (pcp = hpw->p_hnext;
998                             pcp != (struct seg_pcache *)hpw;
999                             pcp = pcp->p_hnext) {
```

```
1001                                    ASSERT(IS_PCP_WIRED(pcp));
1002                                    ASSERT(pcp->p_hashp ==
1003                                        (struct seg_phash *)hpw);
1004
1005                                    if (pcp->p_active) {
1006                                            continue;
1007                                    }
1008                                    pcp->p_hprev->p_hnext = pcp->p_hnext;
1009                                    pcp->p_hnext->p_hprev = pcp->p_hprev;
1010                                    pcp->p_hprev = delcallb_list;
1011                                    delcallb_list = pcp;
1012                            }
1013                            mutex_exit(&hpw->p_hmutex);
1014                    }
1015            }
1016
1017            mutex_enter(&seg_pmem_mtx);
1018            if (seg_pathr_on) {
1019                    mutex_exit(&seg_pmem_mtx);
1020                    goto runcb;
1021            }
1022            seg_pathr_on = 1;
1023            mutex_exit(&seg_pmem_mtx);
1024            ASSERT(seg_pahcur <= 1);
1025            hlix = !seg_pahcur;
1026
1027    again:
1028            for (hlinkp = seg_pahhead[hlix].p_lnext; hlinkp != &seg_pahhead[hlix];
1029                hlinkp = hlnextp) {
1030
1031                    hlnextp = hlinkp->p_lnext;
1032                    ASSERT(hlnextp != NULL);
1033
1034                    hp = hlink2phash(hlinkp, hlix);
1035                    if (hp->p_hnext == (struct seg_pcache *)hp) {
1036                            seg_pathr_empty_ahb++;
1037                            continue;
1038                    }
1039                    seg_pathr_full_ahb++;
1040                    mutex_enter(&hp->p_hmutex);
1041
1042                    for (pcp = hp->p_hnext; pcp != (struct seg_pcache *)hp;
1043                        pcp = pcp->p_hnext) {
1044                            pcache_link_t *pheadp;
1045                            pcache_link_t *plinkp;
1046                            void *htag0;
1047                            kmutex_t *pmtx;
1048
1049                            ASSERT(!IS_PCP_WIRED(pcp));
1050                            ASSERT(pcp->p_hashp == hp);
1051
1052                            if (pcp->p_active) {
1053                                    continue;
1054                            }
1055                            if (!force && pcp->p_ref &&
1056                                PCP_AGE(pcp) < seg_pmax_pcpage) {
1057                                    pcp->p_ref = 0;
1058                                    continue;
1059                            }
1060                            plinkp = &pcp->p_plink;
1061                            htag0 = pcp->p_htag0;
1062                            if (pcp->p_flags & SEGP_AMP) {
1063                                    pheadp = &((amp_t *)htag0)->a_phead;
1064                                    pmtx = &((amp_t *)htag0)->a_pmtx;
1065                            } else {
```



```
1066                                    pheadp = &((seg_t *)htag0)->s_phead;
1067                                    pmtx = &((seg_t *)htag0)->s_pmtx;
1068                            }
1069                            if (!mutex_tryenter(pmtx)) {
1070                                    continue;
1071                            }
1072                            ASSERT(pheadp->p_lnext != pheadp);
1073                            ASSERT(pheadp->p_lprev != pheadp);
1074                            plinkp->p_lprev->p_lnext =
1075                                plinkp->p_lnext;
1076                            plinkp->p_lnext->p_lprev =
1077                                plinkp->p_lprev;
1078                            pcp->p_hprev->p_hnext = pcp->p_hnext;
1079                            pcp->p_hnext->p_hprev = pcp->p_hprev;
1080                            mutex_exit(pmtx);
1081                            pcp->p_hprev = delcallb_list;
1082                            delcallb_list = pcp;
1083                            npgs_purged += btop(pcp->p_len);
1084                    }
1085                    if (hp->p_hnext == (struct seg_pcache *)hp) {
1086                            seg_premove_abuck(hp, 1);
1087                    }
1088                    mutex_exit(&hp->p_hmutex);
1089                    if (npgs_purged >= seg_plocked_window) {
1090                            break;
1091                    }
1092                    if (!force) {
1093                            if (npgs_purged >= npgs_to_purge) {
1094                                    break;
1095                            }
1096                            if (!(seg_pathr_full_ahb & 15)) {
1112                            if (!trim && !(seg_pathr_full_ahb & 15)) {
1097                                    ASSERT(lowmem);
1098                                    if (freemem >= lotsfree + needfree) {
1099                                            break;
1100                                    }
1101                            }
1102                    }
1103            }
1104
1105            if (hlinkp == &seg_pahhead[hlix]) {
1106                    /*
1107                     * We processed the entire hlix active bucket list
1108                     * but didn't find enough pages to reclaim.
1109                     * Switch the lists and walk the other list
1110                     * if we haven't done it yet.
1111                     */
1112                    mutex_enter(&seg_pmem_mtx);
1113                    ASSERT(seg_pathr_on);
1114                    ASSERT(seg_pahcur == !hlix);
1115                    seg_pahcur = hlix;
1116                    mutex_exit(&seg_pmem_mtx);
1117                    if (++hlinks < 2) {
1118                            hlix = !hlix;
1119                            goto again;
1120                    }
1121            } else if ((hlinkp = hlnextp) != &seg_pahhead[hlix] &&
1122                seg_pahhead[hlix].p_lnext != hlinkp) {
1123                    ASSERT(hlinkp != NULL);
1124                    ASSERT(hlinkp->p_lprev != &seg_pahhead[hlix]);
1125                    ASSERT(seg_pahhead[hlix].p_lnext != &seg_pahhead[hlix]);
1126                    ASSERT(seg_pahhead[hlix].p_lprev != &seg_pahhead[hlix]);
1127
1128                    /*
1129                     * Reinsert the header to point to hlinkp
1130                     * so that we start from hlinkp bucket next time around.
```

```
1131                  */
1132                 seg_pahhead[hlix].p_lnext->p_lprev = seg_pahhead[hlix].p_lprev;
1133                 seg_pahhead[hlix].p_lprev->p_lnext = seg_pahhead[hlix].p_lnext;
1134                 seg_pahhead[hlix].p_lnext = hlinkp;
1135                 seg_pahhead[hlix].p_lprev = hlinkp->p_lprev;
1136                 hlinkp->p_lprev->p_lnext = &seg_pahhead[hlix];
1137                 hlinkp->p_lprev = &seg_pahhead[hlix];
1138         }

1140         mutex_enter(&seg_pmem_mtx);
1141         ASSERT(seg_pathr_on);
1142         seg_pathr_on = 0;
1143         mutex_exit(&seg_pmem_mtx);

1145 runcb:
1146         /*
1147          * Run the delayed callback list. segments/amps can't go away until
1148          * callback is executed since they must have non 0 softlockcnt. That's
1149          * why we don't need to hold as/seg/amp locks to execute the callback.
1150          */
1151         while (delcallb_list != NULL) {
1152                 pcp = delcallb_list;
1153                 delcallb_list = pcp->p_hprev;
1154                 ASSERT(!pcp->p_active);
1155                 (void) (*pcp->p_callback)(pcp->p_htag0, pcp->p_addr,
1156                     pcp->p_len, pcp->p_pp, pcp->p_write ? S_WRITE : S_READ, 1);
1157                 npages += btop(pcp->p_len);
1158                 if (!IS_PCP_WIRED(pcp)) {
1159                         npages_window += btop(pcp->p_len);
1160                 }
1161                 kmem_cache_free(seg_pkmcache, pcp);
1162         }
1163         if (npages) {
1164                 mutex_enter(&seg_pmem_mtx);
1165                 ASSERT(seg_plocked >= npages);
1166                 ASSERT(seg_plocked_window >= npages_window);
1167                 seg_plocked -= npages;
1168                 seg_plocked_window -= npages_window;
1169                 mutex_exit(&seg_pmem_mtx);
1170         }
1171 }
```

*unchanged_portion_omitted_*

```
1348 static void seg_pinit_mem_config(void);

1350 /*
1351  * setup the pagelock cache
1352  */
1353 static void
1354 seg_pinit(void)
1355 {
1356         struct seg_phash *hp;
1357         ulong_t i;
1358         pgcnt_t physmegs;

1360         seg_plocked = 0;
1361         seg_plocked_window = 0;

1363         if (segpcache_enabled == 0) {
1364                 seg_phashsize_win = 0;
1365                 seg_phashsize_wired = 0;
1366                 seg_pdisabled = 1;
1367                 return;
1368         }

1370         seg_pdisabled = 0;
```



```
1371         seg_pkmcache = kmem_cache_create("seg_pcache",
1372             sizeof (struct seg_pcache), 0, NULL, NULL, NULL, NULL, NULL, 0);
1373         if (segpcache_pcp_maxage_ticks <= 0) {
1374                 segpcache_pcp_maxage_ticks = segpcache_pcp_maxage_sec * hz;
1375         }
1376         seg_pmax_pcpage = segpcache_pcp_maxage_ticks;
1377         seg_pathr_empty_ahb = 0;
1378         seg_pathr_full_ahb = 0;
1379         seg_pshrink_shift = segpcache_shrink_shift;
1380         seg_pmaxapurge_npages = btop(segpcache_maxapurge_bytes);

1382         mutex_init(&seg_pcache_mtx, NULL, MUTEX_DEFAULT, NULL);
1383         mutex_init(&seg_pmem_mtx, NULL, MUTEX_DEFAULT, NULL);
1384         mutex_init(&seg_pasync_mtx, NULL, MUTEX_DEFAULT, NULL);
1385         cv_init(&seg_pasync_cv, NULL, CV_DEFAULT, NULL);

1387         physmegs = physmem >> (20 - PAGESHIFT);

1389         /*
1390          * If segpcache_hashsize_win was not set in /etc/system or it has
1391          * absurd value set it to a default.
1392          */
1393         if (segpcache_hashsize_win == 0 || segpcache_hashsize_win > physmem) {
1394                 /*
1395                  * Create one bucket per 32K (or at least per 8 pages) of
1396                  * available memory.
1397                  */
1398                 pgcnt_t pages_per_bucket = MAX(btop(32 * 1024), 8);
1399                 segpcache_hashsize_win = MAX(1024, physmem / pages_per_bucket);
1400         }
1401         if (!ISP2(segpcache_hashsize_win)) {
1402                 ulong_t rndfac = ~(1UL <<
1403                     (highbit(segpcache_hashsize_win) - 1));
1404                 rndfac &= segpcache_hashsize_win;
1405                 segpcache_hashsize_win += rndfac;
1406                 segpcache_hashsize_win = 1 <<
1407                     (highbit(segpcache_hashsize_win) - 1);
1408         }
1409         seg_phashsize_win = segpcache_hashsize_win;
1410         seg_phashtab_win = kmem_zalloc(
1411             seg_phashsize_win * sizeof (struct seg_phash),
1412             KM_SLEEP);
1413         for (i = 0; i < seg_phashsize_win; i++) {
1414                 hp = &seg_phashtab_win[i];
1415                 hp->p_hnext = (struct seg_pcache *)hp;
1416                 hp->p_hprev = (struct seg_pcache *)hp;
1417                 mutex_init(&hp->p_hmutex, NULL, MUTEX_DEFAULT, NULL);
1418         }

1420         seg_pahcur = 0;
1421         seg_pathr_on = 0;
1422         seg_pahhead[0].p_lnext = &seg_pahhead[0];
1423         seg_pahhead[0].p_lprev = &seg_pahhead[0];
1424         seg_pahhead[1].p_lnext = &seg_pahhead[1];
1425         seg_pahhead[1].p_lprev = &seg_pahhead[1];

1427         /*
1428          * If segpcache_hashsize_wired was not set in /etc/system or it has
1429          * absurd value set it to a default.
1430          */
1431         if (segpcache_hashsize_wired == 0 ||
1432             segpcache_hashsize_wired > physmem / 4) {
1433                 /*
1434                  * Choose segpcache_hashsize_wired based on physmem.
1435                  * Create a bucket per 128K bytes upto 256K buckets.
1436                  */
```

```
1437                 if (physmegs < 20 * 1024) {
1438                         segpcache_hashsize_wired = MAX(1024, physmegs << 3);
1439                 } else {
1440                         segpcache_hashsize_wired = 256 * 1024;
1441                 }
1442         }
1443         if (!ISP2(segpcache_hashsize_wired)) {
1444                 segpcache_hashsize_wired = 1 <<
1445                     highbit(segpcache_hashsize_wired);
1446         }
1447         seg_phashsize_wired = segpcache_hashsize_wired;
1448         seg_phashtab_wired = kmem_zalloc(
1449             seg_phashsize_wired * sizeof (struct seg_phash_wired), KM_SLEEP);
1450         for (i = 0; i < seg_phashsize_wired; i++) {
1451                 hp = (struct seg_phash *)&seg_phashtab_wired[i];
1452                 hp->p_hnext = (struct seg_pcache *)hp;
1453                 hp->p_hprev = (struct seg_pcache *)hp;
1454                 mutex_init(&hp->p_hmutex, NULL, MUTEX_DEFAULT, NULL);
1455         }

1473         if (segpcache_maxwindow == 0) {
1474                 if (physmegs < 64) {
1475                         /* 3% of memory */
1476                         segpcache_maxwindow = availrmem >> 5;
1477                 } else if (physmegs < 512) {
1478                         /* 12% of memory */
1479                         segpcache_maxwindow = availrmem >> 3;
1480                 } else if (physmegs < 1024) {
1481                         /* 25% of memory */
1482                         segpcache_maxwindow = availrmem >> 2;
1483                 } else if (physmegs < 2048) {
1484                         /* 50% of memory */
1485                         segpcache_maxwindow = availrmem >> 1;
1486                 } else {
1487                         /* no limit */
1488                         segpcache_maxwindow = (pgcnt_t)-1;
1489                 }
1490         }
1491         seg_pmaxwindow = segpcache_maxwindow;
1457         seg_pinit_mem_config();
1458 }
```

_____***unchanged_portion_omitted_***

```
1601 /*
1602  * Unmap a segment and free it from its associated address space.
1603  * This should be called by anybody who's finished with a whole segment's
1604  * mapping.  Just calls segop_unmap() on the whole mapping .  It is the
1639  * mapping.  Just calls SEGOP_UNMAP() on the whole mapping .  It is the
1605  * responsibility of the segment driver to unlink the the segment
1606  * from the address space, and to free public and private data structures
1607  * associated with the segment.  (This is typically done by a call to
1608  * seg_free()).
1609  */
1610 void
1611 seg_unmap(struct seg *seg)
1612 {
1613 #ifdef DEBUG
1614         int ret;
1615 #endif /* DEBUG */

1617         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

1619         /* Shouldn't have called seg_unmap if mapping isn't yet established */
1620         ASSERT(seg->s_data != NULL);

1622         /* Unmap the whole mapping */
```



```
1623 #ifdef DEBUG
1624         ret = segop_unmap(seg, seg->s_base, seg->s_size);
1659         ret = SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
1625         ASSERT(ret == 0);
1626 #else
1627         segop_unmap(seg, seg->s_base, seg->s_size);
1662         SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
1628 #endif /* DEBUG */
1629 }

1631 /*
1632  * Free the segment from its associated as. This should only be called
1633  * if a mapping to the segment has not yet been established (e.g., if
1634  * an error occurs in the middle of doing an as_map when the segment
1635  * has already been partially set up) or if it has already been deleted
1636  * (e.g., from a segment driver unmap routine if the unmap applies to the
1637  * entire segment). If the mapping is currently set up then seg_unmap() should
1638  * be called instead.
1639  */
1640 void
1641 seg_free(struct seg *seg)
1642 {
1643         register struct as *as = seg->s_as;
1644         struct seg *tseg = as_removeseg(as, seg);

1646         ASSERT(tseg == seg);

1648         /*
1649          * If the segment private data field is NULL,
1650          * then segment driver is not attached yet.
1651          */
1652         if (seg->s_data != NULL)
1653                 segop_free(seg);
1688                 SEGOP_FREE(seg);

1655         mutex_destroy(&seg->s_pmtx);
1656         ASSERT(seg->s_phead.p_lnext == &seg->s_phead);
1657         ASSERT(seg->s_phead.p_lprev == &seg->s_phead);
1658         kmem_cache_free(seg_cache, seg);
1659 }
```

_____***unchanged_portion_omitted_***

```
1821 /*
1822  * segop wrappers
1857  * General not supported function for SEGOP_INHERIT
1823  */
1859 /* ARGSUSED */
1824 int
1825 segop_dup(struct seg *seg, struct seg *new)
1826 {
1827         VERIFY3P(seg->s_ops->dup, !=, NULL);

1829         return (seg->s_ops->dup(seg, new));
1830 }

1832 int
1833 segop_unmap(struct seg *seg, caddr_t addr, size_t len)
1861 seg_inherit_notsup(struct seg *seg, caddr_t addr, size_t len, uint_t op)
1834 {
1835         VERIFY3P(seg->s_ops->unmap, !=, NULL);

1837         return (seg->s_ops->unmap(seg, addr, len));
1838 }

1840 void
1841 segop_free(struct seg *seg)
```

```
1842 {
1843         VERIFY3P(seg->s_ops->free, !=, NULL);
1844
1845         seg->s_ops->free(seg);
1846 }
1847
1848 faultcode_t
1849 segop_fault(struct hat *hat, struct seg *seg, caddr_t addr, size_t len,
1850     enum fault_type type, enum seg_rw rw)
1851 {
1852         VERIFY3P(seg->s_ops->fault, !=, NULL);
1853
1854         return (seg->s_ops->fault(hat, seg, addr, len, type, rw));
1855 }
1856
1857 faultcode_t
1858 segop_faulta(struct seg *seg, caddr_t addr)
1859 {
1860         VERIFY3P(seg->s_ops->faulta, !=, NULL);
1861
1862         return (seg->s_ops->faulta(seg, addr));
1863 }
1864
1865 int
1866 segop_setprot(struct seg *seg, caddr_t addr, size_t len, uint_t prot)
1867 {
1868         VERIFY3P(seg->s_ops->setprot, !=, NULL);
1869
1870         return (seg->s_ops->setprot(seg, addr, len, prot));
1871 }
1872
1873 int
1874 segop_checkprot(struct seg *seg, caddr_t addr, size_t len, uint_t prot)
1875 {
1876         VERIFY3P(seg->s_ops->checkprot, !=, NULL);
1877
1878         return (seg->s_ops->checkprot(seg, addr, len, prot));
1879 }
1880
1881 int
1882 segop_kluster(struct seg *seg, caddr_t addr, ssize_t d)
1883 {
1884         VERIFY3P(seg->s_ops->kluster, !=, NULL);
1885
1886         return (seg->s_ops->kluster(seg, addr, d));
1887 }
1888
1889 int
1890 segop_sync(struct seg *seg, caddr_t addr, size_t len, int atr, uint_t f)
1891 {
1892         VERIFY3P(seg->s_ops->sync, !=, NULL);
1893
1894         return (seg->s_ops->sync(seg, addr, len, atr, f));
1895 }
1896
1897 size_t
1898 segop_incore(struct seg *seg, caddr_t addr, size_t len, char *v)
1899 {
1900         VERIFY3P(seg->s_ops->incore, !=, NULL);
1901
1902         return (seg->s_ops->incore(seg, addr, len, v));
1903 }
1904
1905 int
1906 segop_lockop(struct seg *seg, caddr_t addr, size_t len, int atr, int op,
1907     ulong_t *b, size_t p)
```



```
1908 {
1909         VERIFY3P(seg->s_ops->lockop, !=, NULL);
1910
1911         return (seg->s_ops->lockop(seg, addr, len, atr, op, b, p));
1912 }
1913
1914 int
1915 segop_getprot(struct seg *seg, caddr_t addr, size_t len, uint_t *p)
1916 {
1917         VERIFY3P(seg->s_ops->getprot, !=, NULL);
1918
1919         return (seg->s_ops->getprot(seg, addr, len, p));
1920 }
1921
1922 u_offset_t
1923 segop_getoffset(struct seg *seg, caddr_t addr)
1924 {
1925         VERIFY3P(seg->s_ops->getoffset, !=, NULL);
1926
1927         return (seg->s_ops->getoffset(seg, addr));
1928 }
1929
1930 int
1931 segop_gettype(struct seg *seg, caddr_t addr)
1932 {
1933         VERIFY3P(seg->s_ops->gettype, !=, NULL);
1934
1935         return (seg->s_ops->gettype(seg, addr));
1936 }
1937
1938 int
1939 segop_getvp(struct seg *seg, caddr_t addr, struct vnode **vpp)
1940 {
1941         VERIFY3P(seg->s_ops->getvp, !=, NULL);
1942
1943         return (seg->s_ops->getvp(seg, addr, vpp));
1944 }
1945
1946 int
1947 segop_advise(struct seg *seg, caddr_t addr, size_t len, uint_t b)
1948 {
1949         VERIFY3P(seg->s_ops->advise, !=, NULL);
1950
1951         return (seg->s_ops->advise(seg, addr, len, b));
1952 }
1953
1954 void
1955 segop_dump(struct seg *seg)
1956 {
1957         if (seg->s_ops->dump == NULL)
1958                 return;
1959
1960         seg->s_ops->dump(seg);
1961 }
1962
1963 int
1964 segop_pagelock(struct seg *seg, caddr_t addr, size_t len, struct page ***page,
1965     enum lock_type type, enum seg_rw rw)
1966 {
1967         VERIFY3P(seg->s_ops->pagelock, !=, NULL);
1968
1969         return (seg->s_ops->pagelock(seg, addr, len, page, type, rw));
1970 }
1971
1972 int
1973 segop_setpagesize(struct seg *seg, caddr_t addr, size_t len, uint_t szc)
```

```
1974 {
1975         if (seg->s_ops->setpagesize == NULL)
1976                 return (ENOTSUP);
1977
1978         return (seg->s_ops->setpagesize(seg, addr, len, szc));
1979 }
1980
1981 int
1982 segop_getmemid(struct seg *seg, caddr_t addr, memid_t *mp)
1983 {
1984         if (seg->s_ops->getmemid == NULL)
1985                 return (ENODEV);
1986
1987         return (seg->s_ops->getmemid(seg, addr, mp));
1988 }
1989
1990 struct lgrp_mem_policy_info *
1991 segop_getpolicy(struct seg *seg, caddr_t addr)
1992 {
1993         if (seg->s_ops->getpolicy == NULL)
1994                 return (NULL);
1995
1996         return (seg->s_ops->getpolicy(seg, addr));
1997 }
1998
1999 int
2000 segop_capable(struct seg *seg, segcapability_t cap)
2001 {
2002         if (seg->s_ops->capable == NULL)
2003                 return (0);
2004
2005         return (seg->s_ops->capable(seg, cap));
2006 }
2007
2008 int
2009 segop_inherit(struct seg *seg, caddr_t addr, size_t len, uint_t op)
2010 {
2011         if (seg->s_ops->inherit == NULL)
2012 #endif /* ! codereview */
2013                 return (ENOTSUP);
2014
2015         return (seg->s_ops->inherit(seg, addr, len, op));
2016 #endif /* ! codereview */
2017 }
```

****
57936 Fri May 8 18:10:38 2015
new/usr/src/uts/common/vm/vm_usage.c
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
****
_____***unchanged_portion_omitted_***

```
297 extern struct as kas;
298 extern proc_t *practive;
299 extern zone_t *global_zone;
300 extern const struct seg_ops segvn_ops;
301 extern const struct seg_ops segspt_shmops;
300 extern struct seg_ops segvn_ops;
301 extern struct seg_ops segspt_shmops;

303 static vmu_data_t vmu_data;
304 static kmem_cache_t *vmu_bound_cache;
305 static kmem_cache_t *vmu_object_cache;

307 /*
308  * Comparison routine for AVL tree. We base our comparison on vmb_start.
309  */
310 static int
311 bounds_cmp(const void *bnd1, const void *bnd2)
312 {
313         const vmu_bound_t *bound1 = bnd1;
314         const vmu_bound_t *bound2 = bnd2;

316         if (bound1->vmb_start == bound2->vmb_start) {
317                 return (0);
318         }
319         if (bound1->vmb_start < bound2->vmb_start) {
320                 return (-1);
321         }

323         return (1);
324 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  142125 Fri May  8 18:10:39 2015
new/usr/src/uts/i86pc/io/rootnex.c
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
**********************************************************
______unchanged_portion_omitted_
```

```
 343 #endif

 345 /*
 346  *  extern hacks
 347  */
 348 extern const struct seg_ops segdev_ops;
 348 extern struct seg_ops segdev_ops;
 349 extern int ignore_hardware_nodes;       /* force flag from ddi_impl.c */
 350 #ifdef  DDI_MAP_DEBUG
 351 extern int ddi_map_debug_flag;
 352 #define ddi_map_debug   if (ddi_map_debug_flag) prom_printf
 353 #endif
 354 extern void i86_pp_map(page_t *pp, caddr_t kaddr);
 355 extern void i86_va_map(caddr_t vaddr, struct as *asp, caddr_t kaddr);
 356 extern int (*psm_intr_ops)(dev_info_t *, ddi_intr_handle_impl_t *,
 357     psm_intr_op_t, int *);
 358 extern int impl_ddi_sunbus_initchild(dev_info_t *dip);
 359 extern void impl_ddi_sunbus_removechild(dev_info_t *dip);

 361 /*
 362  * Use device arena to use for device control register mappings.
 363  * Various kernel memory walkers (debugger, dtrace) need to know
 364  * to avoid this address range to prevent undesired device activity.
 365  */
 366 extern void *device_arena_alloc(size_t size, int vm_flag);
 367 extern void device_arena_free(void * vaddr, size_t size);


 370 /*
 371  *  Internal functions
 372  */
 373 static int rootnex_dma_init();
 374 static void rootnex_add_props(dev_info_t *);
 375 static int rootnex_ctl_reportdev(dev_info_t *dip);
 376 static struct intrspec *rootnex_get_ispec(dev_info_t *rdip, int inum);
 377 static int rootnex_map_regspec(ddi_map_req_t *mp, caddr_t *vaddrp);
 378 static int rootnex_unmap_regspec(ddi_map_req_t *mp, caddr_t *vaddrp);
 379 static int rootnex_map_handle(ddi_map_req_t *mp);
 380 static void rootnex_clean_dmahdl(ddi_dma_impl_t *hp);
 381 static int rootnex_valid_alloc_parms(ddi_dma_attr_t *attr, uint_t maxsegsize);
 382 static int rootnex_valid_bind_parms(ddi_dma_req_t *dmareq,
 383     ddi_dma_attr_t *attr);
 384 static void rootnex_get_sgl(ddi_dma_obj_t *dmar_object, ddi_dma_cookie_t *sgl,
 385     rootnex_sglinfo_t *sglinfo);
 386 static void rootnex_dvma_get_sgl(ddi_dma_obj_t *dmar_object,
 387     ddi_dma_cookie_t *sgl, rootnex_sglinfo_t *sglinfo);
 388 static int rootnex_bind_slowpath(ddi_dma_impl_t *hp, struct ddi_dma_req *dmareq,
 389     rootnex_dma_t *dma, ddi_dma_attr_t *attr, ddi_dma_obj_t *dmao, int kmflag);
 390 static int rootnex_setup_copybuf(ddi_dma_impl_t *hp, struct ddi_dma_req *dmareq,
 391     rootnex_dma_t *dma, ddi_dma_attr_t *attr);
 392 static void rootnex_teardown_copybuf(rootnex_dma_t *dma);
 393 static int rootnex_setup_windows(ddi_dma_impl_t *hp, rootnex_dma_t *dma,
 394     ddi_dma_attr_t *attr, ddi_dma_obj_t *dmao, int kmflag);
 395 static void rootnex_teardown_windows(rootnex_dma_t *dma);
 396 static void rootnex_init_win(ddi_dma_impl_t *hp, rootnex_dma_t *dma,
 397     rootnex_window_t *window, ddi_dma_cookie_t *cookie, off_t cur_offset);
 398 static void rootnex_setup_cookie(ddi_dma_obj_t *dmar_object,
 399     rootnex_dma_t *dma, ddi_dma_cookie_t *cookie, off_t cur_offset,
 400     size_t *copybuf_used, page_t **cur_pp);
```



```
 401 static int rootnex_sgllen_window_boundary(ddi_dma_impl_t *hp,
 402     rootnex_dma_t *dma, rootnex_window_t **windowp, ddi_dma_cookie_t *cookie,
 403     ddi_dma_attr_t *attr, off_t cur_offset);
 404 static int rootnex_copybuf_window_boundary(ddi_dma_impl_t *hp,
 405     rootnex_dma_t *dma, rootnex_window_t **windowp,
 406     ddi_dma_cookie_t *cookie, off_t cur_offset, size_t *copybuf_used);
 407 static int rootnex_maxxfer_window_boundary(ddi_dma_impl_t *hp,
 408     rootnex_dma_t *dma, rootnex_window_t **windowp, ddi_dma_cookie_t *cookie);
 409 static int rootnex_valid_sync_parms(ddi_dma_impl_t *hp, rootnex_window_t *win,
 410     off_t offset, size_t size, uint_t cache_flags);
 411 static int rootnex_verify_buffer(rootnex_dma_t *dma);
 412 static int rootnex_dma_check(dev_info_t *dip, const void *handle,
 413     const void *comp_addr, const void *not_used);
 414 static boolean_t rootnex_need_bounce_seg(ddi_dma_obj_t *dmar_object,
 415     rootnex_sglinfo_t *sglinfo);
 416 static struct as *rootnex_get_as(ddi_dma_obj_t *dmar_object);

 418 /*
 419  * _init()
 420  *
 421  */
 422 int
 423 _init(void)
 424 {

 426         rootnex_state = NULL;
 427         return (mod_install(&rootnex_modlinkage));
 428 }
```

```
______unchanged_portion_omitted_
```

```
**********************************************************
   13648 Fri May  8 18:10:39 2015
new/usr/src/uts/i86pc/os/mlsetup.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_


 95 /*
 96  * Setup routine called right before main(). Interposing this function
 97  * before main() allows us to call it in a machine-independent fashion.
 98  */
 99 void
100 mlsetup(struct regs *rp)
101 {
102         u_longlong_t prop_value;
103         extern struct classfuncs sys_classfuncs;
104         extern disp_t cpu0_disp;
105         extern char t0stack[];
106         extern int post_fastreboot;
107         extern uint64_t plat_dr_options;

109         ASSERT_STACK_ALIGNED();

111         /*
112          * initialize cpu_self
113          */
114         cpu[0]->cpu_self = cpu[0];

116 #if defined(__xpv)
117         /*
118          * Point at the hypervisor's virtual cpu structure
119          */
120         cpu[0]->cpu_m.mcpu_vcpu_info = &HYPERVISOR_shared_info->vcpu_info[0];
121 #endif

123         /*
124          * check if we've got special bits to clear or set
125          * when checking cpu features
126          */

128         if (bootprop_getval("cpuid_feature_ecx_include", &prop_value) != 0)
129                 cpuid_feature_ecx_include = 0;
130         else
131                 cpuid_feature_ecx_include = (uint32_t)prop_value;

133         if (bootprop_getval("cpuid_feature_ecx_exclude", &prop_value) != 0)
134                 cpuid_feature_ecx_exclude = 0;
135         else
136                 cpuid_feature_ecx_exclude = (uint32_t)prop_value;

138         if (bootprop_getval("cpuid_feature_edx_include", &prop_value) != 0)
139                 cpuid_feature_edx_include = 0;
140         else
141                 cpuid_feature_edx_include = (uint32_t)prop_value;

143         if (bootprop_getval("cpuid_feature_edx_exclude", &prop_value) != 0)
144                 cpuid_feature_edx_exclude = 0;
145         else
146                 cpuid_feature_edx_exclude = (uint32_t)prop_value;
```



```
148         /*
149          * Initialize idt0, gdt0, ldt0_default, ktss0 and dftss.
150          */
151         init_desctbls();

153         /*
154          * lgrp_init() and possibly cpuid_pass1() need PCI config
155          * space access
156          */
157 #if defined(__xpv)
158         if (DOMAIN_IS_INITDOMAIN(xen_info))
159                 pci_cfgspace_init();
160 #else
161         pci_cfgspace_init();
162         /*
163          * Initialize the platform type from CPU 0 to ensure that
164          * determine_platform() is only ever called once.
165          */
166         determine_platform();
167 #endif

169         /*
170          * The first lightweight pass (pass0) through the cpuid data
171          * was done in locore before mlsetup was called.  Do the next
172          * pass in C code.
173          *
174          * The x86_featureset is initialized here based on the capabilities
175          * of the boot CPU.  Note that if we choose to support CPUs that have
176          * different feature sets (at which point we would almost certainly
177          * want to set the feature bits to correspond to the feature
178          * minimum) this value may be altered.
179          */
180         cpuid_pass1(cpu[0], x86_featureset);

182 #if !defined(__xpv)
183         if ((get_hwenv() & HW_XEN_HVM) != 0)
184                 xen_hvm_init();

186         /*
187          * Before we do anything with the TSCs, we need to work around
188          * Intel erratum BT81.  On some CPUs, warm reset does not
189          * clear the TSC.  If we are on such a CPU, we will clear TSC ourselves
190          * here.  Other CPUs will clear it when we boot them later, and the
191          * resulting skew will be handled by tsc_sync_master()/_slave();
192          * note that such skew already exists and has to be handled anyway.
193          *
194          * We do this only on metal.  This same problem can occur with a
195          * hypervisor that does not happen to virtualise a TSC that starts from
196          * zero, regardless of CPU type; however, we do not expect hypervisors
197          * that do not virtualise TSC that way to handle writes to TSC
198          * correctly, either.
199          */
200         if (get_hwenv() == HW_NATIVE &&
201             cpuid_getvendor(CPU) == X86_VENDOR_Intel &&
202             cpuid_getfamily(CPU) == 6 &&
203             (cpuid_getmodel(CPU) == 0x2d || cpuid_getmodel(CPU) == 0x3e) &&
204             is_x86_feature(x86_featureset, X86FSET_TSC)) {
205                 (void) wrmsr(REG_TSC, 0UL);
206         }

208         /*
209          * Patch the tsc_read routine with appropriate set of instructions,
210          * depending on the processor family and architecure, to read the
211          * time-stamp counter while ensuring no out-of-order execution.
212          * Patch it while the kernel text is still writable.
```

```
213          *
214          * Note: tsc_read is not patched for intel processors whose family
215          * is >6 and for amd whose family >f (in case they don't support rdtscp
216          * instruction, unlikely). By default tsc_read will use cpuid for
217          * serialization in such cases. The following code needs to be
218          * revisited if intel processors of family >= f retains the
219          * instruction serialization nature of mfence instruction.
220          * Note: tsc_read is not patched for x86 processors which do
221          * not support "mfence". By default tsc_read will use cpuid for
222          * serialization in such cases.
223          *
224          * The Xen hypervisor does not correctly report whether rdtscp is
225          * supported or not, so we must assume that it is not.
226          */
227         if ((get_hwenv() & HW_XEN_HVM) == 0 &&
228             is_x86_feature(x86_featureset, X86FSET_TSCP))
229                 patch_tsc_read(X86_HAVE_TSCP);
230         else if (cpuid_getvendor(CPU) == X86_VENDOR_AMD &&
231             cpuid_getfamily(CPU) <= 0xf &&
232             is_x86_feature(x86_featureset, X86FSET_SSE2))
233                 patch_tsc_read(X86_TSC_MFENCE);
234         else if (cpuid_getvendor(CPU) == X86_VENDOR_Intel &&
235             cpuid_getfamily(CPU) <= 6 &&
236             is_x86_feature(x86_featureset, X86FSET_SSE2))
237                 patch_tsc_read(X86_TSC_LFENCE);

239 #endif  /* !__xpv */

241 #if defined(__i386) && !defined(__xpv)
242         /*
243          * Some i386 processors do not implement the rdtsc instruction,
244          * or at least they do not implement it correctly. Patch them to
245          * return 0.
246          */
247         if (!is_x86_feature(x86_featureset, X86FSET_TSC))
248                 patch_tsc_read(X86_NO_TSC);
249 #endif  /* __i386 && !__xpv */

251 #if defined(__amd64) && !defined(__xpv)
252         patch_memops(cpuid_getvendor(CPU));
253 #endif  /* __amd64 && !__xpv */

255 #if !defined(__xpv)
256         /* XXPV what, if anything, should be dorked with here under xen? */

258         /*
259          * While we're thinking about the TSC, let's set up %cr4 so that
260          * userland can issue rdtsc, and initialize the TSC_AUX value
261          * (the cpuid) for the rdtscp instruction on appropriately
262          * capable hardware.
263          */
264         if (is_x86_feature(x86_featureset, X86FSET_TSC))
265                 setcr4(getcr4() & ~CR4_TSD);

267         if (is_x86_feature(x86_featureset, X86FSET_TSCP))
268                 (void) wrmsr(MSR_AMD_TSCAUX, 0);

270         if (is_x86_feature(x86_featureset, X86FSET_DE))
271                 setcr4(getcr4() | CR4_DE);
272 #endif /* __xpv */

274         /*
275          * initialize t0
276          */
277         t0.t_stk = (caddr_t)rp - MINFRAME;
278         t0.t_stkbase = t0stack;
```



```
279         t0.t_pri = maxclsyspri - 3;
280         t0.t_schedflag = 0;
280         t0.t_schedflag = TS_LOAD | TS_DONT_SWAP;
281         t0.t_procp = &p0;
282         t0.t_plockp = &p0lock.pl_lock;
283         t0.t_lwp = &lwp0;
284         t0.t_forw = &t0;
285         t0.t_back = &t0;
286         t0.t_next = &t0;
287         t0.t_prev = &t0;
288         t0.t_cpu = cpu[0];
289         t0.t_disp_queue = &cpu0_disp;
290         t0.t_bind_cpu = PBIND_NONE;
291         t0.t_bind_pset = PS_NONE;
292         t0.t_bindflag = (uchar_t)default_binding_mode;
293         t0.t_cpupart = &cp_default;
294         t0.t_clfuncs = &sys_classfuncs.thread;
295         t0.t_copyops = NULL;
296         THREAD_ONPROC(&t0, CPU);

298         lwp0.lwp_thread = &t0;
299         lwp0.lwp_regs = (void *)rp;
300         lwp0.lwp_procp = &p0;
301         t0.t_tid = p0.p_lwpcnt = p0.p_lwprcnt = p0.p_lwpid = 1;

303         p0.p_exec = NULL;
304         p0.p_stat = SRUN;
305         p0.p_flag = SSYS;
306         p0.p_tlist = &t0;
307         p0.p_stksize = 2*PAGESIZE;
308         p0.p_stkpageszc = 0;
309         p0.p_as = &kas;
310         p0.p_lockp = &p0lock;
311         p0.p_brkpageszc = 0;
312         p0.p_t1_lgrpid = LGRP_NONE;
313         p0.p_tr_lgrpid = LGRP_NONE;
314         sigorset(&p0.p_ignore, &ignoredefault);

316         CPU->cpu_thread = &t0;
317         bzero(&cpu0_disp, sizeof (disp_t));
318         CPU->cpu_disp = &cpu0_disp;
319         CPU->cpu_disp->disp_cpu = CPU;
320         CPU->cpu_dispthread = &t0;
321         CPU->cpu_idle_thread = &t0;
322         CPU->cpu_flags = CPU_READY | CPU_RUNNING | CPU_EXISTS | CPU_ENABLE;
323         CPU->cpu_dispatch_pri = t0.t_pri;

325         CPU->cpu_id = 0;

327         CPU->cpu_pri = 12;              /* initial PIL for the boot CPU */

329         /*
330          * The kernel doesn't use LDTs unless a process explicitly requests one.
331          */
332         p0.p_ldt_desc = null_sdesc;

334         /*
335          * Initialize thread/cpu microstate accounting
336          */
337         init_mstate(&t0, LMS_SYSTEM);
338         init_cpu_mstate(CPU, CMS_SYSTEM);

340         /*
341          * Initialize lists of available and active CPUs.
342          */
343         cpu_list_init(CPU);
```

```
345         pg_cpu_bootstrap(CPU);

347         /*
348          * Now that we have taken over the GDT, IDT and have initialized
349          * active CPU list it's time to inform kmdb if present.
350          */
351         if (boothowto & RB_DEBUG)
352                 kdi_idt_sync();

354         /*
355          * Explicitly set console to text mode (0x3) if this is a boot
356          * post Fast Reboot, and the console is set to CONS_SCREEN_TEXT.
357          */
358         if (post_fastreboot && boot_console_type(NULL) == CONS_SCREEN_TEXT)
359                 set_console_mode(0x3);

361         /*
362          * If requested (boot -d) drop into kmdb.
363          *
364          * This must be done after cpu_list_init() on the 64-bit kernel
365          * since taking a trap requires that we re-compute gsbase based
366          * on the cpu list.
367          */
368         if (boothowto & RB_DEBUGENTER)
369                 kmdb_enter();

371         cpu_vm_data_init(CPU);

373         rp->r_fp = 0;   /* terminate kernel stack traces! */

375         prom_init("kernel", (void *)NULL);

377         /* User-set option overrides firmware value. */
378         if (bootprop_getval(PLAT_DR_OPTIONS_NAME, &prop_value) == 0) {
379                 plat_dr_options = (uint64_t)prop_value;
380         }
381 #if defined(__xpv)
382         /* No support of DR operations on xpv */
383         plat_dr_options = 0;
384 #else   /* __xpv */
385         /* Flag PLAT_DR_FEATURE_ENABLED should only be set by DR driver. */
386         plat_dr_options &= ~PLAT_DR_FEATURE_ENABLED;
387 #ifndef __amd64
388         /* Only enable CPU/memory DR on 64 bits kernel. */
389         plat_dr_options &= ~PLAT_DR_FEATURE_MEMORY;
390         plat_dr_options &= ~PLAT_DR_FEATURE_CPU;
391 #endif  /* __amd64 */
392 #endif  /* __xpv */

394         /*
395          * Get value of "plat_dr_physmax" boot option.
396          * It overrides values calculated from MSCT or SRAT table.
397          */
398         if (bootprop_getval(PLAT_DR_PHYSMAX_NAME, &prop_value) == 0) {
399                 plat_dr_physmax = ((uint64_t)prop_value) >> PAGESHIFT;
400         }

402         /* Get value of boot_ncpus. */
403         if (bootprop_getval(BOOT_NCPUS_NAME, &prop_value) != 0) {
404                 boot_ncpus = NCPU;
405         } else {
406                 boot_ncpus = (int)prop_value;
407                 if (boot_ncpus <= 0 || boot_ncpus > NCPU)
408                         boot_ncpus = NCPU;
409         }
```



```
411         /*
412          * Set max_ncpus and boot_max_ncpus to boot_ncpus if platform doesn't
413          * support CPU DR operations.
414          */
415         if (plat_dr_support_cpu() == 0) {
416                 max_ncpus = boot_max_ncpus = boot_ncpus;
417         } else {
418                 if (bootprop_getval(PLAT_MAX_NCPUS_NAME, &prop_value) != 0) {
419                         max_ncpus = NCPU;
420                 } else {
421                         max_ncpus = (int)prop_value;
422                         if (max_ncpus <= 0 || max_ncpus > NCPU) {
423                                 max_ncpus = NCPU;
424                         }
425                         if (boot_ncpus > max_ncpus) {
426                                 boot_ncpus = max_ncpus;
427                         }
428                 }

430                 if (bootprop_getval(BOOT_MAX_NCPUS_NAME, &prop_value) != 0) {
431                         boot_max_ncpus = boot_ncpus;
432                 } else {
433                         boot_max_ncpus = (int)prop_value;
434                         if (boot_max_ncpus <= 0 || boot_max_ncpus > NCPU) {
435                                 boot_max_ncpus = boot_ncpus;
436                         } else if (boot_max_ncpus > max_ncpus) {
437                                 boot_max_ncpus = max_ncpus;
438                         }
439                 }
440         }

442         /*
443          * Initialize the lgrp framework
444          */
445         lgrp_init(LGRP_INIT_STAGE1);

447         if (boothowto & RB_HALT) {
448                 prom_printf("unix: kernel halted by -h flag\n");
449                 prom_enter_mon();
450         }

452         ASSERT_STACK_ALIGNED();

454         /*
455          * Fill out cpu_ucode_info.  Update microcode if necessary.
456          */
457         ucode_check(CPU);

459         if (workaround_errata(CPU) != 0)
460                 panic("critical workaround(s) missing for boot cpu");
461 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   61422 Fri May  8 18:10:39 2015
new/usr/src/uts/i86pc/os/trap.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

 453 #endif /* OPTERON_ERRATUM_91 */

 455 /*
 456  * Called from the trap handler when a processor trap occurs.
 457  *
 458  * Note: All user-level traps that might call stop() must exit
 459  * trap() by 'goto out' or by falling through.
 460  * Note Also: trap() is usually called with interrupts enabled, (PS_IE == 1)
 461  * however, there are paths that arrive here with PS_IE == 0 so special care
 462  * must be taken in those cases.
 463  */
 464 void
 465 trap(struct regs *rp, caddr_t addr, processorid_t cpuid)
 466 {
 467         kthread_t *ct = curthread;
 468         enum seg_rw rw;
 469         unsigned type;
 470         proc_t *p = ttoproc(ct);
 471         klwp_t *lwp = ttolwp(ct);
 472         uintptr_t lofault;
 473         label_t *onfault;
 474         faultcode_t pagefault(), res, errcode;
 475         enum fault_type fault_type;
 476         k_siginfo_t siginfo;
 477         uint_t fault = 0;
 478         int mstate;
 479         int sicode = 0;
 480         int watchcode;
 481         int watchpage;
 482         caddr_t vaddr;
 483         int singlestep_twiddle;
 484         size_t sz;
 485         int ta;
 486 #ifdef __amd64
 487         uchar_t instr;
 488 #endif

 490         ASSERT_STACK_ALIGNED();

 492         type = rp->r_trapno;
 493         CPU_STATS_ADDQ(CPU, sys, trap, 1);
 494         ASSERT(ct->t_schedflag & TS_DONT_SWAP);

 495         if (type == T_PGFLT) {

 497                 errcode = rp->r_err;
 498                 if (errcode & PF_ERR_WRITE)
 499                         rw = S_WRITE;
 500                 else if ((caddr_t)rp->r_pc == addr ||
 501                     (mmu.pt_nx != 0 && (errcode & PF_ERR_EXEC)))
 502                         rw = S_EXEC;
 503                 else
 504                         rw = S_READ;
```



```
 506 #if defined(__i386)
 507                 /*
 508                  * Pentium Pro work-around
 509                  */
 510                 if ((errcode & PF_ERR_PROT) && pentiumpro_bug4046376) {
 511                         uint_t  attr;
 512                         uint_t  priv_violation;
 513                         uint_t  access_violation;

 515                         if (hat_getattr(addr < (caddr_t)kernelbase ?
 516                             curproc->p_as->a_hat : kas.a_hat, addr, &attr)
 517                             == -1) {
 518                                 errcode &= ~PF_ERR_PROT;
 519                         } else {
 520                                 priv_violation = (errcode & PF_ERR_USER) &&
 521                                     !(attr & PROT_USER);
 522                                 access_violation = (errcode & PF_ERR_WRITE) &&
 523                                     !(attr & PROT_WRITE);
 524                                 if (!priv_violation && !access_violation)
 525                                         goto cleanup;
 526                         }
 527                 }
 528 #endif /* __i386 */

 530         } else if (type == T_SGLSTP && lwp != NULL)
 531                 lwp->lwp_pcb.pcb_drstat = (uintptr_t)addr;

 533         if (tdebug)
 534                 showregs(type, rp, addr);

 536         if (USERMODE(rp->r_cs)) {
 537                 /*
 538                  * Set up the current cred to use during this trap. u_cred
 539                  * no longer exists.  t_cred is used instead.
 540                  * The current process credential applies to the thread for
 541                  * the entire trap.  If trapping from the kernel, this
 542                  * should already be set up.
 543                  */
 544                 if (ct->t_cred != p->p_cred) {
 545                         cred_t *oldcred = ct->t_cred;
 546                         /*
 547                          * DTrace accesses t_cred in probe context.  t_cred
 548                          * must always be either NULL, or point to a valid,
 549                          * allocated cred structure.
 550                          */
 551                         ct->t_cred = crgetcred();
 552                         crfree(oldcred);
 553                 }
 554                 ASSERT(lwp != NULL);
 555                 type |= USER;
 556                 ASSERT(lwptoregs(lwp) == rp);
 557                 lwp->lwp_state = LWP_SYS;

 559                 switch (type) {
 560                 case T_PGFLT + USER:
 561                         if ((caddr_t)rp->r_pc == addr)
 562                                 mstate = LMS_TFAULT;
 563                         else
 564                                 mstate = LMS_DFAULT;
 565                         break;
 566                 default:
 567                         mstate = LMS_TRAP;
 568                         break;
 569                 }
 570                 /* Kernel probe */
```

```
571                 TNF_PROBE_1(thread_state, "thread", /* CSTYLED */,
572                     tnf_microstate, state, mstate);
573                 mstate = new_mstate(ct, mstate);

575                 bzero(&siginfo, sizeof (siginfo));
576         }

578         switch (type) {
579         case T_PGFLT + USER:
580         case T_SGLSTP:
581         case T_SGLSTP + USER:
582         case T_BPTFLT + USER:
583                 break;

585         default:
586                 FTRACE_2("trap(): type=0x%lx, regs=0x%lx",
587                     (ulong_t)type, (ulong_t)rp);
588                 break;
589         }

591         switch (type) {
592         case T_SIMDFPE:
593                 /* Make sure we enable interrupts before die()ing */
594                 sti();  /* The SIMD exception comes in via cmninttrap */
595                 /*FALLTHROUGH*/
596         default:
597                 if (type & USER) {
598                         if (tudebug)
599                                 showregs(type, rp, (caddr_t)0);
600                         printf("trap: Unknown trap type %d in user mode\n",
601                             type & ~USER);
602                         siginfo.si_signo = SIGILL;
603                         siginfo.si_code  = ILL_ILLTRP;
604                         siginfo.si_addr  = (caddr_t)rp->r_pc;
605                         siginfo.si_trapno = type & ~USER;
606                         fault = FLTILL;
607                         break;
608                 } else {
609                         (void) die(type, rp, addr, cpuid);
610                         /*NOTREACHED*/
611                 }

613         case T_PGFLT:           /* system page fault */
614                 /*
615                  * If we're under on_trap() protection (see <sys/ontrap.h>),
616                  * set ot_trap and bounce back to the on_trap() call site
617                  * via the installed trampoline.
618                  */
619                 if ((ct->t_ontrap != NULL) &&
620                     (ct->t_ontrap->ot_prot & OT_DATA_ACCESS)) {
621                         ct->t_ontrap->ot_trap |= OT_DATA_ACCESS;
622                         rp->r_pc = ct->t_ontrap->ot_trampoline;
623                         goto cleanup;
624                 }

626                 /*
627                  * See if we can handle as pagefault. Save lofault and onfault
628                  * across this. Here we assume that an address less than
629                  * KERNELBASE is a user fault.  We can do this as copy.s
630                  * routines verify that the starting address is less than
631                  * KERNELBASE before starting and because we know that we
632                  * always have KERNELBASE mapped as invalid to serve as a
633                  * "barrier".
634                  */
635                 lofault = ct->t_lofault;
636                 onfault = ct->t_onfault;
```



```
637                 ct->t_lofault = 0;

639                 mstate = new_mstate(ct, LMS_KFAULT);

641                 if (addr < (caddr_t)kernelbase) {
642                         res = pagefault(addr,
643                             (errcode & PF_ERR_PROT)? F_PROT: F_INVAL, rw, 0);
644                         if (res == FC_NOMAP &&
645                             addr < p->p_usrstack &&
646                             grow(addr))
647                                 res = 0;
648                 } else {
649                         res = pagefault(addr,
650                             (errcode & PF_ERR_PROT)? F_PROT: F_INVAL, rw, 1);
651                 }
652                 (void) new_mstate(ct, mstate);

654                 /*
655                  * Restore lofault and onfault. If we resolved the fault, exit.
656                  * If we didn't and lofault wasn't set, die.
657                  */
658                 ct->t_lofault = lofault;
659                 ct->t_onfault = onfault;
660                 if (res == 0)
661                         goto cleanup;

663 #if defined(OPTERON_ERRATUM_93) && defined(_LP64)
664                 if (lofault == 0 && opteron_erratum_93) {
665                         /*
666                          * Workaround for Opteron Erratum 93. On return from
667                          * a System Managment Interrupt at a HLT instruction
668                          * the %rip might be truncated to a 32 bit value.
669                          * BIOS is supposed to fix this, but some don't.
670                          * If this occurs we simply restore the high order bits.
671                          * The HLT instruction is 1 byte of 0xf4.
672                          */
673                         uintptr_t       rip = rp->r_pc;

675                         if ((rip & 0xffffffffUL) == rip) {
676                                 rip |= 0xffffffffUL << 32;
677                                 if (hat_getpfnum(kas.a_hat, (caddr_t)rip) !=
678                                     PFN_INVALID &&
679                                     (*(uchar_t *)rip == 0xf4 ||
680                                     *(uchar_t *)(rip - 1) == 0xf4)) {
681                                         rp->r_pc = rip;
682                                         goto cleanup;
683                                 }
684                         }
685                 }
686 #endif /* OPTERON_ERRATUM_93 && _LP64 */

688 #ifdef OPTERON_ERRATUM_91
689                 if (lofault == 0 && opteron_erratum_91) {
690                         /*
691                          * Workaround for Opteron Erratum 91. Prefetches may
692                          * generate a page fault (they're not supposed to do
693                          * that!). If this occurs we simply return back to the
694                          * instruction.
695                          */
696                         caddr_t         pc = (caddr_t)rp->r_pc;

698                         /*
699                          * If the faulting PC is not mapped, this is a
700                          * legitimate kernel page fault that must result in a
701                          * panic. If the faulting PC is mapped, it could contain
702                          * a prefetch instruction. Check for that here.
```

```
703                          */
704                         if (hat_getpfnum(kas.a_hat, pc) != PFN_INVALID) {
705                                 if (cmp_to_prefetch((uchar_t *)pc)) {
706 #ifdef DEBUG
707                                         cmn_err(CE_WARN, "Opteron erratum 91 "
708                                             "occurred: kernel prefetch"
709                                             " at %p generated a page fault!",
710                                             (void *)rp->r_pc);
711 #endif /* DEBUG */
712                                         goto cleanup;
713                                 }
714                         }
715                         (void) die(type, rp, addr, cpuid);
716                 }
717 #endif /* OPTERON_ERRATUM_91 */

719                 if (lofault == 0)
720                         (void) die(type, rp, addr, cpuid);

722                 /*
723                  * Cannot resolve fault.  Return to lofault.
724                  */
725                 if (lodebug) {
726                         showregs(type, rp, addr);
727                         traceregs(rp);
728                 }
729                 if (FC_CODE(res) == FC_OBJERR)
730                         res = FC_ERRNO(res);
731                 else
732                         res = EFAULT;
733                 rp->r_r0 = res;
734                 rp->r_pc = ct->t_lofault;
735                 goto cleanup;

737         case T_PGFLT + USER:    /* user page fault */
738                 if (faultdebug) {
739                         char *fault_str;

741                         switch (rw) {
742                         case S_READ:
743                                 fault_str = "read";
744                                 break;
745                         case S_WRITE:
746                                 fault_str = "write";
747                                 break;
748                         case S_EXEC:
749                                 fault_str = "exec";
750                                 break;
751                         default:
752                                 fault_str = "";
753                                 break;
754                         }
755                         printf("user %s fault:  addr=0x%lx errcode=0x%x\n",
756                             fault_str, (uintptr_t)addr, errcode);
757                 }

759 #if defined(OPTERON_ERRATUM_100) && defined(_LP64)
760                 /*
761                  * Workaround for AMD erratum 100
762                  *
763                  * A 32-bit process may receive a page fault on a non
764                  * 32-bit address by mistake. The range of the faulting
765                  * address will be
766                  *
767                  *      0xffffffff80000000 .. 0xffffffffffffffff or
768                  *      0x0000000100000000 .. 0x000000017fffffff
```



```
769                  *
770                  * The fault is always due to an instruction fetch, however
771                  * the value of r_pc should be correct (in 32 bit range),
772                  * so we ignore the page fault on the bogus address.
773                  */
774                 if (p->p_model == DATAMODEL_ILP32 &&
775                     (0xffffffff80000000 <= (uintptr_t)addr ||
776                     (0x100000000 <= (uintptr_t)addr &&
777                     (uintptr_t)addr <= 0x17fffffff))) {
778                         if (!opteron_erratum_100)
779                                 panic("unexpected erratum #100");
780                         if (rp->r_pc <= 0xffffffff)
781                                 goto out;
782                 }
783 #endif /* OPTERON_ERRATUM_100 && _LP64 */

785                 ASSERT(!(curthread->t_flag & T_WATCHPT));
786                 watchpage = (pr_watch_active(p) && pr_is_watchpage(addr, rw));
787 #ifdef __i386
788                 /*
789                  * In 32-bit mode, the lcall (system call) instruction fetches
790                  * one word from the stack, at the stack pointer, because of the
791                  * way the call gate is constructed.  This is a bogus
792                  * read and should not be counted as a read watchpoint.
793                  * We work around the problem here by testing to see if
794                  * this situation applies and, if so, simply jumping to
795                  * the code in locore.s that fields the system call trap.
796                  * The registers on the stack are already set up properly
797                  * due to the match between the call gate sequence and the
798                  * trap gate sequence.  We just have to adjust the pc.
799                  */
800                 if (watchpage && addr == (caddr_t)rp->r_sp &&
801                     rw == S_READ && instr_is_lcall_syscall((caddr_t)rp->r_pc)) {
802                         extern void watch_syscall(void);

804                         rp->r_pc += LCALLSIZE;
805                         watch_syscall();        /* never returns */
806                         /* NOTREACHED */
807                 }
808 #endif /* __i386 */
809                 vaddr = addr;
810                 if (!watchpage || (sz = instr_size(rp, &vaddr, rw)) <= 0)
811                         fault_type = (errcode & PF_ERR_PROT)? F_PROT: F_INVAL;
812                 else if ((watchcode = pr_is_watchpoint(&vaddr, &ta,
813                     sz, NULL, rw)) != 0) {
814                         if (ta) {
815                                 do_watch_step(vaddr, sz, rw,
816                                     watchcode, rp->r_pc);
817                                 fault_type = F_INVAL;
818                         } else {
819                                 bzero(&siginfo, sizeof (siginfo));
820                                 siginfo.si_signo = SIGTRAP;
821                                 siginfo.si_code = watchcode;
822                                 siginfo.si_addr = vaddr;
823                                 siginfo.si_trapafter = 0;
824                                 siginfo.si_pc = (caddr_t)rp->r_pc;
825                                 fault = FLTWATCH;
826                                 break;
827                         }
828                 } else {
829                         /* XXX pr_watch_emul() never succeeds (for now) */
830                         if (rw != S_EXEC && pr_watch_emul(rp, vaddr, rw))
831                                 goto out;
832                         do_watch_step(vaddr, sz, rw, 0, 0);
833                         fault_type = F_INVAL;
834                 }
```

```
836                 res = pagefault(addr, fault_type, rw, 0);
837
838                 /*
839                  * If pagefault() succeeded, ok.
840                  * Otherwise attempt to grow the stack.
841                  */
842                 if (res == 0 ||
843                     (res == FC_NOMAP &&
844                     addr < p->p_usrstack &&
845                     grow(addr))) {
846                         lwp->lwp_lastfault = FLTPAGE;
847                         lwp->lwp_lastfaddr = addr;
848                         if (prismember(&p->p_fltmask, FLTPAGE)) {
849                                 bzero(&siginfo, sizeof (siginfo));
850                                 siginfo.si_addr = addr;
851                                 (void) stop_on_fault(FLTPAGE, &siginfo);
852                         }
853                         goto out;
854                 } else if (res == FC_PROT && addr < p->p_usrstack &&
855                     (mmu.pt_nx != 0 && (errcode & PF_ERR_EXEC))) {
856                         report_stack_exec(p, addr);
857                 }
858
859 #ifdef OPTERON_ERRATUM_91
860                 /*
861                  * Workaround for Opteron Erratum 91. Prefetches may generate a
862                  * page fault (they're not supposed to do that!). If this
863                  * occurs we simply return back to the instruction.
864                  *
865                  * We rely on copyin to properly fault in the page with r_pc.
866                  */
867                 if (opteron_erratum_91 &&
868                     addr != (caddr_t)rp->r_pc &&
869                     instr_is_prefetch((caddr_t)rp->r_pc)) {
870 #ifdef DEBUG
871                         cmn_err(CE_WARN, "Opteron erratum 91 occurred: "
872                             "prefetch at %p in pid %d generated a trap!",
873                             (void *)rp->r_pc, p->p_pid);
874 #endif /* DEBUG */
875                         goto out;
876                 }
877 #endif /* OPTERON_ERRATUM_91 */
878
879                 if (tudebug)
880                         showregs(type, rp, addr);
881                 /*
882                  * In the case where both pagefault and grow fail,
883                  * set the code to the value provided by pagefault.
884                  * We map all errors returned from pagefault() to SIGSEGV.
885                  */
886                 bzero(&siginfo, sizeof (siginfo));
887                 siginfo.si_addr = addr;
888                 switch (FC_CODE(res)) {
889                 case FC_HWERR:
890                 case FC_NOSUPPORT:
891                         siginfo.si_signo = SIGBUS;
892                         siginfo.si_code = BUS_ADRERR;
893                         fault = FLTACCESS;
894                         break;
895                 case FC_ALIGN:
896                         siginfo.si_signo = SIGBUS;
897                         siginfo.si_code = BUS_ADRALN;
898                         fault = FLTACCESS;
899                         break;
900                 case FC_OBJERR:
```



```
901                         if ((siginfo.si_errno = FC_ERRNO(res)) != EINTR) {
902                                 siginfo.si_signo = SIGBUS;
903                                 siginfo.si_code = BUS_OBJERR;
904                                 fault = FLTACCESS;
905                         }
906                         break;
907                 default:        /* FC_NOMAP or FC_PROT */
908                         siginfo.si_signo = SIGSEGV;
909                         siginfo.si_code =
910                             (res == FC_NOMAP)? SEGV_MAPERR : SEGV_ACCERR;
911                         fault = FLTBOUNDS;
912                         break;
913                 }
914                 break;
915
916         case T_ILLINST + USER:  /* invalid opcode fault */
917                 /*
918                  * If the syscall instruction is disabled due to LDT usage, a
919                  * user program that attempts to execute it will trigger a #ud
920                  * trap. Check for that case here. If this occurs on a CPU which
921                  * doesn't even support syscall, the result of all of this will
922                  * be to emulate that particular instruction.
923                  */
924                 if (p->p_ldt != NULL &&
925                     ldt_rewrite_syscall(rp, p, X86FSET_ASYSC))
926                         goto out;
927
928 #ifdef __amd64
929                 /*
930                  * Emulate the LAHF and SAHF instructions if needed.
931                  * See the instr_is_lsahf function for details.
932                  */
933                 if (p->p_model == DATAMODEL_LP64 &&
934                     instr_is_lsahf((caddr_t)rp->r_pc, &instr)) {
935                         emulate_lsahf(rp, instr);
936                         goto out;
937                 }
938 #endif
939
940                 /*FALLTHROUGH*/
941
942                 if (tudebug)
943                         showregs(type, rp, (caddr_t)0);
944                 siginfo.si_signo = SIGILL;
945                 siginfo.si_code  = ILL_ILLOPC;
946                 siginfo.si_addr  = (caddr_t)rp->r_pc;
947                 fault = FLTILL;
948                 break;
949
950         case T_ZERODIV + USER:          /* integer divide by zero */
951                 if (tudebug && tudebugfpe)
952                         showregs(type, rp, (caddr_t)0);
953                 siginfo.si_signo = SIGFPE;
954                 siginfo.si_code  = FPE_INTDIV;
955                 siginfo.si_addr  = (caddr_t)rp->r_pc;
956                 fault = FLTIZDIV;
957                 break;
958
959         case T_OVFLW + USER:    /* integer overflow */
960                 if (tudebug && tudebugfpe)
961                         showregs(type, rp, (caddr_t)0);
962                 siginfo.si_signo = SIGFPE;
963                 siginfo.si_code  = FPE_INTOVF;
964                 siginfo.si_addr  = (caddr_t)rp->r_pc;
965                 fault = FLTIOVF;
966                 break;
```

```
968         case T_NOEXTFLT + USER: /* math coprocessor not available */
969                 if (tudebug && tudebugfpe)
970                         showregs(type, rp, addr);
971                 if (fpnoextflt(rp)) {
972                         siginfo.si_signo = SIGILL;
973                         siginfo.si_code  = ILL_ILLOPC;
974                         siginfo.si_addr  = (caddr_t)rp->r_pc;
975                         fault = FLTILL;
976                 }
977                 break;

979         case T_EXTOVRFLT:       /* extension overrun fault */
980                 /* check if we took a kernel trap on behalf of user */
981                 {
982                         extern  void ndptrap_frstor(void);
983                         if (rp->r_pc != (uintptr_t)ndptrap_frstor) {
984                                 sti(); /* T_EXTOVRFLT comes in via cmninttrap */
985                                 (void) die(type, rp, addr, cpuid);
986                         }
987                         type |= USER;
988                 }
989                 /*FALLTHROUGH*/
990         case T_EXTOVRFLT + USER:        /* extension overrun fault */
991                 if (tudebug && tudebugfpe)
992                         showregs(type, rp, addr);
993                 if (fpextovrflt(rp)) {
994                         siginfo.si_signo = SIGSEGV;
995                         siginfo.si_code  = SEGV_MAPERR;
996                         siginfo.si_addr  = (caddr_t)rp->r_pc;
997                         fault = FLTBOUNDS;
998                 }
999                 break;

1001        case T_EXTERRFLT:       /* x87 floating point exception pending */
1002                /* check if we took a kernel trap on behalf of user */
1003                {
1004                        extern  void ndptrap_frstor(void);
1005                        if (rp->r_pc != (uintptr_t)ndptrap_frstor) {
1006                                sti(); /* T_EXTERRFLT comes in via cmninttrap */
1007                                (void) die(type, rp, addr, cpuid);
1008                        }
1009                        type |= USER;
1010                }
1011                /*FALLTHROUGH*/

1013        case T_EXTERRFLT + USER: /* x87 floating point exception pending */
1014                if (tudebug && tudebugfpe)
1015                        showregs(type, rp, addr);
1016                if (sicode = fpexterrflt(rp)) {
1017                        siginfo.si_signo = SIGFPE;
1018                        siginfo.si_code  = sicode;
1019                        siginfo.si_addr  = (caddr_t)rp->r_pc;
1020                        fault = FLTFPE;
1021                }
1022                break;

1024        case T_SIMDFPE + USER:          /* SSE and SSE2 exceptions */
1025                if (tudebug && tudebugsse)
1026                        showregs(type, rp, addr);
1027                if (!is_x86_feature(x86_featureset, X86FSET_SSE) &&
1028                    !is_x86_feature(x86_featureset, X86FSET_SSE2)) {
1029                        /*
1030                         * There are rumours that some user instructions
1031                         * on older CPUs can cause this trap to occur; in
1032                         * which case send a SIGILL instead of a SIGFPE.
```



```
1033                         */
1034                        siginfo.si_signo = SIGILL;
1035                        siginfo.si_code  = ILL_ILLTRP;
1036                        siginfo.si_addr  = (caddr_t)rp->r_pc;
1037                        siginfo.si_trapno = type & ~USER;
1038                        fault = FLTILL;
1039                } else if ((sicode = fpsimderrflt(rp)) != 0) {
1040                        siginfo.si_signo = SIGFPE;
1041                        siginfo.si_code = sicode;
1042                        siginfo.si_addr = (caddr_t)rp->r_pc;
1043                        fault = FLTFPE;
1044                }

1046                sti();  /* The SIMD exception comes in via cmninttrap */
1047                break;

1049        case T_BPTFLT:  /* breakpoint trap */
1050                /*
1051                 * Kernel breakpoint traps should only happen when kmdb is
1052                 * active, and even then, it'll have interposed on the IDT, so
1053                 * control won't get here.  If it does, we've hit a breakpoint
1054                 * without the debugger, which is very strange, and very
1055                 * fatal.
1056                 */
1057                if (tudebug && tudebugbpt)
1058                        showregs(type, rp, (caddr_t)0);

1060                (void) die(type, rp, addr, cpuid);
1061                break;

1063        case T_SGLSTP: /* single step/hw breakpoint exception */

1065                /* Now evaluate how we got here */
1066                if (lwp != NULL && (lwp->lwp_pcb.pcb_drstat & DR_SINGLESTEP)) {
1067                        /*
1068                         * i386 single-steps even through lcalls which
1069                         * change the privilege level. So we take a trap at
1070                         * the first instruction in privileged mode.
1071                         *
1072                         * Set a flag to indicate that upon completion of
1073                         * the system call, deal with the single-step trap.
1074                         *
1075                         * The same thing happens for sysenter, too.
1076                         */
1077                        singlestep_twiddle = 0;
1078                        if (rp->r_pc == (uintptr_t)sys_sysenter ||
1079                            rp->r_pc == (uintptr_t)brand_sys_sysenter) {
1080                                singlestep_twiddle = 1;
1081 #if defined(__amd64)
1082                                /*
1083                                 * Since we are already on the kernel's
1084                                 * %gs, on 64-bit systems the sysenter case
1085                                 * needs to adjust the pc to avoid
1086                                 * executing the swapgs instruction at the
1087                                 * top of the handler.
1088                                 */
1089                                if (rp->r_pc == (uintptr_t)sys_sysenter)
1090                                        rp->r_pc = (uintptr_t)
1091                                            _sys_sysenter_post_swapgs;
1092                                else
1093                                        rp->r_pc = (uintptr_t)
1094                                            _brand_sys_sysenter_post_swapgs;
1095 #endif
1096                        }
1097 #if defined(__i386)
1098                        else if (rp->r_pc == (uintptr_t)sys_call ||
```

```
1099                            rp->r_pc == (uintptr_t)brand_sys_call) {
1100                                singlestep_twiddle = 1;
1101                        }
1102 #endif
1103                        else {
1104                                /* not on sysenter/syscall; uregs available */
1105                                if (tudebug && tudebugbpt)
1106                                        showregs(type, rp, (caddr_t)0);
1107                        }
1108                        if (singlestep_twiddle) {
1109                                rp->r_ps &= ~PS_T; /* turn off trace */
1110                                lwp->lwp_pcb.pcb_flags |= DEBUG_PENDING;
1111                                ct->t_post_sys = 1;
1112                                aston(curthread);
1113                                goto cleanup;
1114                        }
1115                }
1116                /* XXX - needs review on debugger interface? */
1117                if (boothowto & RB_DEBUG)
1118                        debug_enter((char *)NULL);
1119                else
1120                        (void) die(type, rp, addr, cpuid);
1121                break;

1123        case T_NMIFLT:  /* NMI interrupt */
1124                printf("Unexpected NMI in system mode\n");
1125                goto cleanup;

1127        case T_NMIFLT + USER:   /* NMI interrupt */
1128                printf("Unexpected NMI in user mode\n");
1129                break;

1131        case T_GPFLT:   /* general protection violation */
1132                /*
1133                 * Any #GP that occurs during an on_trap .. no_trap bracket
1134                 * with OT_DATA_ACCESS or OT_SEGMENT_ACCESS protection,
1135                 * or in a on_fault .. no_fault bracket, is forgiven
1136                 * and we trampoline.  This protection is given regardless
1137                 * of whether we are 32/64 bit etc - if a distinction is
1138                 * required then define new on_trap protection types.
1139                 *
1140                 * On amd64, we can get a #gp from referencing addresses
1141                 * in the virtual address hole e.g. from a copyin or in
1142                 * update_sregs while updating user segment registers.
1143                 *
1144                 * On the 32-bit hypervisor we could also generate one in
1145                 * mfn_to_pfn by reaching around or into where the hypervisor
1146                 * lives which is protected by segmentation.
1147                 */

1149                /*
1150                 * If we're under on_trap() protection (see <sys/ontrap.h>),
1151                 * set ot_trap and trampoline back to the on_trap() call site
1152                 * for OT_DATA_ACCESS or OT_SEGMENT_ACCESS.
1153                 */
1154                if (ct->t_ontrap != NULL) {
1155                        int ttype =  ct->t_ontrap->ot_prot &
1156                            (OT_DATA_ACCESS | OT_SEGMENT_ACCESS);

1158                        if (ttype != 0) {
1159                                ct->t_ontrap->ot_trap |= ttype;
1160                                if (tudebug)
1161                                        showregs(type, rp, (caddr_t)0);
1162                                rp->r_pc = ct->t_ontrap->ot_trampoline;
1163                                goto cleanup;
1164                        }
```



```
1165                }

1167                /*
1168                 * If we're under lofault protection (copyin etc.),
1169                 * longjmp back to lofault with an EFAULT.
1170                 */
1171                if (ct->t_lofault) {
1172                        /*
1173                         * Fault is not resolvable, so just return to lofault
1174                         */
1175                        if (lodebug) {
1176                                showregs(type, rp, addr);
1177                                traceregs(rp);
1178                        }
1179                        rp->r_r0 = EFAULT;
1180                        rp->r_pc = ct->t_lofault;
1181                        goto cleanup;
1182                }

1184                /*
1185                 * We fall through to the next case, which repeats
1186                 * the OT_SEGMENT_ACCESS check which we've already
1187                 * done, so we'll always fall through to the
1188                 * T_STKFLT case.
1189                 */
1190                /*FALLTHROUGH*/
1191        case T_SEGFLT:  /* segment not present fault */
1192                /*
1193                 * One example of this is #NP in update_sregs while
1194                 * attempting to update a user segment register
1195                 * that points to a descriptor that is marked not
1196                 * present.
1197                 */
1198                if (ct->t_ontrap != NULL &&
1199                    ct->t_ontrap->ot_prot & OT_SEGMENT_ACCESS) {
1200                        ct->t_ontrap->ot_trap |= OT_SEGMENT_ACCESS;
1201                        if (tudebug)
1202                                showregs(type, rp, (caddr_t)0);
1203                        rp->r_pc = ct->t_ontrap->ot_trampoline;
1204                        goto cleanup;
1205                }
1206                /*FALLTHROUGH*/
1207        case T_STKFLT:  /* stack fault */
1208        case T_TSSFLT:  /* invalid TSS fault */
1209                if (tudebug)
1210                        showregs(type, rp, (caddr_t)0);
1211                if (kern_gpfault(rp))
1212                        (void) die(type, rp, addr, cpuid);
1213                goto cleanup;

1215        /*
1216         * ONLY 32-bit PROCESSES can USE a PRIVATE LDT! 64-bit apps
1217         * should have no need for them, so we put a stop to it here.
1218         *
1219         * So: not-present fault is ONLY valid for 32-bit processes with
1220         * a private LDT trying to do a system call. Emulate it.
1221         *
1222         * #gp fault is ONLY valid for 32-bit processes also, which DO NOT
1223         * have a private LDT, and are trying to do a system call. Emulate it.
1224         */

1226        case T_SEGFLT + USER:   /* segment not present fault */
1227        case T_GPFLT + USER:    /* general protection violation */
1228 #ifdef _SYSCALL32_IMPL
1229                if (p->p_model != DATAMODEL_NATIVE) {
1230 #endif /* _SYSCALL32_IMPL */
```

```
1231                if (instr_is_lcall_syscall((caddr_t)rp->r_pc)) {
1232                        if (type == T_SEGFLT + USER)
1233                                ASSERT(p->p_ldt != NULL);

1235                        if ((p->p_ldt == NULL && type == T_GPFLT + USER) ||
1236                            type == T_SEGFLT + USER) {

1238                        /*
1239                         * The user attempted a system call via the obsolete
1240                         * call gate mechanism. Because the process doesn't have
1241                         * an LDT (i.e. the ldtr contains 0), a #gp results.
1242                         * Emulate the syscall here, just as we do above for a
1243                         * #np trap.
1244                         */

1246                        /*
1247                         * Since this is a not-present trap, rp->r_pc points to
1248                         * the trapping lcall instruction. We need to bump it
1249                         * to the next insn so the app can continue on.
1250                         */
1251                        rp->r_pc += LCALLSIZE;
1252                        lwp->lwp_regs = rp;

1254                        /*
1255                         * Normally the microstate of the LWP is forced back to
1256                         * LMS_USER by the syscall handlers. Emulate that
1257                         * behavior here.
1258                         */
1259                        mstate = LMS_USER;

1261                        dosyscall();
1262                        goto out;
1263                        }
1264                }
1265 #ifdef _SYSCALL32_IMPL
1266                 }
1267 #endif /* _SYSCALL32_IMPL */
1268                 /*
1269                  * If the current process is using a private LDT and the
1270                  * trapping instruction is sysenter, the sysenter instruction
1271                  * has been disabled on the CPU because it destroys segment
1272                  * registers. If this is the case, rewrite the instruction to
1273                  * be a safe system call and retry it. If this occurs on a CPU
1274                  * which doesn't even support sysenter, the result of all of
1275                  * this will be to emulate that particular instruction.
1276                  */
1277                 if (p->p_ldt != NULL &&
1278                     ldt_rewrite_syscall(rp, p, X86FSET_SEP))
1279                         goto out;

1281                 /*FALLTHROUGH*/

1283         case T_BOUNDFLT + USER: /* bound fault */
1284         case T_STKFLT + USER:   /* stack fault */
1285         case T_TSSFLT + USER:   /* invalid TSS fault */
1286                 if (tudebug)
1287                         showregs(type, rp, (caddr_t)0);
1288                 siginfo.si_signo = SIGSEGV;
1289                 siginfo.si_code  = SEGV_MAPERR;
1290                 siginfo.si_addr  = (caddr_t)rp->r_pc;
1291                 fault = FLTBOUNDS;
1292                 break;

1294         case T_ALIGNMENT + USER:        /* user alignment error (486) */
1295                 if (tudebug)
1296                         showregs(type, rp, (caddr_t)0);
```



```
1297                 bzero(&siginfo, sizeof (siginfo));
1298                 siginfo.si_signo = SIGBUS;
1299                 siginfo.si_code = BUS_ADRALN;
1300                 siginfo.si_addr = (caddr_t)rp->r_pc;
1301                 fault = FLTACCESS;
1302                 break;

1304         case T_SGLSTP + USER: /* single step/hw breakpoint exception */
1305                 if (tudebug && tudebugbpt)
1306                         showregs(type, rp, (caddr_t)0);

1308                 /* Was it single-stepping? */
1309                 if (lwp->lwp_pcb.pcb_drstat & DR_SINGLESTEP) {
1310                         pcb_t *pcb = &lwp->lwp_pcb;

1312                         rp->r_ps &= ~PS_T;
1313                         /*
1314                          * If both NORMAL_STEP and WATCH_STEP are in effect,
1315                          * give precedence to WATCH_STEP.  If neither is set,
1316                          * user must have set the PS_T bit in %efl; treat this
1317                          * as NORMAL_STEP.
1318                          */
1319                         if ((fault = undo_watch_step(&siginfo)) == 0 &&
1320                             ((pcb->pcb_flags & NORMAL_STEP) ||
1321                             !(pcb->pcb_flags & WATCH_STEP))) {
1322                                 siginfo.si_signo = SIGTRAP;
1323                                 siginfo.si_code = TRAP_TRACE;
1324                                 siginfo.si_addr = (caddr_t)rp->r_pc;
1325                                 fault = FLTTRACE;
1326                         }
1327                         pcb->pcb_flags &= ~(NORMAL_STEP|WATCH_STEP);
1328                 }
1329                 break;

1331         case T_BPTFLT + USER:   /* breakpoint trap */
1332                 if (tudebug && tudebugbpt)
1333                         showregs(type, rp, (caddr_t)0);
1334                 /*
1335                  * int 3 (the breakpoint instruction) leaves the pc referring
1336                  * to the address one byte after the breakpointed address.
1337                  * If the P_PR_BPTADJ flag has been set via /proc, We adjust
1338                  * it back so it refers to the breakpointed address.
1339                  */
1340                 if (p->p_proc_flag & P_PR_BPTADJ)
1341                         rp->r_pc--;
1342                 siginfo.si_signo = SIGTRAP;
1343                 siginfo.si_code  = TRAP_BRKPT;
1344                 siginfo.si_addr  = (caddr_t)rp->r_pc;
1345                 fault = FLTBPT;
1346                 break;

1348         case T_AST:
1349                 /*
1350                  * This occurs only after the cs register has been made to
1351                  * look like a kernel selector, either through debugging or
1352                  * possibly by functions like setcontext().  The thread is
1353                  * about to cause a general protection fault at common_iret()
1354                  * in locore.  We let that happen immediately instead of
1355                  * doing the T_AST processing.
1356                  */
1357                 goto cleanup;

1359         case T_AST + USER:      /* profiling, resched, h/w error pseudo trap */
1360                 if (lwp->lwp_pcb.pcb_flags & ASYNC_HWERR) {
1361                         proc_t *p = ttoproc(curthread);
1362                         extern void print_msg_hwerr(ctid_t ct_id, proc_t *p);
```

```
1364                        lwp->lwp_pcb.pcb_flags &= ~ASYNC_HWERR;
1365                        print_msg_hwerr(p->p_ct_process->conp_contract.ct_id,
1366                            p);
1367                        contract_process_hwerr(p->p_ct_process, p);
1368                        siginfo.si_signo = SIGKILL;
1369                        siginfo.si_code = SI_NOINFO;
1370                } else if (lwp->lwp_pcb.pcb_flags & CPC_OVERFLOW) {
1371                        lwp->lwp_pcb.pcb_flags &= ~CPC_OVERFLOW;
1372                        if (kcpc_overflow_ast()) {
1373                                /*
1374                                 * Signal performance counter overflow
1375                                 */
1376                                if (tudebug)
1377                                        showregs(type, rp, (caddr_t)0);
1378                                bzero(&siginfo, sizeof (siginfo));
1379                                siginfo.si_signo = SIGEMT;
1380                                siginfo.si_code = EMT_CPCOVF;
1381                                siginfo.si_addr = (caddr_t)rp->r_pc;
1382                                fault = FLTCPCOVF;
1383                        }
1384                }

1386                break;
1387        }

1389        /*
1390         * We can't get here from a system trap
1391         */
1392        ASSERT(type & USER);

1394        if (fault) {
1395                /* We took a fault so abort single step. */
1396                lwp->lwp_pcb.pcb_flags &= ~(NORMAL_STEP|WATCH_STEP);
1397                /*
1398                 * Remember the fault and fault adddress
1399                 * for real-time (SIGPROF) profiling.
1400                 */
1401                lwp->lwp_lastfault = fault;
1402                lwp->lwp_lastfaddr = siginfo.si_addr;

1404                DTRACE_PROC2(fault, int, fault, ksiginfo_t *, &siginfo);

1406                /*
1407                 * If a debugger has declared this fault to be an
1408                 * event of interest, stop the lwp.  Otherwise just
1409                 * deliver the associated signal.
1410                 */
1411                if (siginfo.si_signo != SIGKILL &&
1412                    prismember(&p->p_fltmask, fault) &&
1413                    stop_on_fault(fault, &siginfo) == 0)
1414                        siginfo.si_signo = 0;
1415        }

1417        if (siginfo.si_signo)
1418                trapsig(&siginfo, (fault != FLTFPE && fault != FLTCPCOVF));

1420        if (lwp->lwp_oweupc)
1421                profil_tick(rp->r_pc);

1423        if (ct->t_astflag | ct->t_sig_check) {
1424                /*
1425                 * Turn off the AST flag before checking all the conditions that
1426                 * may have caused an AST.  This flag is on whenever a signal or
1427                 * unusual condition should be handled after the next trap or
1428                 * syscall.
```



```
1429                 */
1430                astoff(ct);
1431                /*
1432                 * If a single-step trap occurred on a syscall (see above)
1433                 * recognize it now.  Do this before checking for signals
1434                 * because deferred_singlestep_trap() may generate a SIGTRAP to
1435                 * the LWP or may otherwise mark the LWP to call issig(FORREAL).
1436                 */
1437                if (lwp->lwp_pcb.pcb_flags & DEBUG_PENDING)
1438                        deferred_singlestep_trap((caddr_t)rp->r_pc);

1440                ct->t_sig_check = 0;

1442                mutex_enter(&p->p_lock);
1443                if (curthread->t_proc_flag & TP_CHANGEBIND) {
1444                        timer_lwpbind();
1445                        curthread->t_proc_flag &= ~TP_CHANGEBIND;
1446                }
1447                mutex_exit(&p->p_lock);

1449                /*
1450                 * for kaio requests that are on the per-process poll queue,
1451                 * aiop->aio_pollq, they're AIO_POLL bit is set, the kernel
1452                 * should copyout their result_t to user memory. by copying
1453                 * out the result_t, the user can poll on memory waiting
1454                 * for the kaio request to complete.
1455                 */
1456                if (p->p_aio)
1457                        aio_cleanup(0);
1458                /*
1459                 * If this LWP was asked to hold, call holdlwp(), which will
1460                 * stop.  holdlwps() sets this up and calls pokelwps() which
1461                 * sets the AST flag.
1462                 *
1463                 * Also check TP_EXITLWP, since this is used by fresh new LWPs
1464                 * through lwp_rtt().  That flag is set if the lwp_create(2)
1465                 * syscall failed after creating the LWP.
1466                 */
1467                if (ISHOLD(p))
1468                        holdlwp();

1470                /*
1471                 * All code that sets signals and makes ISSIG evaluate true must
1472                 * set t_astflag afterwards.
1473                 */
1474                if (ISSIG_PENDING(ct, lwp, p)) {
1475                        if (issig(FORREAL))
1476                                psig();
1477                        ct->t_sig_check = 1;
1478                }

1480                if (ct->t_rprof != NULL) {
1481                        realsigprof(0, 0, 0);
1482                        ct->t_sig_check = 1;
1483                }

1485                /*
1486                 * /proc can't enable/disable the trace bit itself
1487                 * because that could race with the call gate used by
1488                 * system calls via "lcall". If that happened, an
1489                 * invalid EFLAGS would result. prstep()/prnostep()
1490                 * therefore schedule an AST for the purpose.
1491                 */
1492                if (lwp->lwp_pcb.pcb_flags & REQUEST_STEP) {
1493                        lwp->lwp_pcb.pcb_flags &= ~REQUEST_STEP;
1494                        rp->r_ps |= PS_T;
```

```
1495                 }
1496                 if (lwp->lwp_pcb.pcb_flags & REQUEST_NOSTEP) {
1497                         lwp->lwp_pcb.pcb_flags &= ~REQUEST_NOSTEP;
1498                         rp->r_ps &= ~PS_T;
1499                 }
1500         }

1502 out:    /* We can't get here from a system trap */
1503         ASSERT(type & USER);

1505         if (ISHOLD(p))
1506                 holdlwp();

1508         /*
1509          * Set state to LWP_USER here so preempt won't give us a kernel
1510          * priority if it occurs after this point.  Call CL_TRAPRET() to
1511          * restore the user-level priority.
1512          *
1513          * It is important that no locks (other than spinlocks) be entered
1514          * after this point before returning to user mode (unless lwp_state
1515          * is set back to LWP_SYS).
1516          */
1517         lwp->lwp_state = LWP_USER;

1519         if (ct->t_trapret) {
1520                 ct->t_trapret = 0;
1521                 thread_lock(ct);
1522                 CL_TRAPRET(ct);
1523                 thread_unlock(ct);
1524         }
1525         if (CPU->cpu_runrun || curthread->t_schedflag & TS_ANYWAITQ)
1526                 preempt();
1527         prunstop();
1528         (void) new_mstate(ct, mstate);

1530         /* Kernel probe */
1531         TNF_PROBE_1(thread_state, "thread", /* CSTYLED */,
1532             tnf_microstate, state, LMS_USER);

1534         return;

1536 cleanup:        /* system traps end up here */
1537         ASSERT(!(type & USER));
1538 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  105681 Fri May  8 18:10:40 2015
new/usr/src/uts/i86pc/vm/hat_i86.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

1127 /*
1128  * Allocate any hat resources required for a process being swapped in.
1129  */
1130 /*ARGSUSED*/
1131 void
1132 hat_swapin(hat_t *hat)
1133 {
1134         /* do nothing - we let everything fault back in */
1135 }

1137 /*
1138  * Unload all translations associated with an address space of a process
1139  * that is being swapped out.
1140  */
1141 void
1142 hat_swapout(hat_t *hat)
1143 {
1144         uintptr_t       vaddr = (uintptr_t)0;
1145         uintptr_t       eaddr = _userlimit;
1146         htable_t        *ht = NULL;
1147         level_t         l;

1149         XPV_DISALLOW_MIGRATE();
1150         /*
1151          * We can't just call hat_unload(hat, 0, _userlimit...)  here, because
1152          * seg_spt and shared pagetables can't be swapped out.
1153          * Take a look at segspt_shmswapout() - it's a big no-op.
1154          *
1155          * Instead we'll walk through all the address space and unload
1156          * any mappings which we are sure are not shared, not locked.
1157          */
1158         ASSERT(IS_PAGEALIGNED(vaddr));
1159         ASSERT(IS_PAGEALIGNED(eaddr));
1160         ASSERT(AS_LOCK_HELD(hat->hat_as, &hat->hat_as->a_lock));
1161         if ((uintptr_t)hat->hat_as->a_userlimit < eaddr)
1162                 eaddr = (uintptr_t)hat->hat_as->a_userlimit;

1164         while (vaddr < eaddr) {
1165                 (void) htable_walk(hat, &ht, &vaddr, eaddr);
1166                 if (ht == NULL)
1167                         break;

1169                 ASSERT(!IN_VA_HOLE(vaddr));

1171                 /*
1172                  * If the page table is shared skip its entire range.
1173                  */
1174                 l = ht->ht_level;
1175                 if (ht->ht_flags & HTABLE_SHARED_PFN) {
1176                         vaddr = ht->ht_vaddr + LEVEL_SIZE(l + 1);
1177                         htable_release(ht);
1178                         ht = NULL;
1179                         continue;
```



```
1180                 }

1182                 /*
1183                  * If the page table has no locked entries, unload this one.
1184                  */
1185                 if (ht->ht_lock_cnt == 0)
1186                         hat_unload(hat, (caddr_t)vaddr, LEVEL_SIZE(l),
1187                             HAT_UNLOAD_UNMAP);

1189                 /*
1190                  * If we have a level 0 page table with locked entries,
1191                  * skip the entire page table, otherwise skip just one entry.
1192                  */
1193                 if (ht->ht_lock_cnt > 0 && l == 0)
1194                         vaddr = ht->ht_vaddr + LEVEL_SIZE(1);
1195                 else
1196                         vaddr += LEVEL_SIZE(l);
1197         }
1198         if (ht)
1199                 htable_release(ht);

1201         /*
1202          * We're in swapout because the system is low on memory, so
1203          * go back and flush all the htables off the cached list.
1204          */
1205         htable_purge_hat(hat);
1206         XPV_ALLOW_MIGRATE();
1207 }

1209 /*
1128  * returns number of bytes that have valid mappings in hat.
1129  */
1130 size_t
1131 hat_get_mapped_size(hat_t *hat)
1132 {
1133         size_t total = 0;
1134         int l;

1136         for (l = 0; l <= mmu.max_page_level; l++)
1137                 total += (hat->hat_pages_mapped[l] << LEVEL_SHIFT(l));
1138         total += hat->hat_ism_pgcnt;

1140         return (total);
1141 }
______unchanged_portion_omitted_
```

```
**********************************************************
   16469 Fri May  8 18:10:40 2015
new/usr/src/uts/i86xpv/vm/seg_mf.c
use NULL dump segop as a shorthand for no-op
Instead of forcing every segment driver to implement a dummy function that
does nothing, handle NULL dump segop function pointer as a no-op shorthand.
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
use NULL setpagesize segop as a shorthand for ENOTSUP
Instead of forcing every segment driver to implement a dummp function to
return (hopefully) ENOTSUP, handle NULL setpagesize segop function pointer
as "return ENOTSUP" shorthand.
use NULL capable segop as a shorthand for no-capabilities
Instead of forcing every segment driver to implement a dummy "return 0"
function, handle NULL capable segop function pointer as "no copabilities
supported" shorthand.
segop_getpolicy already checks for a NULL op
seg_inherit_notsup is redundant since segop_inherit checks for NULL properly
use C99 initializers in segment ops structures
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

 102 static const struct seg_ops segmf_ops;
 102 static struct seg_ops segmf_ops;

 104 static int segmf_fault_gref_range(struct seg *seg, caddr_t addr, size_t len);

 106 static struct segmf_data *
 107 segmf_data_zalloc(struct seg *seg)
 108 {
 109         struct segmf_data *data = kmem_zalloc(sizeof (*data), KM_SLEEP);

 111         mutex_init(&data->lock, "segmf.lock", MUTEX_DEFAULT, NULL);
 112         seg->s_ops = &segmf_ops;
 113         seg->s_data = data;
 114         return (data);
 115 }
______unchanged_portion_omitted_

 472 /*ARGSUSED*/
 473 static void
 474 segmf_dump(struct seg *seg)
 475 {}

 477 /*ARGSUSED*/
 473 static int
 474 segmf_pagelock(struct seg *seg, caddr_t addr, size_t len,
 475     struct page ***ppp, enum lock_type type, enum seg_rw rw)
 476 {
 477         return (ENOTSUP);
 478 }

 485 /*ARGSUSED*/
 486 static int
 487 segmf_setpagesize(struct seg *seg, caddr_t addr, size_t len, uint_t szc)
 488 {
 489         return (ENOTSUP);
 490 }
```



```
 480 static int
 481 segmf_getmemid(struct seg *seg, caddr_t addr, memid_t *memid)
 482 {
 483         struct segmf_data *data = seg->s_data;

 485         memid->val[0] = (uintptr_t)VTOCVP(data->vp);
 486         memid->val[1] = (uintptr_t)seg_page(seg, addr);
 487         return (0);
 488 }

 502 /*ARGSUSED*/
 503 static lgrp_mem_policy_info_t *
 504 segmf_getpolicy(struct seg *seg, caddr_t addr)
 505 {
 506         return (NULL);
 507 }

 509 /*ARGSUSED*/
 510 static int
 511 segmf_capable(struct seg *seg, segcapability_t capability)
 512 {
 513         return (0);
 514 }

 490 /*
 491  * Add a set of contiguous foreign MFNs to the segment. soft-locking them.  The
 492  * pre-faulting is necessary due to live migration; in particular we must
 493  * return an error in response to IOCTL_PRIVCMD_MMAPBATCH rather than faulting
 494  * later on a bad MFN.  Whilst this isn't necessary for the other MMAP
 495  * ioctl()s, we lock them too, as they should be transitory.
 496  */
 497 int
 498 segmf_add_mfns(struct seg *seg, caddr_t addr, mfn_t mfn,
 499     pgcnt_t pgcnt, domid_t domid)
 500 {
 501         struct segmf_data *data = seg->s_data;
 502         pgcnt_t base;
 503         faultcode_t fc;
 504         pgcnt_t i;
 505         int error = 0;

 507         if (seg->s_ops != &segmf_ops)
 508                 return (EINVAL);

 510         /*
 511          * Don't mess with dom0.
 512          *
 513          * Only allow the domid to be set once for the segment.
 514          * After that attempts to add mappings to this segment for
 515          * other domains explicitly fails.
 516          */

 518         if (domid == 0 || domid == DOMID_SELF)
 519                 return (EACCES);

 521         mutex_enter(&data->lock);

 523         if (data->domid == 0)
 524                 data->domid = domid;

 526         if (data->domid != domid) {
 527                 error = EINVAL;
 528                 goto out;
 529         }

 531         base = seg_page(seg, addr);
```

```
533         for (i = 0; i < pgcnt; i++) {
534                 data->map[base + i].t_type = SEGMF_MAP_MFN;
535                 data->map[base + i].u.m.m_mfn = mfn++;
536         }

538         fc = segmf_fault_range(seg->s_as->a_hat, seg, addr,
539             pgcnt * MMU_PAGESIZE, F_SOFTLOCK, S_OTHER);

541         if (fc != 0) {
542                 error = fc_decode(fc);
543                 for (i = 0; i < pgcnt; i++) {
544                         data->map[base + i].t_type = SEGMF_MAP_EMPTY;
545                 }
546         }

548 out:
549         mutex_exit(&data->lock);
550         return (error);
551 }
```

*unchanged_portion_omitted_*

```
734 static const struct seg_ops segmf_ops = {
735         .dup            = segmf_dup,
736         .unmap          = segmf_unmap,
737         .free           = segmf_free,
738         .fault          = segmf_fault,
739         .faulta         = segmf_faulta,
740         .setprot        = segmf_setprot,
741         .checkprot      = segmf_checkprot,
742         .kluster        = segmf_kluster,
743         .sync           = segmf_sync,
744         .incore         = segmf_incore,
745         .lockop         = segmf_lockop,
746         .getprot        = segmf_getprot,
747         .getoffset      = segmf_getoffset,
748         .gettype        = segmf_gettype,
749         .getvp          = segmf_getvp,
750         .advise         = segmf_advise,
751         .pagelock       = segmf_pagelock,
752         .getmemid       = segmf_getmemid,
760 static struct seg_ops segmf_ops = {
761         segmf_dup,
762         segmf_unmap,
763         segmf_free,
764         segmf_fault,
765         segmf_faulta,
766         segmf_setprot,
767         segmf_checkprot,
768         (int (*)())segmf_kluster,
769         (size_t (*)(struct seg *))NULL, /* swapout */
770         segmf_sync,
771         segmf_incore,
772         segmf_lockop,
773         segmf_getprot,
774         segmf_getoffset,
775         segmf_gettype,
776         segmf_getvp,
777         segmf_advise,
778         segmf_dump,
779         segmf_pagelock,
780         segmf_setpagesize,
781         segmf_getmemid,
782         segmf_getpolicy,
783         segmf_capable,
784         seg_inherit_notsup
```



```
753 };
```

*unchanged_portion_omitted_*

```
**********************************************************
   35882 Fri May  8 18:10:40 2015
new/usr/src/uts/intel/ia32/os/syscall.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

 137 /*
 138  * Called from syscall() when a non-trivial 32-bit system call occurs.
 139  *      Sets up the args and returns a pointer to the handler.
 140  */
 141 struct sysent *
 142 syscall_entry(kthread_t *t, long *argp)
 143 {
 144         klwp_t *lwp = ttolwp(t);
 145         struct regs *rp = lwptoregs(lwp);
 146         unsigned int code;
 147         struct sysent *callp;
 148         struct sysent *se = LWP_GETSYSENT(lwp);
 149         int error = 0;
 150         uint_t nargs;

 152         ASSERT(t == curthread);
 152         ASSERT(t == curthread && curthread->t_schedflag & TS_DONT_SWAP);

 154         lwp->lwp_ru.sysc++;
 155         lwp->lwp_eosys = NORMALRETURN;  /* assume this will be normal */

 157         /*
 158          * Set lwp_ap to point to the args, even if none are needed for this
 159          * system call.  This is for the loadable-syscall case where the
 160          * number of args won't be known until the system call is loaded, and
 161          * also maintains a non-NULL lwp_ap setup for get_syscall_args(). Note
 162          * that lwp_ap MUST be set to a non-NULL value _BEFORE_ t_sysnum is
 163          * set to non-zero; otherwise get_syscall_args(), seeing a non-zero
 164          * t_sysnum for this thread, will charge ahead and dereference lwp_ap.
 165          */
 166         lwp->lwp_ap = argp;             /* for get_syscall_args */

 168         code = rp->r_r0;
 169         t->t_sysnum = (short)code;
 170         callp = code >= NSYSCALL ? &nosys_ent : se + code;

 172         if ((t->t_pre_sys | syscalltrace) != 0) {
 173                 error = pre_syscall();

 175                 /*
 176                  * pre_syscall() has taken care so that lwp_ap is current;
 177                  * it either points to syscall-entry-saved amd64 regs,
 178                  * or it points to lwp_arg[], which has been re-copied from
 179                  * the ia32 ustack, but either way, it's a current copy after
 180                  * /proc has possibly mucked with the syscall args.
 181                  */

 183                 if (error)
 184                         return (&sysent_err);   /* use dummy handler */
 185         }

 187         /*
 188          * Fetch the system call arguments to the kernel stack copy used
```



```
 189          * for syscall handling.
 190          * Note: for loadable system calls the number of arguments required
 191          * may not be known at this point, and will be zero if the system call
 192          * was never loaded.  Once the system call has been loaded, the number
 193          * of args is not allowed to be changed.
 194          */
 195         if ((nargs = (uint_t)callp->sy_narg) != 0 &&
 196             COPYIN_ARGS32(rp, argp, nargs)) {
 197                 (void) set_errno(EFAULT);
 198                 return (&sysent_err);   /* use dummy handler */
 199         }

 201         return (callp);         /* return sysent entry for caller */
 202 }
______unchanged_portion_omitted_

 227 /*
 228  * Perform pre-system-call processing, including stopping for tracing,
 229  * auditing, etc.
 230  *
 231  * This routine is called only if the t_pre_sys flag is set. Any condition
 232  * requiring pre-syscall handling must set the t_pre_sys flag. If the
 233  * condition is persistent, this routine will repost t_pre_sys.
 234  */
 235 int
 236 pre_syscall()
 237 {
 238         kthread_t *t = curthread;
 239         unsigned code = t->t_sysnum;
 240         klwp_t *lwp = ttolwp(t);
 241         proc_t *p = ttoproc(t);
 242         int     repost;

 244         t->t_pre_sys = repost = 0;      /* clear pre-syscall processing flag */

 246         ASSERT(t->t_schedflag & TS_DONT_SWAP);

 246 #if defined(DEBUG)
 247         /*
 248          * On the i386 kernel, lwp_ap points at the piece of the thread
 249          * stack that we copy the users arguments into.
 250          *
 251          * On the amd64 kernel, the syscall arguments in the rdi..r9
 252          * registers should be pointed at by lwp_ap.  If the args need to
 253          * be copied so that those registers can be changed without losing
 254          * the ability to get the args for /proc, they can be saved by
 255          * save_syscall_args(), and lwp_ap will be restored by post_syscall().
 256          */
 257         if (lwp_getdatamodel(lwp) == DATAMODEL_NATIVE) {
 258 #if defined(_LP64)
 259                 ASSERT(lwp->lwp_ap == (long *)&lwptoregs(lwp)->r_rdi);
 260         } else {
 261 #endif
 262                 ASSERT((caddr_t)lwp->lwp_ap > t->t_stkbase &&
 263                     (caddr_t)lwp->lwp_ap < t->t_stk);
 264         }
 265 #endif  /* DEBUG */

 267         /*
 268          * Make sure the thread is holding the latest credentials for the
 269          * process.  The credentials in the process right now apply to this
 270          * thread for the entire system call.
 271          */
 272         if (t->t_cred != p->p_cred) {
 273                 cred_t *oldcred = t->t_cred;
 274                 /*
```

```
275                  * DTrace accesses t_cred in probe context.  t_cred must
276                  * always be either NULL, or point to a valid, allocated cred
277                  * structure.
278                  */
279                 t->t_cred = crgetcred();
280                 crfree(oldcred);
281         }
283         /*
284          * From the proc(4) manual page:
285          * When entry to a system call is being traced, the traced process
286          * stops after having begun the call to the system but before the
287          * system call arguments have been fetched from the process.
288          */
289         if (PTOU(p)->u_systrap) {
290                 if (prismember(&PTOU(p)->u_entrymask, code)) {
291                         mutex_enter(&p->p_lock);
292                         /*
293                          * Recheck stop condition, now that lock is held.
294                          */
295                         if (PTOU(p)->u_systrap &&
296                             prismember(&PTOU(p)->u_entrymask, code)) {
297                                 stop(PR_SYSENTRY, code);
299                                 /*
300                                  * /proc may have modified syscall args,
301                                  * either in regs for amd64 or on ustack
302                                  * for ia32.  Either way, arrange to
303                                  * copy them again, both for the syscall
304                                  * handler and for other consumers in
305                                  * post_syscall (like audit).  Here, we
306                                  * only do amd64, and just set lwp_ap
307                                  * back to the kernel-entry stack copy;
308                                  * the syscall ml code redoes
309                                  * move-from-regs to set up for the
310                                  * syscall handler after we return.  For
311                                  * ia32, save_syscall_args() below makes
312                                  * an lwp_ap-accessible copy.
313                                  */
314 #if defined(_LP64)
315                                 if (lwp_getdatamodel(lwp) == DATAMODEL_NATIVE) {
316                                         lwp->lwp_argsaved = 0;
317                                         lwp->lwp_ap =
318                                             (long *)&lwptoregs(lwp)->r_rdi;
319                                 }
320 #endif
321                         }
322                         mutex_exit(&p->p_lock);
323                 }
324                 repost = 1;
325         }
327         /*
328          * ia32 kernel, or ia32 proc on amd64 kernel: keep args in
329          * lwp_arg for post-syscall processing, regardless of whether
330          * they might have been changed in /proc above.
331          */
332 #if defined(_LP64)
333         if (lwp_getdatamodel(lwp) != DATAMODEL_NATIVE)
334 #endif
335                 (void) save_syscall_args();
337         if (lwp->lwp_sysabort) {
338                 /*
339                  * lwp_sysabort may have been set via /proc while the process
340                  * was stopped on PR_SYSENTRY.  If so, abort the system call.
```



```
341                  * Override any error from the copyin() of the arguments.
342                  */
343                 lwp->lwp_sysabort = 0;
344                 (void) set_errno(EINTR);        /* forces post_sys */
345                 t->t_pre_sys = 1;       /* repost anyway */
346                 return (1);             /* don't do system call, return EINTR */
347         }
349         /*
350          * begin auditing for this syscall if the c2audit module is loaded
351          * and auditing is enabled
352          */
353         if (audit_active == C2AUDIT_LOADED) {
354                 uint32_t auditing = au_zone_getstate(NULL);
356                 if (auditing & AU_AUDIT_MASK) {
357                         int error;
358                         if (error = audit_start(T_SYSCALL, code, auditing, \
359                             0, lwp)) {
360                                 t->t_pre_sys = 1;       /* repost anyway */
361                                 (void) set_errno(error);
362                                 return (1);
363                         }
364                         repost = 1;
365                 }
366         }
368 #ifndef NPROBE
369         /* Kernel probe */
370         if (tnf_tracing_active) {
371                 TNF_PROBE_1(syscall_start, "syscall thread", /* CSTYLED */,
372                         tnf_sysnum,     sysnum,         t->t_sysnum);
373                 t->t_post_sys = 1;      /* make sure post_syscall runs */
374                 repost = 1;
375         }
376 #endif /* NPROBE */
378 #ifdef SYSCALLTRACE
379         if (syscalltrace) {
380                 int i;
381                 long *ap;
382                 char *cp;
383                 char *sysname;
384                 struct sysent *callp;
386                 if (code >= NSYSCALL)
387                         callp = &nosys_ent;     /* nosys has no args */
388                 else
389                         callp = LWP_GETSYSENT(lwp) + code;
390                 (void) save_syscall_args();
391                 mutex_enter(&systrace_lock);
392                 printf("%d: ", p->p_pid);
393                 if (code >= NSYSCALL)
394                         printf("0x%x", code);
395                 else {
396                         sysname = mod_getsysname(code);
397                         printf("%s[0x%x/0x%p]", sysname == NULL ? "NULL" :
398                             sysname, code, callp->sy_callc);
399                 }
400                 cp = "(";
401                 for (i = 0, ap = lwp->lwp_ap; i < callp->sy_narg; i++, ap++) {
402                         printf("%s%lx", cp, *ap);
403                         cp = ", ";
404                 }
405                 if (i)
406                         printf(")");
```

```
407                 printf(" %s id=0x%p\n", PTOU(p)->u_comm, curthread);
408                 mutex_exit(&systrace_lock);
409         }
410 #endif /* SYSCALLTRACE */

412         /*
413          * If there was a continuing reason for pre-syscall processing,
414          * set the t_pre_sys flag for the next system call.
415          */
416         if (repost)
417                 t->t_pre_sys = 1;
418         lwp->lwp_error = 0;     /* for old drivers */
419         lwp->lwp_badpriv = PRIV_NONE;
420         return (0);
421 }
```

unchanged_portion_omitted_

```
**********************************************************
  413546 Fri May  8 18:10:40 2015
new/usr/src/uts/sfmmu/vm/hat_sfmmu.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
remove xhat
The xhat infrastructure was added to support hardware such as the zulu
graphics card - hardware which had on-board MMUs.  The VM used the xhat code
to keep the CPU's and Zulu's page tables in-sync.  Since the only xhat user
was zulu (which is gone), we can safely remove it simplifying the whole VM
subsystem.
Assorted notes:
- AS_BUSY flag was used solely by xhat
**********************************************************
```

```
   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) 1993, 2010, Oracle and/or its affiliates. All rights reserved.
  23  */
  24 /*
  25  * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
  26  */

  28 /*
  29  * VM - Hardware Address Translation management for Spitfire MMU.
  30  *
  31  * This file implements the machine specific hardware translation
  32  * needed by the VM system.  The machine independent interface is
  33  * described in <vm/hat.h> while the machine dependent interface
  34  * and data structures are described in <vm/hat_sfmmu.h>.
  35  *
  36  * The hat layer manages the address translation hardware as a cache
  37  * driven by calls from the higher levels in the VM system.
  38  */

  40 #include <sys/types.h>
  41 #include <sys/kstat.h>
  42 #include <vm/hat.h>
  43 #include <vm/hat_sfmmu.h>
  44 #include <vm/page.h>
  45 #include <sys/pte.h>
  46 #include <sys/systm.h>
  47 #include <sys/mman.h>
```



```
  48 #include <sys/sysmacros.h>
  49 #include <sys/machparam.h>
  50 #include <sys/vtrace.h>
  51 #include <sys/kmem.h>
  52 #include <sys/mmu.h>
  53 #include <sys/cmn_err.h>
  54 #include <sys/cpu.h>
  55 #include <sys/cpuvar.h>
  56 #include <sys/debug.h>
  57 #include <sys/lgrp.h>
  58 #include <sys/archsystm.h>
  59 #include <sys/machsystm.h>
  60 #include <sys/vmsystm.h>
  61 #include <vm/as.h>
  62 #include <vm/seg.h>
  63 #include <vm/seg_kp.h>
  64 #include <vm/seg_kmem.h>
  65 #include <vm/seg_kpm.h>
  66 #include <vm/rm.h>
  67 #include <sys/t_lock.h>
  68 #include <sys/obpdefs.h>
  69 #include <sys/vm_machparam.h>
  70 #include <sys/var.h>
  71 #include <sys/trap.h>
  72 #include <sys/machtrap.h>
  73 #include <sys/scb.h>
  74 #include <sys/bitmap.h>
  75 #include <sys/machlock.h>
  76 #include <sys/membar.h>
  77 #include <sys/atomic.h>
  78 #include <sys/cpu_module.h>
  79 #include <sys/prom_debug.h>
  80 #include <sys/ksynch.h>
  81 #include <sys/mem_config.h>
  82 #include <sys/mem_cage.h>
  83 #include <vm/vm_dep.h>
  84 #include <vm/xhat_sfmmu.h>
  84 #include <sys/fpu/fpusystm.h>
  85 #include <vm/mach_kpm.h>
  86 #include <sys/callb.h>

  88 #ifdef  DEBUG
  89 #define SFMMU_VALIDATE_HMERID(hat, rid, saddr, len)                     \
  90         if (SFMMU_IS_SHMERID_VALID(rid)) {                             \
  91                 caddr_t _eaddr = (saddr) + (len);                      \
  92                 sf_srd_t *_srdp;                                       \
  93                 sf_region_t *_rgnp;                                    \
  94                 ASSERT((rid) < SFMMU_MAX_HME_REGIONS);                 \
  95                 ASSERT(SF_RGNMAP_TEST(hat->sfmmu_hmeregion_map, rid)); \
  96                 ASSERT((hat) != ksfmmup);                              \
  97                 _srdp = (hat)->sfmmu_srdp;                             \
  98                 ASSERT(_srdp != NULL);                                 \
  99                 ASSERT(_srdp->srd_refcnt != 0);                        \
 100                 _rgnp = _srdp->srd_hmergnp[(rid)];                     \
 101                 ASSERT(_rgnp != NULL && _rgnp->rgn_id == rid);         \
 102                 ASSERT(_rgnp->rgn_refcnt != 0);                        \
 103                 ASSERT(!(_rgnp->rgn_flags & SFMMU_REGION_FREE));       \
 104                 ASSERT((_rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK) ==  \
 105                     SFMMU_REGION_HME);                                 \
 106                 ASSERT((saddr) >= _rgnp->rgn_saddr);                   \
 107                 ASSERT((saddr) < _rgnp->rgn_saddr + _rgnp->rgn_size);  \
 108                 ASSERT(_eaddr > _rgnp->rgn_saddr);                     \
 109                 ASSERT(_eaddr <= _rgnp->rgn_saddr + _rgnp->rgn_size);  \
 110         }

 112 #define SFMMU_VALIDATE_SHAREDHBLK(hmeblkp, srdp, rgnp, rid)             \
```

```
 113 {                                                                   \
 114                 caddr_t _hsva;                                      \
 115                 caddr_t _heva;                                      \
 116                 caddr_t _rsva;                                      \
 117                 caddr_t _reva;                                      \
 118                 int     _ttesz = get_hblk_ttesz(hmeblkp);           \
 119                 int     _flagtte;                                   \
 120                 ASSERT((srdp)->srd_refcnt != 0);                    \
 121                 ASSERT((rid) < SFMMU_MAX_HME_REGIONS);              \
 122                 ASSERT((rgnp)->rgn_id == rid);                      \
 123                 ASSERT(!((rgnp)->rgn_flags & SFMMU_REGION_FREE));   \
 124                 ASSERT(((rgnp)->rgn_flags & SFMMU_REGION_TYPE_MASK) ==  \
 125                     SFMMU_REGION_HME);                              \
 126                 ASSERT(_ttesz <= (rgnp)->rgn_pgszc);                \
 127                 _hsva = (caddr_t)get_hblk_base(hmeblkp);            \
 128                 _heva = get_hblk_endaddr(hmeblkp);                  \
 129                 _rsva = (caddr_t)P2ALIGN(                           \
 130                     (uintptr_t)(rgnp)->rgn_saddr, HBLK_MIN_BYTES);  \
 131                 _reva = (caddr_t)P2ROUNDUP(                         \
 132                     (uintptr_t)((rgnp)->rgn_saddr + (rgnp)->rgn_size),  \
 133                     HBLK_MIN_BYTES);                                \
 134                 ASSERT(_hsva >= _rsva);                             \
 135                 ASSERT(_hsva < _reva);                              \
 136                 ASSERT(_heva > _rsva);                              \
 137                 ASSERT(_heva <= _reva);                             \
 138                 _flagtte = (_ttesz < HBLK_MIN_TTESZ) ? HBLK_MIN_TTESZ :  \
 139                     _ttesz;                                         \
 140                 ASSERT(rgnp->rgn_hmeflags & (0x1 << _flagtte));     \
 141 }
```

unchanged_portion_omitted_

```
1053 /*
1054  * Initialize the hardware address translation structures.
1055  */
1056 void
1057 hat_init(void)
1058 {
1059         int             i;
1060         uint_t          sz;
1061         size_t          size;

1063         hat_lock_init();
1064         hat_kstat_init();

1066         /*
1067          * Hardware-only bits in a TTE
1068          */
1069         MAKE_TTE_MASK(&hw_tte);

1071         hat_init_pagesizes();

1073         /* Initialize the hash locks */
1074         for (i = 0; i < khmehash_num; i++) {
1075                 mutex_init(&khme_hash[i].hmehash_mutex, NULL,
1076                     MUTEX_DEFAULT, NULL);
1077                 khme_hash[i].hmeh_nextpa = HMEBLK_ENDPA;
1078         }
1079         for (i = 0; i < uhmehash_num; i++) {
1080                 mutex_init(&uhme_hash[i].hmehash_mutex, NULL,
1081                     MUTEX_DEFAULT, NULL);
1082                 uhme_hash[i].hmeh_nextpa = HMEBLK_ENDPA;
1083         }
1084         khmehash_num--;         /* make sure counter starts from 0 */
1085         uhmehash_num--;         /* make sure counter starts from 0 */

1087         /*
```



```
1088          * Allocate context domain structures.
1089          *
1090          * A platform may choose to modify max_mmu_ctxdoms in
1091          * set_platform_defaults(). If a platform does not define
1092          * a set_platform_defaults() or does not choose to modify
1093          * max_mmu_ctxdoms, it gets one MMU context domain for every CPU.
1094          *
1095          * For all platforms that have CPUs sharing MMUs, this
1096          * value must be defined.
1097          */
1098         if (max_mmu_ctxdoms == 0)
1099                 max_mmu_ctxdoms = max_ncpus;

1101         size = max_mmu_ctxdoms * sizeof (mmu_ctx_t *);
1102         mmu_ctxs_tbl = kmem_zalloc(size, KM_SLEEP);

1104         /* mmu_ctx_t is 64 bytes aligned */
1105         mmuctxdom_cache = kmem_cache_create("mmuctxdom_cache",
1106             sizeof (mmu_ctx_t), 64, NULL, NULL, NULL, NULL, NULL, 0);
1107         /*
1108          * MMU context domain initialization for the Boot CPU.
1109          * This needs the context domains array allocated above.
1110          */
1111         mutex_enter(&cpu_lock);
1112         sfmmu_cpu_init(CPU);
1113         mutex_exit(&cpu_lock);

1115         /*
1116          * Intialize ism mapping list lock.
1117          */

1119         mutex_init(&ism_mlist_lock, NULL, MUTEX_DEFAULT, NULL);

1121         /*
1122          * Each sfmmu structure carries an array of MMU context info
1123          * structures, one per context domain. The size of this array depends
1124          * on the maximum number of context domains. So, the size of the
1125          * sfmmu structure varies per platform.
1126          *
1127          * sfmmu is allocated from static arena, because trap
1128          * handler at TL > 0 is not allowed to touch kernel relocatable
1129          * memory. sfmmu's alignment is changed to 64 bytes from
1130          * default 8 bytes, as the lower 6 bits will be used to pass
1131          * pgcnt to vtag_flush_pgcnt_tl1.
1132          */
1133         size = sizeof (sfmmu_t) + sizeof (sfmmu_ctx_t) * (max_mmu_ctxdoms - 1);

1135         sfmmuid_cache = kmem_cache_create("sfmmuid_cache", size,
1136             64, sfmmu_idcache_constructor, sfmmu_idcache_destructor,
1137             NULL, NULL, static_arena, 0);

1139         sfmmu_tsbinfo_cache = kmem_cache_create("sfmmu_tsbinfo_cache",
1140             sizeof (struct tsb_info), 0, NULL, NULL, NULL, NULL, NULL, 0);

1142         /*
1143          * Since we only use the tsb8k cache to "borrow" pages for TSBs
1144          * from the heap when low on memory or when TSB_FORCEALLOC is
1145          * specified, don't use magazines to cache them--we want to return
1146          * them to the system as quickly as possible.
1147          */
1148         sfmmu_tsb8k_cache = kmem_cache_create("sfmmu_tsb8k_cache",
1149             MMU_PAGESIZE, MMU_PAGESIZE, NULL, NULL, NULL, NULL,
1150             static_arena, KMC_NOMAGAZINE);

1152         /*
1153          * Set tsb_alloc_hiwater to 1/tsb_alloc_hiwater_factor of physical
```

```
1154          * memory, which corresponds to the old static reserve for TSBs.
1155          * tsb_alloc_hiwater_factor defaults to 32.  This caps the amount of
1156          * memory we'll allocate for TSB slabs; beyond this point TSB
1157          * allocations will be taken from the kernel heap (via
1158          * sfmmu_tsb8k_cache) and will be throttled as would any other kmem
1159          * consumer.
1160          */
1161         if (tsb_alloc_hiwater_factor == 0) {
1162                 tsb_alloc_hiwater_factor = TSB_ALLOC_HIWATER_FACTOR_DEFAULT;
1163         }
1164         SFMMU_SET_TSB_ALLOC_HIWATER(physmem);

1166         for (sz = tsb_slab_ttesz; sz > 0; sz--) {
1167                 if (!(disable_large_pages & (1 << sz)))
1168                         break;
1169         }

1171         if (sz < tsb_slab_ttesz) {
1172                 tsb_slab_ttesz = sz;
1173                 tsb_slab_shift = MMU_PAGESHIFT + (sz << 1) + sz;
1174                 tsb_slab_size = 1 << tsb_slab_shift;
1175                 tsb_slab_mask = (1 << (tsb_slab_shift - MMU_PAGESHIFT)) - 1;
1176                 use_bigtsb_arena = 0;
1177         } else if (use_bigtsb_arena &&
1178             (disable_large_pages & (1 << bigtsb_slab_ttesz))) {
1179                 use_bigtsb_arena = 0;
1180         }

1182         if (!use_bigtsb_arena) {
1183                 bigtsb_slab_shift = tsb_slab_shift;
1184         }
1185         SFMMU_SET_TSB_MAX_GROWSIZE(physmem);

1187         /*
1188          * On smaller memory systems, allocate TSB memory in smaller chunks
1189          * than the default 4M slab size. We also honor disable_large_pages
1190          * here.
1191          *
1192          * The trap handlers need to be patched with the final slab shift,
1193          * since they need to be able to construct the TSB pointer at runtime.
1194          */
1195         if ((tsb_max_growsize <= TSB_512K_SZCODE) &&
1196             !(disable_large_pages & (1 << TTE512K))) {
1197                 tsb_slab_ttesz = TTE512K;
1198                 tsb_slab_shift = MMU_PAGESHIFT512K;
1199                 tsb_slab_size = MMU_PAGESIZE512K;
1200                 tsb_slab_mask = MMU_PAGEOFFSET512K >> MMU_PAGESHIFT;
1201                 use_bigtsb_arena = 0;
1202         }

1204         if (!use_bigtsb_arena) {
1205                 bigtsb_slab_ttesz = tsb_slab_ttesz;
1206                 bigtsb_slab_shift = tsb_slab_shift;
1207                 bigtsb_slab_size = tsb_slab_size;
1208                 bigtsb_slab_mask = tsb_slab_mask;
1209         }


1212         /*
1213          * Set up memory callback to update tsb_alloc_hiwater and
1214          * tsb_max_growsize.
1215          */
1216         i = kphysm_setup_func_register(&sfmmu_update_vec, (void *) 0);
1217         ASSERT(i == 0);

1219         /*
```



```
1220          * kmem_tsb_arena is the source from which large TSB slabs are
1221          * drawn.  The quantum of this arena corresponds to the largest
1222          * TSB size we can dynamically allocate for user processes.
1223          * Currently it must also be a supported page size since we
1224          * use exactly one translation entry to map each slab page.
1225          *
1226          * The per-lgroup kmem_tsb_default_arena arenas are the arenas from
1227          * which most TSBs are allocated.  Since most TSB allocations are
1228          * typically 8K we have a kmem cache we stack on top of each
1229          * kmem_tsb_default_arena to speed up those allocations.
1230          *
1231          * Note the two-level scheme of arenas is required only
1232          * because vmem_create doesn't allow us to specify alignment
1233          * requirements.  If this ever changes the code could be
1234          * simplified to use only one level of arenas.
1235          *
1236          * If 256M page support exists on sun4v, 256MB kmem_bigtsb_arena
1237          * will be provided in addition to the 4M kmem_tsb_arena.
1238          */
1239         if (use_bigtsb_arena) {
1240                 kmem_bigtsb_arena = vmem_create("kmem_bigtsb", NULL, 0,
1241                     bigtsb_slab_size, sfmmu_vmem_xalloc_aligned_wrapper,
1242                     vmem_xfree, heap_arena, 0, VM_SLEEP);
1243         }

1245         kmem_tsb_arena = vmem_create("kmem_tsb", NULL, 0, tsb_slab_size,
1246             sfmmu_vmem_xalloc_aligned_wrapper,
1247             vmem_xfree, heap_arena, 0, VM_SLEEP);

1249         if (tsb_lgrp_affinity) {
1250                 char s[50];
1251                 for (i = 0; i < NLGRPS_MAX; i++) {
1252                         if (use_bigtsb_arena) {
1253                                 (void) sprintf(s, "kmem_bigtsb_lgrp%d", i);
1254                                 kmem_bigtsb_default_arena[i] = vmem_create(s,
1255                                     NULL, 0, 2 * tsb_slab_size,
1256                                     sfmmu_tsb_segkmem_alloc,
1257                                     sfmmu_tsb_segkmem_free, kmem_bigtsb_arena,
1258                                     0, VM_SLEEP | VM_BESTFIT);
1259                         }

1261                         (void) sprintf(s, "kmem_tsb_lgrp%d", i);
1262                         kmem_tsb_default_arena[i] = vmem_create(s,
1263                             NULL, 0, PAGESIZE, sfmmu_tsb_segkmem_alloc,
1264                             sfmmu_tsb_segkmem_free, kmem_tsb_arena, 0,
1265                             VM_SLEEP | VM_BESTFIT);

1267                         (void) sprintf(s, "sfmmu_tsb_lgrp%d_cache", i);
1268                         sfmmu_tsb_cache[i] = kmem_cache_create(s,
1269                             PAGESIZE, PAGESIZE, NULL, NULL, NULL, NULL,
1270                             kmem_tsb_default_arena[i], 0);
1271                 }
1272         } else {
1273                 if (use_bigtsb_arena) {
1274                         kmem_bigtsb_default_arena[0] =
1275                             vmem_create("kmem_bigtsb_default", NULL, 0,
1276                             2 * tsb_slab_size, sfmmu_tsb_segkmem_alloc,
1277                             sfmmu_tsb_segkmem_free, kmem_bigtsb_arena, 0,
1278                             VM_SLEEP | VM_BESTFIT);
1279                 }

1281                 kmem_tsb_default_arena[0] = vmem_create("kmem_tsb_default",
1282                     NULL, 0, PAGESIZE, sfmmu_tsb_segkmem_alloc,
1283                     sfmmu_tsb_segkmem_free, kmem_tsb_arena, 0,
1284                     VM_SLEEP | VM_BESTFIT);
1285                 sfmmu_tsb_cache[0] = kmem_cache_create("sfmmu_tsb_cache",
```

```
1286                     PAGESIZE, PAGESIZE, NULL, NULL, NULL, NULL,
1287                     kmem_tsb_default_arena[0], 0);
1288         }

1290         sfmmu8_cache = kmem_cache_create("sfmmu8_cache", HME8BLK_SZ,
1291             HMEBLK_ALIGN, sfmmu_hblkcache_constructor,
1292             sfmmu_hblkcache_destructor,
1293             sfmmu_hblkcache_reclaim, (void *)HME8BLK_SZ,
1294             hat_memload_arena, KMC_NOHASH);

1296         hat_memload1_arena = vmem_create("hat_memload1", NULL, 0, PAGESIZE,
1297             segkmem_alloc_permanent, segkmem_free, heap_arena, 0,
1298             VMC_DUMPSAFE | VM_SLEEP);

1300         sfmmu1_cache = kmem_cache_create("sfmmu1_cache", HME1BLK_SZ,
1301             HMEBLK_ALIGN, sfmmu_hblkcache_constructor,
1302             sfmmu_hblkcache_destructor,
1303             NULL, (void *)HME1BLK_SZ,
1304             hat_memload1_arena, KMC_NOHASH);

1306         pa_hment_cache = kmem_cache_create("pa_hment_cache", PAHME_SZ,
1307             0, NULL, NULL, NULL, NULL, static_arena, KMC_NOHASH);

1309         ism_blk_cache = kmem_cache_create("ism_blk_cache",
1310             sizeof (ism_blk_t), ecache_alignsize, NULL, NULL,
1311             NULL, NULL, static_arena, KMC_NOHASH);

1313         ism_ment_cache = kmem_cache_create("ism_ment_cache",
1314             sizeof (ism_ment_t), 0, NULL, NULL,
1315             NULL, NULL, NULL, 0);

1317         /*
1318          * We grab the first hat for the kernel,
1319          */
1320         AS_LOCK_ENTER(&kas, &kas.a_lock, RW_WRITER);
1321         kas.a_hat = hat_alloc(&kas);
1322         AS_LOCK_EXIT(&kas, &kas.a_lock);

1324         /*
1325          * Initialize hblk_reserve.
1326          */
1327         ((struct hme_blk *)hblk_reserve)->hblk_nextpa =
1328             va_to_pa((caddr_t)hblk_reserve);

1330 #ifndef UTSB_PHYS
1331         /*
1332          * Reserve some kernel virtual address space for the locked TTEs
1333          * that allow us to probe the TSB from TL>0.
1334          */
1335         utsb_vabase = vmem_xalloc(heap_arena, tsb_slab_size, tsb_slab_size,
1336             0, 0, NULL, NULL, VM_SLEEP);
1337         utsb4m_vabase = vmem_xalloc(heap_arena, tsb_slab_size, tsb_slab_size,
1338             0, 0, NULL, NULL, VM_SLEEP);
1339 #endif

1341 #ifdef VAC
1342         /*
1343          * The big page VAC handling code assumes VAC
1344          * will not be bigger than the smallest big
1345          * page- which is 64K.
1346          */
1347         if (TTEPAGES(TTE64K) < CACHE_NUM_COLOR) {
1348                 cmn_err(CE_PANIC, "VAC too big!");
1349         }
1350 #endif
```



```
1353         (void) xhat_init();

1352         uhme_hash_pa = va_to_pa(uhme_hash);
1353         khme_hash_pa = va_to_pa(khme_hash);

1355         /*
1356          * Initialize relocation locks. kpr_suspendlock is held
1357          * at PIL_MAX to prevent interrupts from pinning the holder
1358          * of a suspended TTE which may access it leading to a
1359          * deadlock condition.
1360          */
1361         mutex_init(&kpr_mutex, NULL, MUTEX_DEFAULT, NULL);
1362         mutex_init(&kpr_suspendlock, NULL, MUTEX_SPIN, (void *)PIL_MAX);

1364         /*
1365          * If Shared context support is disabled via /etc/system
1366          * set shctx_on to 0 here if it was set to 1 earlier in boot
1367          * sequence by cpu module initialization code.
1368          */
1369         if (shctx_on && disable_shctx) {
1370                 shctx_on = 0;
1371         }

1373         if (shctx_on) {
1374                 srd_buckets = kmem_zalloc(SFMMU_MAX_SRD_BUCKETS *
1375                     sizeof (srd_buckets[0]), KM_SLEEP);
1376                 for (i = 0; i < SFMMU_MAX_SRD_BUCKETS; i++) {
1377                         mutex_init(&srd_buckets[i].srdb_lock, NULL,
1378                             MUTEX_DEFAULT, NULL);
1379                 }

1381                 srd_cache = kmem_cache_create("srd_cache", sizeof (sf_srd_t),
1382                     0, sfmmu_srdcache_constructor, sfmmu_srdcache_destructor,
1383                     NULL, NULL, NULL, 0);
1384                 region_cache = kmem_cache_create("region_cache",
1385                     sizeof (sf_region_t), 0, sfmmu_rgncache_constructor,
1386                     sfmmu_rgncache_destructor, NULL, NULL, NULL, 0);
1387                 scd_cache = kmem_cache_create("scd_cache", sizeof (sf_scd_t),
1388                     0, sfmmu_scdcache_constructor,  sfmmu_scdcache_destructor,
1389                     NULL, NULL, NULL, 0);
1390         }

1392         /*
1393          * Pre-allocate hrm_hashtab before enabling the collection of
1394          * refmod statistics.  Allocating on the fly would mean us
1395          * running the risk of suffering recursive mutex enters or
1396          * deadlocks.
1397          */
1398         hrm_hashtab = kmem_zalloc(HRM_HASHSIZE * sizeof (struct hrmstat *),
1399             KM_SLEEP);

1401         /* Allocate per-cpu pending freelist of hmeblks */
1402         cpu_hme_pend = kmem_zalloc((NCPU * sizeof (cpu_hme_pend_t)) + 64,
1403             KM_SLEEP);
1404         cpu_hme_pend = (cpu_hme_pend_t *)P2ROUNDUP(
1405             (uintptr_t)cpu_hme_pend, 64);

1407         for (i = 0; i < NCPU; i++) {
1408                 mutex_init(&cpu_hme_pend[i].chp_mutex, NULL, MUTEX_DEFAULT,
1409                     NULL);
1410         }

1412         if (cpu_hme_pend_thresh == 0) {
1413                 cpu_hme_pend_thresh = CPU_HME_PEND_THRESH;
1414         }
1415 }
```

_____***unchanged_portion_omitted_***

```
#define SFMMU_KERNEL_MAXVA \
        (kmem64_base ? (uintptr_t)kmem64_end : (SYSLIMIT))

/*
 * Allocate a hat structure.
 * Called when an address space first uses a hat.
 */
struct hat *
hat_alloc(struct as *as)
{
        sfmmu_t *sfmmup;
        int i;
        uint64_t cnum;
        extern uint_t get_color_start(struct as *);

        ASSERT(AS_WRITE_HELD(as, &as->a_lock));
        sfmmup = kmem_cache_alloc(sfmmuid_cache, KM_SLEEP);
        sfmmup->sfmmu_as = as;
        sfmmup->sfmmu_flags = 0;
        sfmmup->sfmmu_tteflags = 0;
        sfmmup->sfmmu_rtteflags = 0;
        LOCK_INIT_CLEAR(&sfmmup->sfmmu_ctx_lock);

        if (as == &kas) {
                ksfmmup = sfmmup;
                sfmmup->sfmmu_cext = 0;
                cnum = KCONTEXT;

                sfmmup->sfmmu_clrstart = 0;
                sfmmup->sfmmu_tsb = NULL;
                /*
                 * hat_kern_setup() will call sfmmu_init_ktsbinfo()
                 * to setup tsb_info for ksfmmup.
                 */
        } else {

                /*
                 * Just set to invalid ctx. When it faults, it will
                 * get a valid ctx. This would avoid the situation
                 * where we get a ctx, but it gets stolen and then
                 * we fault when we try to run and so have to get
                 * another ctx.
                 */
                sfmmup->sfmmu_cext = 0;
                cnum = INVALID_CONTEXT;

                /* initialize original physical page coloring bin */
                sfmmup->sfmmu_clrstart = get_color_start(as);
#ifdef DEBUG
                if (tsb_random_size) {
                        uint32_t randval = (uint32_t)gettick() >> 4;
                        int size = randval % (tsb_max_growsize + 1);

                        /* chose a random tsb size for stress testing */
                        (void) sfmmu_tsbinfo_alloc(&sfmmup->sfmmu_tsb, size,
                            TSB8K|TSB64K|TSB512K, 0, sfmmup);
                } else
#endif /* DEBUG */
                        (void) sfmmu_tsbinfo_alloc(&sfmmup->sfmmu_tsb,
                            default_tsb_size,
                            TSB8K|TSB64K|TSB512K, 0, sfmmup);
                sfmmup->sfmmu_flags = HAT_SWAPPED | HAT_ALLCTX_INVALID;
                ASSERT(sfmmup->sfmmu_tsb != NULL);
        }
```



```
        ASSERT(max_mmu_ctxdoms > 0);
        for (i = 0; i < max_mmu_ctxdoms; i++) {
                sfmmup->sfmmu_ctxs[i].cnum = cnum;
                sfmmup->sfmmu_ctxs[i].gnum = 0;
        }

        for (i = 0; i < max_mmu_page_sizes; i++) {
                sfmmup->sfmmu_ttecnt[i] = 0;
                sfmmup->sfmmu_scdrttecnt[i] = 0;
                sfmmup->sfmmu_ismttecnt[i] = 0;
                sfmmup->sfmmu_scdismttecnt[i] = 0;
                sfmmup->sfmmu_pgsz[i] = TTE8K;
        }
        sfmmup->sfmmu_tsb0_4minflcnt = 0;
        sfmmup->sfmmu_iblk = NULL;
        sfmmup->sfmmu_ismhat = 0;
        sfmmup->sfmmu_scdhat = 0;
        sfmmup->sfmmu_ismblkpa = (uint64_t)-1;
        if (sfmmup == ksfmmup) {
                CPUSET_ALL(sfmmup->sfmmu_cpusran);
        } else {
                CPUSET_ZERO(sfmmup->sfmmu_cpusran);
        }
        sfmmup->sfmmu_free = 0;
        sfmmup->sfmmu_rmstat = 0;
        sfmmup->sfmmu_clrbin = sfmmup->sfmmu_clrstart;
        sfmmup->sfmmu_xhat_provider = NULL;
        cv_init(&sfmmup->sfmmu_tsb_cv, NULL, CV_DEFAULT, NULL);
        sfmmup->sfmmu_srdp = NULL;
        SF_RGNMAP_ZERO(sfmmup->sfmmu_region_map);
        bzero(sfmmup->sfmmu_hmeregion_links, SFMMU_L1_HMERLINKS_SIZE);
        sfmmup->sfmmu_scdp = NULL;
        sfmmup->sfmmu_scd_link.next = NULL;
        sfmmup->sfmmu_scd_link.prev = NULL;
        return (sfmmup);
}
______unchanged_portion_omitted_

/*
 * Free all the translation resources for the specified address space.
 * Called from as_free when an address space is being destroyed.
 */
void
hat_free_start(struct hat *sfmmup)
{
        ASSERT(AS_WRITE_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));
        ASSERT(sfmmup != ksfmmup);
        ASSERT(sfmmup->sfmmu_xhat_provider == NULL);

        sfmmup->sfmmu_free = 1;
        if (sfmmup->sfmmu_scdp != NULL) {
                sfmmu_leave_scd(sfmmup, 0);
        }

        ASSERT(sfmmup->sfmmu_scdp == NULL);
}

void
hat_free_end(struct hat *sfmmup)
{
        int i;

        ASSERT(sfmmup->sfmmu_xhat_provider == NULL);
        ASSERT(sfmmup->sfmmu_free == 1);
        ASSERT(sfmmup->sfmmu_ttecnt[TTE8K] == 0);
        ASSERT(sfmmup->sfmmu_ttecnt[TTE64K] == 0);
```

```
1935         ASSERT(sfmmup->sfmmu_ttecnt[TTE512K] == 0);
1936         ASSERT(sfmmup->sfmmu_ttecnt[TTE4M] == 0);
1937         ASSERT(sfmmup->sfmmu_ttecnt[TTE32M] == 0);
1938         ASSERT(sfmmup->sfmmu_ttecnt[TTE256M] == 0);

1940         if (sfmmup->sfmmu_rmstat) {
1941                 hat_freestat(sfmmup->sfmmu_as, NULL);
1942         }

1944         while (sfmmup->sfmmu_tsb != NULL) {
1945                 struct tsb_info *next = sfmmup->sfmmu_tsb->tsb_next;
1946                 sfmmu_tsbinfo_free(sfmmup->sfmmu_tsb);
1947                 sfmmup->sfmmu_tsb = next;
1948         }

1950         if (sfmmup->sfmmu_srdp != NULL) {
1951                 sfmmu_leave_srd(sfmmup);
1952                 ASSERT(sfmmup->sfmmu_srdp == NULL);
1953                 for (i = 0; i < SFMMU_L1_HMERLINKS; i++) {
1954                         if (sfmmup->sfmmu_hmeregion_links[i] != NULL) {
1955                                 kmem_free(sfmmup->sfmmu_hmeregion_links[i],
1956                                     SFMMU_L2_HMERLINKS_SIZE);
1957                                 sfmmup->sfmmu_hmeregion_links[i] = NULL;
1958                         }
1959                 }
1960         }
1961         sfmmu_free_sfmmu(sfmmup);

1963 #ifdef DEBUG
1964         for (i = 0; i < SFMMU_L1_HMERLINKS; i++) {
1965                 ASSERT(sfmmup->sfmmu_hmeregion_links[i] == NULL);
1966         }
1967 #endif

1969         kmem_cache_free(sfmmuid_cache, sfmmup);
1970 }

1972 /*
1979  * Set up any translation structures, for the specified address space,
1980  * that are needed or preferred when the process is being swapped in.
1981  */
1982 /* ARGSUSED */
1983 void
1984 hat_swapin(struct hat *hat)
1985 {
1986         ASSERT(hat->sfmmu_xhat_provider == NULL);
1987 }

1989 /*
1990  * Free all of the translation resources, for the specified address space,
1991  * that can be freed while the process is swapped out. Called from as_swapout.
1992  * Also, free up the ctx that this process was using.
1993  */
1994 void
1995 hat_swapout(struct hat *sfmmup)
1996 {
1997         struct hmehash_bucket *hmebp;
1998         struct hme_blk *hmeblkp;
1999         struct hme_blk *pr_hblk = NULL;
2000         struct hme_blk *nx_hblk;
2001         int i;
2002         struct hme_blk *list = NULL;
2003         hatlock_t *hatlockp;
2004         struct tsb_info *tsbinfop;
2005         struct free_tsb {
2006                 struct free_tsb *next;
```



```
2007                 struct tsb_info *tsbinfop;
2008         };                      /* free list of TSBs */
2009         struct free_tsb *freelist, *last, *next;

2011         ASSERT(sfmmup->sfmmu_xhat_provider == NULL);
2012         SFMMU_STAT(sf_swapout);

2014         /*
2015          * There is no way to go from an as to all its translations in sfmmu.
2016          * Here is one of the times when we take the big hit and traverse
2017          * the hash looking for hme_blks to free up.  Not only do we free up
2018          * this as hme_blks but all those that are free.  We are obviously
2019          * swapping because we need memory so let's free up as much
2020          * as we can.
2021          *
2022          * Note that we don't flush TLB/TSB here -- it's not necessary
2023          * because:
2024          *  1) we free the ctx we're using and throw away the TSB(s);
2025          *  2) processes aren't runnable while being swapped out.
2026          */
2027         ASSERT(sfmmup != KHATID);
2028         for (i = 0; i <= UHMEHASH_SZ; i++) {
2029                 hmebp = &uhme_hash[i];
2030                 SFMMU_HASH_LOCK(hmebp);
2031                 hmeblkp = hmebp->hmeblkp;
2032                 pr_hblk = NULL;
2033                 while (hmeblkp) {

2035                         ASSERT(!hmeblkp->hblk_xhat_bit);

2037                         if ((hmeblkp->hblk_tag.htag_id == sfmmup) &&
2038                             !hmeblkp->hblk_shw_bit && !hmeblkp->hblk_lckcnt) {
2039                                 ASSERT(!hmeblkp->hblk_shared);
2040                                 (void) sfmmu_hblk_unload(sfmmup, hmeblkp,
2041                                     (caddr_t)get_hblk_base(hmeblkp),
2042                                     get_hblk_endaddr(hmeblkp),
2043                                     NULL, HAT_UNLOAD);
2044                         }
2045                         nx_hblk = hmeblkp->hblk_next;
2046                         if (!hmeblkp->hblk_vcnt && !hmeblkp->hblk_hmecnt) {
2047                                 ASSERT(!hmeblkp->hblk_lckcnt);
2048                                 sfmmu_hblk_hash_rm(hmebp, hmeblkp, pr_hblk,
2049                                     &list, 0);
2050                         } else {
2051                                 pr_hblk = hmeblkp;
2052                         }
2053                         hmeblkp = nx_hblk;
2054                 }
2055                 SFMMU_HASH_UNLOCK(hmebp);
2056         }

2058         sfmmu_hblks_list_purge(&list, 0);

2060         /*
2061          * Now free up the ctx so that others can reuse it.
2062          */
2063         hatlockp = sfmmu_hat_enter(sfmmup);

2065         sfmmu_invalidate_ctx(sfmmup);

2067         /*
2068          * Free TSBs, but not tsbinfos, and set SWAPPED flag.
2069          * If TSBs were never swapped in, just return.
2070          * This implies that we don't support partial swapping
2071          * of TSBs -- either all are swapped out, or none are.
2072          *
```

```
2073          * We must hold the HAT lock here to prevent racing with another
2074          * thread trying to unmap TTEs from the TSB or running the post-
2075          * relocator after relocating the TSB's memory.  Unfortunately, we
2076          * can't free memory while holding the HAT lock or we could
2077          * deadlock, so we build a list of TSBs to be freed after marking
2078          * the tsbinfos as swapped out and free them after dropping the
2079          * lock.
2080          */
2081         if (SFMMU_FLAGS_ISSET(sfmmup, HAT_SWAPPED)) {
2082                 sfmmu_hat_exit(hatlockp);
2083                 return;
2084         }

2086         SFMMU_FLAGS_SET(sfmmup, HAT_SWAPPED);
2087         last = freelist = NULL;
2088         for (tsbinfop = sfmmup->sfmmu_tsb; tsbinfop != NULL;
2089             tsbinfop = tsbinfop->tsb_next) {
2090                 ASSERT((tsbinfop->tsb_flags & TSB_SWAPPED) == 0);

2092                 /*
2093                  * Cast the TSB into a struct free_tsb and put it on the free
2094                  * list.
2095                  */
2096                 if (freelist == NULL) {
2097                         last = freelist = (struct free_tsb *)tsbinfop->tsb_va;
2098                 } else {
2099                         last->next = (struct free_tsb *)tsbinfop->tsb_va;
2100                         last = last->next;
2101                 }
2102                 last->next = NULL;
2103                 last->tsbinfop = tsbinfop;
2104                 tsbinfop->tsb_flags |= TSB_SWAPPED;
2105                 /*
2106                  * Zero out the TTE to clear the valid bit.
2107                  * Note we can't use a value like 0xbad because we want to
2108                  * ensure diagnostic bits are NEVER set on TTEs that might
2109                  * be loaded.  The intent is to catch any invalid access
2110                  * to the swapped TSB, such as a thread running with a valid
2111                  * context without first calling sfmmu_tsb_swapin() to
2112                  * allocate TSB memory.
2113                  */
2114                 tsbinfop->tsb_tte.ll = 0;
2115         }

2117         /* Now we can drop the lock and free the TSB memory. */
2118         sfmmu_hat_exit(hatlockp);
2119         for (; freelist != NULL; freelist = next) {
2120                 next = freelist->next;
2121                 sfmmu_tsb_free(freelist->tsbinfop);
2122         }
2123 }

2125 /*
1973  * Duplicate the translations of an as into another newas
1974  */
1975 /* ARGSUSED */
1976 int
1977 hat_dup(struct hat *hat, struct hat *newhat, caddr_t addr, size_t len,
1978         uint_t flag)
1979 {
1980         sf_srd_t *srdp;
1981         sf_scd_t *scdp;
1982         int i;
1983         extern uint_t get_color_start(struct as *);

2138         ASSERT(hat->sfmmu_xhat_provider == NULL);
```



```
1985         ASSERT((flag == 0) || (flag == HAT_DUP_ALL) || (flag == HAT_DUP_COW) ||
1986             (flag == HAT_DUP_SRD));
1987         ASSERT(hat != ksfmmup);
1988         ASSERT(newhat != ksfmmup);
1989         ASSERT(flag != HAT_DUP_ALL || hat->sfmmu_srdp == newhat->sfmmu_srdp);

1991         if (flag == HAT_DUP_COW) {
1992                 panic("hat_dup: HAT_DUP_COW not supported");
1993         }

1995         if (flag == HAT_DUP_SRD && ((srdp = hat->sfmmu_srdp) != NULL)) {
1996                 ASSERT(srdp->srd_evp != NULL);
1997                 VN_HOLD(srdp->srd_evp);
1998                 ASSERT(srdp->srd_refcnt > 0);
1999                 newhat->sfmmu_srdp = srdp;
2000                 atomic_inc_32((volatile uint_t *)&srdp->srd_refcnt);
2001         }

2003         /*
2004          * HAT_DUP_ALL flag is used after as duplication is done.
2005          */
2006         if (flag == HAT_DUP_ALL && ((srdp = newhat->sfmmu_srdp) != NULL)) {
2007                 ASSERT(newhat->sfmmu_srdp->srd_refcnt >= 2);
2008                 newhat->sfmmu_rtteflags = hat->sfmmu_rtteflags;
2009                 if (hat->sfmmu_flags & HAT_4MTEXT_FLAG) {
2010                         newhat->sfmmu_flags |= HAT_4MTEXT_FLAG;
2011                 }

2013                 /* check if need to join scd */
2014                 if ((scdp = hat->sfmmu_scdp) != NULL &&
2015                     newhat->sfmmu_scdp != scdp) {
2016                         int ret;
2017                         SF_RGNMAP_IS_SUBSET(&newhat->sfmmu_region_map,
2018                             &scdp->scd_region_map, ret);
2019                         ASSERT(ret);
2020                         sfmmu_join_scd(scdp, newhat);
2021                         ASSERT(newhat->sfmmu_scdp == scdp &&
2022                             scdp->scd_refcnt >= 2);
2023                         for (i = 0; i < max_mmu_page_sizes; i++) {
2024                                 newhat->sfmmu_ismttecnt[i] =
2025                                     hat->sfmmu_ismttecnt[i];
2026                                 newhat->sfmmu_scdismttecnt[i] =
2027                                     hat->sfmmu_scdismttecnt[i];
2028                         }
2029                 }

2031                 sfmmu_check_page_sizes(newhat, 1);
2032         }

2034         if (flag == HAT_DUP_ALL && consistent_coloring == 0 &&
2035             update_proc_pgcolorbase_after_fork != 0) {
2036                 hat->sfmmu_clrbin = get_color_start(hat->sfmmu_as);
2037         }
2038         return (0);
2039 }
______unchanged_portion_omitted_

2049 void
2050 hat_memload_region(struct hat *hat, caddr_t addr, struct page *pp,
2051         uint_t attr, uint_t flags, hat_region_cookie_t rcookie)
2052 {
2053         uint_t rid;
2054         if (rcookie == HAT_INVALID_REGION_COOKIE) {
2208         if (rcookie == HAT_INVALID_REGION_COOKIE ||
2209             hat->sfmmu_xhat_provider != NULL) {
2055                 hat_do_memload(hat, addr, pp, attr, flags,
```

```
2056                    SFMMU_INVALID_SHMERID);
2057                return;
2058        }
2059        rid = (uint_t)((uint64_t)rcookie);
2060        ASSERT(rid < SFMMU_MAX_HME_REGIONS);
2061        hat_do_memload(hat, addr, pp, attr, flags, rid);
2062 }

2064 /*
2065  * Set up addr to map to page pp with protection prot.
2066  * As an optimization we also load the TSB with the
2067  * corresponding tte but it is no big deal if  the tte gets kicked out.
2068  */
2069 static void
2070 hat_do_memload(struct hat *hat, caddr_t addr, struct page *pp,
2071        uint_t attr, uint_t flags, uint_t rid)
2072 {
2073        tte_t tte;


2076        ASSERT(hat != NULL);
2077        ASSERT(PAGE_LOCKED(pp));
2078        ASSERT(!((uintptr_t)addr & MMU_PAGEOFFSET));
2079        ASSERT(!(flags & ~SFMMU_LOAD_ALLFLAG));
2080        ASSERT(!(attr & ~SFMMU_LOAD_ALLATTR));
2081        SFMMU_VALIDATE_HMERID(hat, rid, addr, MMU_PAGESIZE);

2083        if (PP_ISFREE(pp)) {
2084                panic("hat_memload: loading a mapping to free page %p",
2085                    (void *)pp);
2086        }

2243        if (hat->sfmmu_xhat_provider) {
2244                /* no regions for xhats */
2245                ASSERT(!SFMMU_IS_SHMERID_VALID(rid));
2246                XHAT_MEMLOAD(hat, addr, pp, attr, flags);
2247                return;
2248        }

2088        ASSERT((hat == ksfmmup) ||
2089            AS_LOCK_HELD(hat->sfmmu_as, &hat->sfmmu_as->a_lock));

2091        if (flags & ~SFMMU_LOAD_ALLFLAG)
2092                cmn_err(CE_NOTE, "hat_memload: unsupported flags %d",
2093                    flags & ~SFMMU_LOAD_ALLFLAG);

2095        if (hat->sfmmu_rmstat)
2096                hat_resvstat(MMU_PAGESIZE, hat->sfmmu_as, addr);

2098 #if defined(SF_ERRATA_57)
2099        if ((hat != ksfmmup) && AS_TYPE_64BIT(hat->sfmmu_as) &&
2100            (addr < errata57_limit) && (attr & PROT_EXEC) &&
2101            !(flags & HAT_LOAD_SHARE)) {
2102                cmn_err(CE_WARN, "hat_memload: illegal attempt to make user "
2103                    " page executable");
2104                attr &= ~PROT_EXEC;
2105        }
2106 #endif

2108        sfmmu_memtte(&tte, pp->p_pagenum, attr, TTE8K);
2109        (void) sfmmu_tteload_array(hat, &tte, addr, &pp, flags, rid);

2111        /*
2112         * Check TSB and TLB page sizes.
2113         */
2114        if ((flags & HAT_LOAD_SHARE) == 0) {
```



```
2115                sfmmu_check_page_sizes(hat, 1);
2116        }
2117 }

2119 /*
2120  * hat_devload can be called to map real memory (e.g.
2121  * /dev/kmem) and even though hat_devload will determine pf is
2122  * for memory, it will be unable to get a shared lock on the
2123  * page (because someone else has it exclusively) and will
2124  * pass dp = NULL.  If tteload doesn't get a non-NULL
2125  * page pointer it can't cache memory.
2126  */
2127 void
2128 hat_devload(struct hat *hat, caddr_t addr, size_t len, pfn_t pfn,
2129        uint_t attr, int flags)
2130 {
2131        tte_t tte;
2132        struct page *pp = NULL;
2133        int use_lgpg = 0;

2135        ASSERT(hat != NULL);

2299        if (hat->sfmmu_xhat_provider) {
2300                XHAT_DEVLOAD(hat, addr, len, pfn, attr, flags);
2301                return;
2302        }

2137        ASSERT(!(flags & ~SFMMU_LOAD_ALLFLAG));
2138        ASSERT(!(attr & ~SFMMU_LOAD_ALLATTR));
2139        ASSERT((hat == ksfmmup) ||
2140            AS_LOCK_HELD(hat->sfmmu_as, &hat->sfmmu_as->a_lock));
2141        if (len == 0)
2142                panic("hat_devload: zero len");
2143        if (flags & ~SFMMU_LOAD_ALLFLAG)
2144                cmn_err(CE_NOTE, "hat_devload: unsupported flags %d",
2145                    flags & ~SFMMU_LOAD_ALLFLAG);

2147 #if defined(SF_ERRATA_57)
2148        if ((hat != ksfmmup) && AS_TYPE_64BIT(hat->sfmmu_as) &&
2149            (addr < errata57_limit) && (attr & PROT_EXEC) &&
2150            !(flags & HAT_LOAD_SHARE)) {
2151                cmn_err(CE_WARN, "hat_devload: illegal attempt to make user "
2152                    " page executable");
2153                attr &= ~PROT_EXEC;
2154        }
2155 #endif

2157        /*
2158         * If it's a memory page find its pp
2159         */
2160        if (!(flags & HAT_LOAD_NOCONSIST) && pf_is_memory(pfn)) {
2161                pp = page_numtopp_nolock(pfn);
2162                if (pp == NULL) {
2163                        flags |= HAT_LOAD_NOCONSIST;
2164                } else {
2165                        if (PP_ISFREE(pp)) {
2166                                panic("hat_memload: loading "
2167                                    "a mapping to free page %p",
2168                                    (void *)pp);
2169                        }
2170                        if (!PAGE_LOCKED(pp) && !PP_ISNORELOC(pp)) {
2171                                panic("hat_memload: loading a mapping "
2172                                    "to unlocked relocatable page %p",
2173                                    (void *)pp);
2174                        }
2175                        ASSERT(len == MMU_PAGESIZE);
```

```
2176                }
2177        }

2179        if (hat->sfmmu_rmstat)
2180                hat_resvstat(len, hat->sfmmu_as, addr);

2182        if (flags & HAT_LOAD_NOCONSIST) {
2183                attr |= SFMMU_UNCACHEVTTE;
2184                use_lgpg = 1;
2185        }
2186        if (!pf_is_memory(pfn)) {
2187                attr |= SFMMU_UNCACHEPTTE | HAT_NOSYNC;
2188                use_lgpg = 1;
2189                switch (attr & HAT_ORDER_MASK) {
2190                        case HAT_STRICTORDER:
2191                        case HAT_UNORDERED_OK:
2192                                /*
2193                                 * we set the side effect bit for all non
2194                                 * memory mappings unless merging is ok
2195                                 */
2196                                attr |= SFMMU_SIDEFFECT;
2197                                break;
2198                        case HAT_MERGING_OK:
2199                        case HAT_LOADCACHING_OK:
2200                        case HAT_STORECACHING_OK:
2201                                break;
2202                        default:
2203                                panic("hat_devload: bad attr");
2204                                break;
2205                }
2206        }
2207        while (len) {
2208                if (!use_lgpg) {
2209                        sfmmu_memtte(&tte, pfn, attr, TTE8K);
2210                        (void) sfmmu_tteload_array(hat, &tte, addr, &pp,
2211                            flags, SFMMU_INVALID_SHMERID);
2212                        len -= MMU_PAGESIZE;
2213                        addr += MMU_PAGESIZE;
2214                        pfn++;
2215                        continue;
2216                }
2217                /*
2218                 *  try to use large pages, check va/pa alignments
2219                 *  Note that 32M/256M page sizes are not (yet) supported.
2220                 */
2221                if ((len >= MMU_PAGESIZE4M) &&
2222                    !((uintptr_t)addr & MMU_PAGEOFFSET4M) &&
2223                    !(disable_large_pages & (1 << TTE4M)) &&
2224                    !(mmu_ptob(pfn) & MMU_PAGEOFFSET4M)) {
2225                        sfmmu_memtte(&tte, pfn, attr, TTE4M);
2226                        (void) sfmmu_tteload_array(hat, &tte, addr, &pp,
2227                            flags, SFMMU_INVALID_SHMERID);
2228                        len -= MMU_PAGESIZE4M;
2229                        addr += MMU_PAGESIZE4M;
2230                        pfn += MMU_PAGESIZE4M / MMU_PAGESIZE;
2231                } else if ((len >= MMU_PAGESIZE512K) &&
2232                    !((uintptr_t)addr & MMU_PAGEOFFSET512K) &&
2233                    !(disable_large_pages & (1 << TTE512K)) &&
2234                    !(mmu_ptob(pfn) & MMU_PAGEOFFSET512K)) {
2235                        sfmmu_memtte(&tte, pfn, attr, TTE512K);
2236                        (void) sfmmu_tteload_array(hat, &tte, addr, &pp,
2237                            flags, SFMMU_INVALID_SHMERID);
2238                        len -= MMU_PAGESIZE512K;
2239                        addr += MMU_PAGESIZE512K;
2240                        pfn += MMU_PAGESIZE512K / MMU_PAGESIZE;
2241                } else if ((len >= MMU_PAGESIZE64K) &&
```



```
2242                    !((uintptr_t)addr & MMU_PAGEOFFSET64K) &&
2243                    !(disable_large_pages & (1 << TTE64K)) &&
2244                    !(mmu_ptob(pfn) & MMU_PAGEOFFSET64K)) {
2245                        sfmmu_memtte(&tte, pfn, attr, TTE64K);
2246                        (void) sfmmu_tteload_array(hat, &tte, addr, &pp,
2247                            flags, SFMMU_INVALID_SHMERID);
2248                        len -= MMU_PAGESIZE64K;
2249                        addr += MMU_PAGESIZE64K;
2250                        pfn += MMU_PAGESIZE64K / MMU_PAGESIZE;
2251                } else {
2252                        sfmmu_memtte(&tte, pfn, attr, TTE8K);
2253                        (void) sfmmu_tteload_array(hat, &tte, addr, &pp,
2254                            flags, SFMMU_INVALID_SHMERID);
2255                        len -= MMU_PAGESIZE;
2256                        addr += MMU_PAGESIZE;
2257                        pfn++;
2258                }
2259        }

2261        /*
2262         * Check TSB and TLB page sizes.
2263         */
2264        if ((flags & HAT_LOAD_SHARE) == 0) {
2265                sfmmu_check_page_sizes(hat, 1);
2266        }
2267 }
_____unchanged_portion_omitted_

2277 void
2278 hat_memload_array_region(struct hat *hat, caddr_t addr, size_t len,
2279         struct page **pps, uint_t attr, uint_t flags,
2280         hat_region_cookie_t rcookie)
2281 {
2282         uint_t rid;
2283         if (rcookie == HAT_INVALID_REGION_COOKIE) {
2450         if (rcookie == HAT_INVALID_REGION_COOKIE ||
2451             hat->sfmmu_xhat_provider != NULL) {
2284                 hat_do_memload_array(hat, addr, len, pps, attr, flags,
2285                     SFMMU_INVALID_SHMERID);
2286                 return;
2287         }
2288         rid = (uint_t)((uint64_t)rcookie);
2289         ASSERT(rid < SFMMU_MAX_HME_REGIONS);
2290         hat_do_memload_array(hat, addr, len, pps, attr, flags, rid);
2291 }

2293 /*
2294  * Map the largest extend possible out of the page array. The array may NOT
2295  * be in order.  The largest possible mapping a page can have
2296  * is specified in the p_szc field.  The p_szc field
2297  * cannot change as long as there any mappings (large or small)
2298  * to any of the pages that make up the large page. (ie. any
2299  * promotion/demotion of page size is not up to the hat but up to
2300  * the page free list manager).  The array
2301  * should consist of properly aligned contigous pages that are
2302  * part of a big page for a large mapping to be created.
2303  */
2304 static void
2305 hat_do_memload_array(struct hat *hat, caddr_t addr, size_t len,
2306         struct page **pps, uint_t attr, uint_t flags, uint_t rid)
2307 {
2308         int  ttesz;
2309         size_t mapsz;
2310         pgcnt_t numpg, npgs;
2311         tte_t tte;
2312         page_t *pp;
```

```
2313         uint_t large_pages_disable;

2315         ASSERT(!((uintptr_t)addr & MMU_PAGEOFFSET));
2316         SFMMU_VALIDATE_HMERID(hat, rid, addr, len);

2486         if (hat->sfmmu_xhat_provider) {
2487                 ASSERT(!SFMMU_IS_SHMERID_VALID(rid));
2488                 XHAT_MEMLOAD_ARRAY(hat, addr, len, pps, attr, flags);
2489                 return;
2490         }

2318         if (hat->sfmmu_rmstat)
2319                 hat_resvstat(len, hat->sfmmu_as, addr);

2321 #if defined(SF_ERRATA_57)
2322         if ((hat != ksfmmup) && AS_TYPE_64BIT(hat->sfmmu_as) &&
2323             (addr < errata57_limit) && (attr & PROT_EXEC) &&
2324             !(flags & HAT_LOAD_SHARE)) {
2325                 cmn_err(CE_WARN, "hat_memload_array: illegal attempt to make "
2326                     "user page executable");
2327                 attr &= ~PROT_EXEC;
2328         }
2329 #endif

2331         /* Get number of pages */
2332         npgs = len >> MMU_PAGESHIFT;

2334         if (flags & HAT_LOAD_SHARE) {
2335                 large_pages_disable = disable_ism_large_pages;
2336         } else {
2337                 large_pages_disable = disable_large_pages;
2338         }

2340         if (npgs < NHMENTS || large_pages_disable == LARGE_PAGES_OFF) {
2341                 sfmmu_memload_batchsmall(hat, addr, pps, attr, flags, npgs,
2342                     rid);
2343                 return;
2344         }

2346         while (npgs >= NHMENTS) {
2347                 pp = *pps;
2348                 for (ttesz = pp->p_szc; ttesz != TTE8K; ttesz--) {
2349                         /*
2350                          * Check if this page size is disabled.
2351                          */
2352                         if (large_pages_disable & (1 << ttesz))
2353                                 continue;

2355                         numpg = TTEPAGES(ttesz);
2356                         mapsz = numpg << MMU_PAGESHIFT;
2357                         if ((npgs >= numpg) &&
2358                             IS_P2ALIGNED(addr, mapsz) &&
2359                             IS_P2ALIGNED(pp->p_pagenum, numpg)) {
2360                                 /*
2361                                  * At this point we have enough pages and
2362                                  * we know the virtual address and the pfn
2363                                  * are properly aligned.  We still need
2364                                  * to check for physical contiguity but since
2365                                  * it is very likely that this is the case
2366                                  * we will assume they are so and undo
2367                                  * the request if necessary.  It would
2368                                  * be great if we could get a hint flag
2369                                  * like HAT_CONTIG which would tell us
2370                                  * the pages are contigous for sure.
2371                                  */
2372                                 sfmmu_memtte(&tte, (*pps)->p_pagenum,
```



```
2373                                     attr, ttesz);
2374                                 if (!sfmmu_tteload_array(hat, &tte, addr,
2375                                     pps, flags, rid)) {
2376                                         break;
2377                                 }
2378                         }
2379                 }
2380                 if (ttesz == TTE8K) {
2381                         /*
2382                          * We were not able to map array using a large page
2383                          * batch a hmeblk or fraction at a time.
2384                          */
2385                         numpg = ((uintptr_t)addr >> MMU_PAGESHIFT)
2386                             & (NHMENTS-1);
2387                         numpg = NHMENTS - numpg;
2388                         ASSERT(numpg <= npgs);
2389                         mapsz = numpg * MMU_PAGESIZE;
2390                         sfmmu_memload_batchsmall(hat, addr, pps, attr, flags,
2391                             numpg, rid);
2392                 }
2393                 addr += mapsz;
2394                 npgs -= numpg;
2395                 pps += numpg;
2396         }

2398         if (npgs) {
2399                 sfmmu_memload_batchsmall(hat, addr, pps, attr, flags, npgs,
2400                     rid);
2401         }

2403         /*
2404          * Check TSB and TLB page sizes.
2405          */
2406         if ((flags & HAT_LOAD_SHARE) == 0) {
2407                 sfmmu_check_page_sizes(hat, 1);
2408         }
2409 }
```

_____***unchanged_portion_omitted_***

```
3787 /*
3788  * Release one hardware address translation lock on the given address range.
3789  */
3790 void
3791 hat_unlock(struct hat *sfmmup, caddr_t addr, size_t len)
3792 {
3793         struct hmehash_bucket *hmebp;
3794         hmeblk_tag hblktag;
3795         int hmeshift, hashno = 1;
3796         struct hme_blk *hmeblkp, *list = NULL;
3797         caddr_t endaddr;

3799         ASSERT(sfmmup != NULL);
3974         ASSERT(sfmmup->sfmmu_xhat_provider == NULL);

3801         ASSERT((sfmmup == ksfmmup) ||
3802             AS_LOCK_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));
3803         ASSERT((len & MMU_PAGEOFFSET) == 0);
3804         endaddr = addr + len;
3805         hblktag.htag_id = sfmmup;
3806         hblktag.htag_rid = SFMMU_INVALID_SHMERID;

3808         /*
3809          * Spitfire supports 4 page sizes.
3810          * Most pages are expected to be of the smallest page size (8K) and
3811          * these will not need to be rehashed. 64K pages also don't need to be
3812          * rehashed because an hmeblk spans 64K of address space. 512K pages
```

```
3813          * might need 1 rehash and and 4M pages might need 2 rehashes.
3814          */
3815         while (addr < endaddr) {
3816                 hmeshift = HME_HASH_SHIFT(hashno);
3817                 hblktag.htag_bspage = HME_HASH_BSPAGE(addr, hmeshift);
3818                 hblktag.htag_rehash = hashno;
3819                 hmebp = HME_HASH_FUNCTION(sfmmup, addr, hmeshift);

3821                 SFMMU_HASH_LOCK(hmebp);

3823                 HME_HASH_SEARCH(hmebp, hblktag, hmeblkp, &list);
3824                 if (hmeblkp != NULL) {
3825                         ASSERT(!hmeblkp->hblk_shared);
3826                         /*
3827                          * If we encounter a shadow hmeblk then
3828                          * we know there are no valid hmeblks mapping
3829                          * this address at this size or larger.
3830                          * Just increment address by the smallest
3831                          * page size.
3832                          */
3833                         if (hmeblkp->hblk_shw_bit) {
3834                                 addr += MMU_PAGESIZE;
3835                         } else {
3836                                 addr = sfmmu_hblk_unlock(hmeblkp, addr,
3837                                     endaddr);
3838                         }
3839                         SFMMU_HASH_UNLOCK(hmebp);
3840                         hashno = 1;
3841                         continue;
3842                 }
3843                 SFMMU_HASH_UNLOCK(hmebp);

3845                 if (!HME_REHASH(sfmmup) || (hashno >= mmu_hashcnt)) {
3846                         /*
3847                          * We have traversed the whole list and rehashed
3848                          * if necessary without finding the address to unlock
3849                          * which should never happen.
3850                          */
3851                         panic("sfmmu_unlock: addr not found. "
3852                             "addr %p hat %p", (void *)addr, (void *)sfmmup);
3853                 } else {
3854                         hashno++;
3855                 }
3856         }

3858         sfmmu_hblks_list_purge(&list, 0);
3859 }

3861 void
3862 hat_unlock_region(struct hat *sfmmup, caddr_t addr, size_t len,
3863     hat_region_cookie_t rcookie)
3864 {
3865         sf_srd_t *srdp;
3866         sf_region_t *rgnp;
3867         int ttesz;
3868         uint_t rid;
3869         caddr_t eaddr;
3870         caddr_t va;
3871         int hmeshift;
3872         hmeblk_tag hblktag;
3873         struct hmehash_bucket *hmebp;
3874         struct hme_blk *hmeblkp;
3875         struct hme_blk *pr_hblk;
3876         struct hme_blk *list;

3878         if (rcookie == HAT_INVALID_REGION_COOKIE) {
```



```
3879                 hat_unlock(sfmmup, addr, len);
3880                 return;
3881         }

3883         ASSERT(sfmmup != NULL);
4059         ASSERT(sfmmup->sfmmu_xhat_provider == NULL);
3884         ASSERT(sfmmup != ksfmmup);

3886         srdp = sfmmup->sfmmu_srdp;
3887         rid = (uint_t)((uint64_t)rcookie);
3888         VERIFY3U(rid, <, SFMMU_MAX_HME_REGIONS);
3889         eaddr = addr + len;
3890         va = addr;
3891         list = NULL;
3892         rgnp = srdp->srd_hmergnp[rid];
3893         SFMMU_VALIDATE_HMERID(sfmmup, rid, addr, len);

3895         ASSERT(IS_P2ALIGNED(addr, TTEBYTES(rgnp->rgn_pgszc)));
3896         ASSERT(IS_P2ALIGNED(len, TTEBYTES(rgnp->rgn_pgszc)));
3897         if (rgnp->rgn_pgszc < HBLK_MIN_TTESZ) {
3898                 ttesz = HBLK_MIN_TTESZ;
3899         } else {
3900                 ttesz = rgnp->rgn_pgszc;
3901         }
3902         while (va < eaddr) {
3903                 while (ttesz < rgnp->rgn_pgszc &&
3904                     IS_P2ALIGNED(va, TTEBYTES(ttesz + 1))) {
3905                         ttesz++;
3906                 }
3907                 while (ttesz >= HBLK_MIN_TTESZ) {
3908                         if (!(rgnp->rgn_hmeflags & (1 << ttesz))) {
3909                                 ttesz--;
3910                                 continue;
3911                         }
3912                         hmeshift = HME_HASH_SHIFT(ttesz);
3913                         hblktag.htag_bspage = HME_HASH_BSPAGE(va, hmeshift);
3914                         hblktag.htag_rehash = ttesz;
3915                         hblktag.htag_rid = rid;
3916                         hblktag.htag_id = srdp;
3917                         hmebp = HME_HASH_FUNCTION(srdp, va, hmeshift);
3918                         SFMMU_HASH_LOCK(hmebp);
3919                         HME_HASH_SEARCH_PREV(hmebp, hblktag, hmeblkp, pr_hblk,
3920                             &list);
3921                         if (hmeblkp == NULL) {
3922                                 SFMMU_HASH_UNLOCK(hmebp);
3923                                 ttesz--;
3924                                 continue;
3925                         }
3926                         ASSERT(hmeblkp->hblk_shared);
3927                         va = sfmmu_hblk_unlock(hmeblkp, va, eaddr);
3928                         ASSERT(va >= eaddr ||
3929                             IS_P2ALIGNED((uintptr_t)va, TTEBYTES(ttesz)));
3930                         SFMMU_HASH_UNLOCK(hmebp);
3931                         break;
3932                 }
3933                 if (ttesz < HBLK_MIN_TTESZ) {
3934                         panic("hat_unlock_region: addr not found "
3935                             "addr %p hat %p", (void *)va, (void *)sfmmup);
3936                 }
3937         }
3938         sfmmu_hblks_list_purge(&list, 0);
3939 }
______unchanged_portion_omitted_

4578 /*
4579  * hat_probe returns 1 if the translation for the address 'addr' is
```

```
4580  * loaded, zero otherwise.
4581  *
4582  * hat_probe should be used only for advisorary purposes because it may
4583  * occasionally return the wrong value. The implementation must guarantee that
4584  * returning the wrong value is a very rare event. hat_probe is used
4585  * to implement optimizations in the segment drivers.
4586  *
4587  */
4588 int
4589 hat_probe(struct hat *sfmmup, caddr_t addr)
4590 {
4591         pfn_t pfn;
4592         tte_t tte;

4594         ASSERT(sfmmup != NULL);
4771         ASSERT(sfmmup->sfmmu_xhat_provider == NULL);

4596         ASSERT((sfmmup == ksfmmup) ||
4597             AS_LOCK_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));

4599         if (sfmmup == ksfmmup) {
4600                 while ((pfn = sfmmu_vatopfn(addr, sfmmup, &tte))
4601                     == PFN_SUSPENDED) {
4602                         sfmmu_vatopfn_suspended(addr, sfmmup, &tte);
4603                 }
4604         } else {
4605                 pfn = sfmmu_uvatopfn(addr, sfmmup, NULL);
4606         }

4608         if (pfn != PFN_INVALID)
4609                 return (1);
4610         else
4611                 return (0);
4612 }

4614 ssize_t
4615 hat_getpagesize(struct hat *sfmmup, caddr_t addr)
4616 {
4617         tte_t tte;

4796         ASSERT(sfmmup->sfmmu_xhat_provider == NULL);

4619         if (sfmmup == ksfmmup) {
4620                 if (sfmmu_vatopfn(addr, sfmmup, &tte) == PFN_INVALID) {
4621                         return (-1);
4622                 }
4623         } else {
4624                 if (sfmmu_uvatopfn(addr, sfmmup, &tte) == PFN_INVALID) {
4625                         return (-1);
4626                 }
4627         }

4629         ASSERT(TTE_IS_VALID(&tte));
4630         return (TTEBYTES(TTE_CSZ(&tte)));
4631 }

4633 uint_t
4634 hat_getattr(struct hat *sfmmup, caddr_t addr, uint_t *attr)
4635 {
4636         tte_t tte;

4817         ASSERT(sfmmup->sfmmu_xhat_provider == NULL);

4638         if (sfmmup == ksfmmup) {
4639                 if (sfmmu_vatopfn(addr, sfmmup, &tte) == PFN_INVALID) {
4640                         tte.ll = 0;
```



```
4641                 }
4642         } else {
4643                 if (sfmmu_uvatopfn(addr, sfmmup, &tte) == PFN_INVALID) {
4644                         tte.ll = 0;
4645                 }
4646         }
4647         if (TTE_IS_VALID(&tte)) {
4648                 *attr = sfmmu_ptov_attr(&tte);
4649                 return (0);
4650         }
4651         *attr = 0;
4652         return ((uint_t)0xffffffff);
4653 }

4655 /*
4656  * Enables more attributes on specified address range (ie. logical OR)
4657  */
4658 void
4659 hat_setattr(struct hat *hat, caddr_t addr, size_t len, uint_t attr)
4660 {
4842         if (hat->sfmmu_xhat_provider) {
4843                 XHAT_SETATTR(hat, addr, len, attr);
4844                 return;
4845         } else {
4846                 /*
4847                  * This must be a CPU HAT. If the address space has
4848                  * XHATs attached, change attributes for all of them,
4849                  * just in case
4850                  */
4661                 ASSERT(hat->sfmmu_as != NULL);
4852                 if (hat->sfmmu_as->a_xhat != NULL)
4853                         xhat_setattr_all(hat->sfmmu_as, addr, len, attr);
4854         }

4663         sfmmu_chgattr(hat, addr, len, attr, SFMMU_SETATTR);
4664 }

4666 /*
4667  * Assigns attributes to the specified address range.  All the attributes
4668  * are specified.
4669  */
4670 void
4671 hat_chgattr(struct hat *hat, caddr_t addr, size_t len, uint_t attr)
4672 {
4866         if (hat->sfmmu_xhat_provider) {
4867                 XHAT_CHGATTR(hat, addr, len, attr);
4868                 return;
4869         } else {
4870                 /*
4871                  * This must be a CPU HAT. If the address space has
4872                  * XHATs attached, change attributes for all of them,
4873                  * just in case
4874                  */
4673                 ASSERT(hat->sfmmu_as != NULL);
4876                 if (hat->sfmmu_as->a_xhat != NULL)
4877                         xhat_chgattr_all(hat->sfmmu_as, addr, len, attr);
4878         }

4675         sfmmu_chgattr(hat, addr, len, attr, SFMMU_CHGATTR);
4676 }

4678 /*
4679  * Remove attributes on the specified address range (ie. loginal NAND)
4680  */
4681 void
4682 hat_clrattr(struct hat *hat, caddr_t addr, size_t len, uint_t attr)
```

```
4683 {
4889         if (hat->sfmmu_xhat_provider) {
4890                 XHAT_CLRATTR(hat, addr, len, attr);
4891                 return;
4892         } else {
4893                 /*
4894                  * This must be a CPU HAT. If the address space has
4895                  * XHATs attached, change attributes for all of them,
4896                  * just in case
4897                  */
4684         ASSERT(hat->sfmmu_as != NULL);
4899                 if (hat->sfmmu_as->a_xhat != NULL)
4900                         xhat_clrattr_all(hat->sfmmu_as, addr, len, attr);
4901         }

4686         sfmmu_chgattr(hat, addr, len, attr, SFMMU_CLRATTR);
4687 }
______unchanged_portion_omitted_

5013 /*
5014  * hat_chgprot is a deprecated hat call.  New segment drivers
5015  * should store all attributes and use hat_*attr calls.
5016  *
5017  * Change the protections in the virtual address range
5018  * given to the specified virtual protection.  If vprot is ~PROT_WRITE,
5019  * then remove write permission, leaving the other
5020  * permissions unchanged.  If vprot is ~PROT_USER, remove user permissions.
5021  *
5022  */
5023 void
5024 hat_chgprot(struct hat *sfmmup, caddr_t addr, size_t len, uint_t vprot)
5025 {
5026         struct hmehash_bucket *hmebp;
5027         hmeblk_tag hblktag;
5028         int hmeshift, hashno = 1;
5029         struct hme_blk *hmeblkp, *list = NULL;
5030         caddr_t endaddr;
5031         cpuset_t cpuset;
5032         demap_range_t dmr;

5034         ASSERT((len & MMU_PAGEOFFSET) == 0);
5035         ASSERT(((uintptr_t)addr & MMU_PAGEOFFSET) == 0);

5254         if (sfmmup->sfmmu_xhat_provider) {
5255                 XHAT_CHGPROT(sfmmup, addr, len, vprot);
5256                 return;
5257         } else {
5258                 /*
5259                  * This must be a CPU HAT. If the address space has
5260                  * XHATs attached, change attributes for all of them,
5261                  * just in case
5262                  */
5037         ASSERT(sfmmup->sfmmu_as != NULL);
5264                 if (sfmmup->sfmmu_as->a_xhat != NULL)
5265                         xhat_chgprot_all(sfmmup->sfmmu_as, addr, len, vprot);
5266         }

5039         CPUSET_ZERO(cpuset);

5041         if ((vprot != (uint_t)~PROT_WRITE) && (vprot & PROT_USER) &&
5042             ((addr + len) > (caddr_t)USERLIMIT)) {
5043                 panic("user addr %p vprot %x in kernel space",
5044                     (void *)addr, vprot);
5045         }
5046         endaddr = addr + len;
5047         hblktag.htag_id = sfmmup;
```



```
5048         hblktag.htag_rid = SFMMU_INVALID_SHMERID;
5049         DEMAP_RANGE_INIT(sfmmup, &dmr);

5051         while (addr < endaddr) {
5052                 hmeshift = HME_HASH_SHIFT(hashno);
5053                 hblktag.htag_bspage = HME_HASH_BSPAGE(addr, hmeshift);
5054                 hblktag.htag_rehash = hashno;
5055                 hmebp = HME_HASH_FUNCTION(sfmmup, addr, hmeshift);

5057                 SFMMU_HASH_LOCK(hmebp);

5059                 HME_HASH_SEARCH(hmebp, hblktag, hmeblkp, &list);
5060                 if (hmeblkp != NULL) {
5061                         ASSERT(!hmeblkp->hblk_shared);
5062                         /*
5063                          * We've encountered a shadow hmeblk so skip the range
5064                          * of the next smaller mapping size.
5065                          */
5066                         if (hmeblkp->hblk_shw_bit) {
5067                                 ASSERT(sfmmup != ksfmmup);
5068                                 ASSERT(hashno > 1);
5069                                 addr = (caddr_t)P2END((uintptr_t)addr,
5070                                     TTEBYTES(hashno - 1));
5071                         } else {
5072                                 addr = sfmmu_hblk_chgprot(sfmmup, hmeblkp,
5073                                     addr, endaddr, &dmr, vprot);
5074                         }
5075                         SFMMU_HASH_UNLOCK(hmebp);
5076                         hashno = 1;
5077                         continue;
5078                 }
5079                 SFMMU_HASH_UNLOCK(hmebp);

5081                 if (!HME_REHASH(sfmmup) || (hashno >= mmu_hashcnt)) {
5082                         /*
5083                          * We have traversed the whole list and rehashed
5084                          * if necessary without finding the address to chgprot.
5085                          * This is ok so we increment the address by the
5086                          * smallest hmeblk range for kernel mappings and the
5087                          * largest hmeblk range, to account for shadow hmeblks,
5088                          * for user mappings and continue.
5089                          */
5090                         if (sfmmup == ksfmmup)
5091                                 addr = (caddr_t)P2END((uintptr_t)addr,
5092                                     TTEBYTES(1));
5093                         else
5094                                 addr = (caddr_t)P2END((uintptr_t)addr,
5095                                     TTEBYTES(hashno));
5096                         hashno = 1;
5097                 } else {
5098                         hashno++;
5099                 }
5100         }

5102         sfmmu_hblks_list_purge(&list, 0);
5103         DEMAP_RANGE_FLUSH(&dmr);
5104         cpuset = sfmmup->sfmmu_cpusran;
5105         xt_sync(cpuset);
5106 }
______unchanged_portion_omitted_

5441 /*
5442  * Unload all the mappings in the range [addr..addr+len). addr and len must
5443  * be MMU_PAGESIZE aligned.
5444  */
```

```
5446 extern struct seg *segkmap;
5447 #define ISSEGKMAP(sfmmup, addr) (sfmmup == ksfmmup && \
5448 segkmap->s_base <= (addr) && (addr) < (segkmap->s_base + segkmap->s_size))


5451 void
5452 hat_unload_callback(
5453         struct hat *sfmmup,
5454         caddr_t addr,
5455         size_t len,
5456         uint_t flags,
5457         hat_callback_t *callback)
5458 {
5459         struct hmehash_bucket *hmebp;
5460         hmeblk_tag hblktag;
5461         int hmeshift, hashno, iskernel;
5462         struct hme_blk *hmeblkp, *pr_hblk, *list = NULL;
5463         caddr_t endaddr;
5464         cpuset_t cpuset;
5465         int addr_count = 0;
5466         int a;
5467         caddr_t cb_start_addr[MAX_CB_ADDR];
5468         caddr_t cb_end_addr[MAX_CB_ADDR];
5469         int issegkmap = ISSEGKMAP(sfmmup, addr);
5470         demap_range_t dmr, *dmrp;

5701         if (sfmmup->sfmmu_xhat_provider) {
5702                 XHAT_UNLOAD_CALLBACK(sfmmup, addr, len, flags, callback);
5703                 return;
5704         } else {
5705                 /*
5706                  * This must be a CPU HAT. If the address space has
5707                  * XHATs attached, unload the mappings for all of them,
5708                  * just in case
5709                  */
5472         ASSERT(sfmmup->sfmmu_as != NULL);
5711                 if (sfmmup->sfmmu_as->a_xhat != NULL)
5712                         xhat_unload_callback_all(sfmmup->sfmmu_as, addr,
5713                             len, flags, callback);
5714         }

5474         ASSERT((sfmmup == ksfmmup) || (flags & HAT_UNLOAD_OTHER) || \
5475             AS_LOCK_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));

5477         ASSERT(sfmmup != NULL);
5478         ASSERT((len & MMU_PAGEOFFSET) == 0);
5479         ASSERT(!((uintptr_t)addr & MMU_PAGEOFFSET));

5481         /*
5482          * Probing through a large VA range (say 63 bits) will be slow, even
5483          * at 4 Meg steps between the probes. So, when the virtual address range
5484          * is very large, search the HME entries for what to unload.
5485          *
5486          *      len >> TTE_PAGE_SHIFT(TTE4M) is the # of 4Meg probes we'd need
5487          *
5488          *      UHMEHASH_SZ is number of hash buckets to examine
5489          *
5490          */
5491         if (sfmmup != KHATID && (len >> TTE_PAGE_SHIFT(TTE4M)) > UHMEHASH_SZ) {
5492                 hat_unload_large_virtual(sfmmup, addr, len, flags, callback);
5493                 return;
5494         }

5496         CPUSET_ZERO(cpuset);

5498         /*
```



```
5499          * If the process is exiting, we can save a lot of fuss since
5500          * we'll flush the TLB when we free the ctx anyway.
5501          */
5502         if (sfmmup->sfmmu_free) {
5503                 dmrp = NULL;
5504         } else {
5505                 dmrp = &dmr;
5506                 DEMAP_RANGE_INIT(sfmmup, dmrp);
5507         }

5509         endaddr = addr + len;
5510         hblktag.htag_id = sfmmup;
5511         hblktag.htag_rid = SFMMU_INVALID_SHMERID;

5513         /*
5514          * It is likely for the vm to call unload over a wide range of
5515          * addresses that are actually very sparsely populated by
5516          * translations.  In order to speed this up the sfmmu hat supports
5517          * the concept of shadow hmeblks. Dummy large page hmeblks that
5518          * correspond to actual small translations are allocated at tteload
5519          * time and are referred to as shadow hmeblks.  Now, during unload
5520          * time, we first check if we have a shadow hmeblk for that
5521          * translation.  The absence of one means the corresponding address
5522          * range is empty and can be skipped.
5523          *
5524          * The kernel is an exception to above statement and that is why
5525          * we don't use shadow hmeblks and hash starting from the smallest
5526          * page size.
5527          */
5528         if (sfmmup == KHATID) {
5529                 iskernel = 1;
5530                 hashno = TTE64K;
5531         } else {
5532                 iskernel = 0;
5533                 if (mmu_page_sizes == max_mmu_page_sizes) {
5534                         hashno = TTE256M;
5535                 } else {
5536                         hashno = TTE4M;
5537                 }
5538         }
5539         while (addr < endaddr) {
5540                 hmeshift = HME_HASH_SHIFT(hashno);
5541                 hblktag.htag_bspage = HME_HASH_BSPAGE(addr, hmeshift);
5542                 hblktag.htag_rehash = hashno;
5543                 hmebp = HME_HASH_FUNCTION(sfmmup, addr, hmeshift);

5545                 SFMMU_HASH_LOCK(hmebp);

5547                 HME_HASH_SEARCH_PREV(hmebp, hblktag, hmeblkp, pr_hblk, &list);
5548                 if (hmeblkp == NULL) {
5549                         /*
5550                          * didn't find an hmeblk. skip the appropiate
5551                          * address range.
5552                          */
5553                         SFMMU_HASH_UNLOCK(hmebp);
5554                         if (iskernel) {
5555                                 if (hashno < mmu_hashcnt) {
5556                                         hashno++;
5557                                         continue;
5558                                 } else {
5559                                         hashno = TTE64K;
5560                                         addr = (caddr_t)roundup((uintptr_t)addr
5561                                             + 1, MMU_PAGESIZE64K);
5562                                         continue;
5563                                 }
5564                         }
```

```
5565                        addr = (caddr_t)roundup((uintptr_t)addr + 1,
5566                            (1 << hmeshift));
5567                        if ((uintptr_t)addr & MMU_PAGEOFFSET512K) {
5568                                ASSERT(hashno == TTE64K);
5569                                continue;
5570                        }
5571                        if ((uintptr_t)addr & MMU_PAGEOFFSET4M) {
5572                                hashno = TTE512K;
5573                                continue;
5574                        }
5575                        if (mmu_page_sizes == max_mmu_page_sizes) {
5576                                if ((uintptr_t)addr & MMU_PAGEOFFSET32M) {
5577                                        hashno = TTE4M;
5578                                        continue;
5579                                }
5580                                if ((uintptr_t)addr & MMU_PAGEOFFSET256M) {
5581                                        hashno = TTE32M;
5582                                        continue;
5583                                }
5584                                hashno = TTE256M;
5585                                continue;
5586                        } else {
5587                                hashno = TTE4M;
5588                                continue;
5589                        }
5590                }
5591                ASSERT(hmeblkp);
5592                ASSERT(!hmeblkp->hblk_shared);
5593                if (!hmeblkp->hblk_vcnt && !hmeblkp->hblk_hmecnt) {
5594                        /*
5595                         * If the valid count is zero we can skip the range
5596                         * mapped by this hmeblk.
5597                         * We free hblks in the case of HAT_UNMAP.  HAT_UNMAP
5598                         * is used by segment drivers as a hint
5599                         * that the mapping resource won't be used any longer.
5600                         * The best example of this is during exit().
5601                         */
5602                        addr = (caddr_t)roundup((uintptr_t)addr + 1,
5603                            get_hblk_span(hmeblkp));
5604                        if ((flags & HAT_UNLOAD_UNMAP) ||
5605                            (iskernel && !issegkmap)) {
5606                                sfmmu_hblk_hash_rm(hmebp, hmeblkp, pr_hblk,
5607                                    &list, 0);
5608                        }
5609                        SFMMU_HASH_UNLOCK(hmebp);

5611                        if (iskernel) {
5612                                hashno = TTE64K;
5613                                continue;
5614                        }
5615                        if ((uintptr_t)addr & MMU_PAGEOFFSET512K) {
5616                                ASSERT(hashno == TTE64K);
5617                                continue;
5618                        }
5619                        if ((uintptr_t)addr & MMU_PAGEOFFSET4M) {
5620                                hashno = TTE512K;
5621                                continue;
5622                        }
5623                        if (mmu_page_sizes == max_mmu_page_sizes) {
5624                                if ((uintptr_t)addr & MMU_PAGEOFFSET32M) {
5625                                        hashno = TTE4M;
5626                                        continue;
5627                                }
5628                                if ((uintptr_t)addr & MMU_PAGEOFFSET256M) {
5629                                        hashno = TTE32M;
5630                                        continue;
```



```
5631                                }
5632                                hashno = TTE256M;
5633                                continue;
5634                        } else {
5635                                hashno = TTE4M;
5636                                continue;
5637                        }
5638                }
5639                if (hmeblkp->hblk_shw_bit) {
5640                        /*
5641                         * If we encounter a shadow hmeblk we know there is
5642                         * smaller sized hmeblks mapping the same address space.
5643                         * Decrement the hash size and rehash.
5644                         */
5645                        ASSERT(sfmmup != KHATID);
5646                        hashno--;
5647                        SFMMU_HASH_UNLOCK(hmebp);
5648                        continue;
5649                }

5651                /*
5652                 * track callback address ranges.
5653                 * only start a new range when it's not contiguous
5654                 */
5655                if (callback != NULL) {
5656                        if (addr_count > 0 &&
5657                            addr == cb_end_addr[addr_count - 1])
5658                                --addr_count;
5659                        else
5660                                cb_start_addr[addr_count] = addr;
5661                }

5663                addr = sfmmu_hblk_unload(sfmmup, hmeblkp, addr, endaddr,
5664                    dmrp, flags);

5666                if (callback != NULL)
5667                        cb_end_addr[addr_count++] = addr;

5669                if (((flags & HAT_UNLOAD_UNMAP) || (iskernel && !issegkmap)) &&
5670                    !hmeblkp->hblk_vcnt && !hmeblkp->hblk_hmecnt) {
5671                        sfmmu_hblk_hash_rm(hmebp, hmeblkp, pr_hblk, &list, 0);
5672                }
5673                SFMMU_HASH_UNLOCK(hmebp);

5675                /*
5676                 * Notify our caller as to exactly which pages
5677                 * have been unloaded. We do these in clumps,
5678                 * to minimize the number of xt_sync()s that need to occur.
5679                 */
5680                if (callback != NULL && addr_count == MAX_CB_ADDR) {
5681                        if (dmrp != NULL) {
5682                                DEMAP_RANGE_FLUSH(dmrp);
5683                                cpuset = sfmmup->sfmmu_cpusran;
5684                                xt_sync(cpuset);
5685                        }

5687                        for (a = 0; a < MAX_CB_ADDR; ++a) {
5688                                callback->hcb_start_addr = cb_start_addr[a];
5689                                callback->hcb_end_addr = cb_end_addr[a];
5690                                callback->hcb_function(callback);
5691                        }
5692                        addr_count = 0;
5693                }
5694                if (iskernel) {
5695                        hashno = TTE64K;
5696                        continue;
```

```
5697                }
5698                if ((uintptr_t)addr & MMU_PAGEOFFSET512K) {
5699                        ASSERT(hashno == TTE64K);
5700                        continue;
5701                }
5702                if ((uintptr_t)addr & MMU_PAGEOFFSET4M) {
5703                        hashno = TTE512K;
5704                        continue;
5705                }
5706                if (mmu_page_sizes == max_mmu_page_sizes) {
5707                        if ((uintptr_t)addr & MMU_PAGEOFFSET32M) {
5708                                hashno = TTE4M;
5709                                continue;
5710                        }
5711                        if ((uintptr_t)addr & MMU_PAGEOFFSET256M) {
5712                                hashno = TTE32M;
5713                                continue;
5714                        }
5715                        hashno = TTE256M;
5716                } else {
5717                        hashno = TTE4M;
5718                }
5719        }

5721        sfmmu_hblks_list_purge(&list, 0);
5722        if (dmrp != NULL) {
5723                DEMAP_RANGE_FLUSH(dmrp);
5724                cpuset = sfmmup->sfmmu_cpusran;
5725                xt_sync(cpuset);
5726        }
5727        if (callback && addr_count != 0) {
5728                for (a = 0; a < addr_count; ++a) {
5729                        callback->hcb_start_addr = cb_start_addr[a];
5730                        callback->hcb_end_addr = cb_end_addr[a];
5731                        callback->hcb_function(callback);
5732                }
5733        }

5735        /*
5736         * Check TSB and TLB page sizes if the process isn't exiting.
5737         */
5738        if (!sfmmup->sfmmu_free)
5739                sfmmu_check_page_sizes(sfmmup, 0);
5740 }

5742 /*
5743  * Unload all the mappings in the range [addr..addr+len). addr and len must
5744  * be MMU_PAGESIZE aligned.
5745  */
5746 void
5747 hat_unload(struct hat *sfmmup, caddr_t addr, size_t len, uint_t flags)
5748 {
5991        if (sfmmup->sfmmu_xhat_provider) {
5992                XHAT_UNLOAD(sfmmup, addr, len, flags);
5993                return;
5994        }
5749        hat_unload_callback(sfmmup, addr, len, flags, NULL);
5750 }
```

*unchanged_portion_omitted_*

```
6072 /*
6073  * Synchronize all the mappings in the range [addr..addr+len).
6074  * Can be called with clearflag having two states:
6075  * HAT_SYNC_DONTZERO means just return the rm stats
6076  * HAT_SYNC_ZERORM means zero rm bits in the tte and return the stats
6077  */
```



```
6078 void
6079 hat_sync(struct hat *sfmmup, caddr_t addr, size_t len, uint_t clearflag)
6080 {
6081        struct hmehash_bucket *hmebp;
6082        hmeblk_tag hblktag;
6083        int hmeshift, hashno = 1;
6084        struct hme_blk *hmeblkp, *list = NULL;
6085        caddr_t endaddr;
6086        cpuset_t cpuset;

6334        ASSERT(sfmmup->sfmmu_xhat_provider == NULL);
6088        ASSERT((sfmmup == ksfmmup) ||
6089            AS_LOCK_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));
6090        ASSERT((len & MMU_PAGEOFFSET) == 0);
6091        ASSERT((clearflag == HAT_SYNC_DONTZERO) ||
6092            (clearflag == HAT_SYNC_ZERORM));

6094        CPUSET_ZERO(cpuset);

6096        endaddr = addr + len;
6097        hblktag.htag_id = sfmmup;
6098        hblktag.htag_rid = SFMMU_INVALID_SHMERID;

6100        /*
6101         * Spitfire supports 4 page sizes.
6102         * Most pages are expected to be of the smallest page
6103         * size (8K) and these will not need to be rehashed. 64K
6104         * pages also don't need to be rehashed because the an hmeblk
6105         * spans 64K of address space. 512K pages might need 1 rehash and
6106         * and 4M pages 2 rehashes.
6107         */
6108        while (addr < endaddr) {
6109                hmeshift = HME_HASH_SHIFT(hashno);
6110                hblktag.htag_bspage = HME_HASH_BSPAGE(addr, hmeshift);
6111                hblktag.htag_rehash = hashno;
6112                hmebp = HME_HASH_FUNCTION(sfmmup, addr, hmeshift);

6114                SFMMU_HASH_LOCK(hmebp);

6116                HME_HASH_SEARCH(hmebp, hblktag, hmeblkp, &list);
6117                if (hmeblkp != NULL) {
6118                        ASSERT(!hmeblkp->hblk_shared);
6119                        /*
6120                         * We've encountered a shadow hmeblk so skip the range
6121                         * of the next smaller mapping size.
6122                         */
6123                        if (hmeblkp->hblk_shw_bit) {
6124                                ASSERT(sfmmup != ksfmmup);
6125                                ASSERT(hashno > 1);
6126                                addr = (caddr_t)P2END((uintptr_t)addr,
6127                                    TTEBYTES(hashno - 1));
6128                        } else {
6129                                addr = sfmmu_hblk_sync(sfmmup, hmeblkp,
6130                                    addr, endaddr, clearflag);
6131                        }
6132                        SFMMU_HASH_UNLOCK(hmebp);
6133                        hashno = 1;
6134                        continue;
6135                }
6136                SFMMU_HASH_UNLOCK(hmebp);

6138                if (!HME_REHASH(sfmmup) || (hashno >= mmu_hashcnt)) {
6139                        /*
6140                         * We have traversed the whole list and rehashed
6141                         * if necessary without finding the address to sync.
6142                         * This is ok so we increment the address by the
```

```
6143                          * smallest hmeblk range for kernel mappings and the
6144                          * largest hmeblk range, to account for shadow hmeblks,
6145                          * for user mappings and continue.
6146                          */
6147                         if (sfmmup == ksfmmup)
6148                                 addr = (caddr_t)P2END((uintptr_t)addr,
6149                                     TTEBYTES(1));
6150                         else
6151                                 addr = (caddr_t)P2END((uintptr_t)addr,
6152                                     TTEBYTES(hashno));
6153                         hashno = 1;
6154                 } else {
6155                         hashno++;
6156                 }
6157         }
6158         sfmmu_hblks_list_purge(&list, 0);
6159         cpuset = sfmmup->sfmmu_cpusran;
6160         xt_sync(cpuset);
6161 }
_____unchanged_portion_omitted_

6879 /*
6880  * Remove all mappings to page 'pp'.
6881  */
6882 int
6883 hat_pageunload(struct page *pp, uint_t forceflag)
6884 {
6885         struct page *origpp = pp;
6886         struct sf_hment *sfhme, *tmphme;
6887         struct hme_blk *hmeblkp;
6888         kmutex_t *pml;
6889 #ifdef VAC
6890         kmutex_t *pmtx;
6891 #endif
6892         cpuset_t cpuset, tset;
6893         int index, cons;
7141         int xhme_blks;
6894         int pa_hments;

6896         ASSERT(PAGE_EXCL(pp));

7146 retry_xhat:
6898         tmphme = NULL;
7148         xhme_blks = 0;
6899         pa_hments = 0;
6900         CPUSET_ZERO(cpuset);

6902         pml = sfmmu_mlist_enter(pp);

6904 #ifdef VAC
6905         if (pp->p_kpmref)
6906                 sfmmu_kpm_pageunload(pp);
6907         ASSERT(!PP_ISMAPPED_KPM(pp));
6908 #endif
6909         /*
6910          * Clear vpm reference. Since the page is exclusively locked
6911          * vpm cannot be referencing it.
6912          */
6913         if (vpm_enable) {
6914                 pp->p_vpmref = 0;
6915         }

6917         index = PP_MAPINDEX(pp);
6918         cons = TTE8K;
6919 retry:
6920         for (sfhme = pp->p_mapping; sfhme; sfhme = tmphme) {
```



```
6921                 tmphme = sfhme->hme_next;

6923                 if (IS_PAHME(sfhme)) {
6924                         ASSERT(sfhme->hme_data != NULL);
6925                         pa_hments++;
6926                         continue;
6927                 }

6929                 hmeblkp = sfmmu_hmetohblk(sfhme);
7180                 if (hmeblkp->hblk_xhat_bit) {
7181                         struct xhat_hme_blk *xblk =
7182                             (struct xhat_hme_blk *)hmeblkp;

7184                         (void) XHAT_PAGEUNLOAD(xblk->xhat_hme_blk_hat,
7185                             pp, forceflag, XBLK2PROVBLK(xblk));

7187                         xhme_blks = 1;
7188                         continue;
7189                 }

6931                 /*
6932                  * If there are kernel mappings don't unload them, they will
6933                  * be suspended.
6934                  */
6935                 if (forceflag == SFMMU_KERNEL_RELOC && hmeblkp->hblk_lckcnt &&
6936                     hmeblkp->hblk_tag.htag_id == ksfmmup)
6937                         continue;

6939                 tset = sfmmu_pageunload(pp, sfhme, cons);
6940                 CPUSET_OR(cpuset, tset);
6941         }

6943         while (index != 0) {
6944                 index = index >> 1;
6945                 if (index != 0)
6946                         cons++;
6947                 if (index & 0x1) {
6948                         /* Go to leading page */
6949                         pp = PP_GROUPLEADER(pp, cons);
6950                         ASSERT(sfmmu_mlist_held(pp));
6951                         goto retry;
6952                 }
6953         }

6955         /*
6956          * cpuset may be empty if the page was only mapped by segkpm,
6957          * in which case we won't actually cross-trap.
6958          */
6959         xt_sync(cpuset);

6961         /*
6962          * The page should have no mappings at this point, unless
6963          * we were called from hat_page_relocate() in which case we
6964          * leave the locked mappings which will be suspended later.
6965          */
6966         ASSERT(!PP_ISMAPPED(origpp) || pa_hments ||
7226         ASSERT(!PP_ISMAPPED(origpp) || xhme_blks || pa_hments ||
6967             (forceflag == SFMMU_KERNEL_RELOC));

6969 #ifdef VAC
6970         if (PP_ISTNC(pp)) {
6971                 if (cons == TTE8K) {
6972                         pmtx = sfmmu_page_enter(pp);
6973                         PP_CLRTNC(pp);
6974                         sfmmu_page_exit(pmtx);
6975                 } else {
```

```
6976                        conv_tnc(pp, cons);
6977                }
6978        }
6979 #endif  /* VAC */

6981        if (pa_hments && forceflag != SFMMU_KERNEL_RELOC) {
6982                /*
6983                 * Unlink any pa_hments and free them, calling back
6984                 * the responsible subsystem to notify it of the error.
6985                 * This can occur in situations such as drivers leaking
6986                 * DMA handles: naughty, but common enough that we'd like
6987                 * to keep the system running rather than bringing it
6988                 * down with an obscure error like "pa_hment leaked"
6989                 * which doesn't aid the user in debugging their driver.
6990                 */
6991                for (sfhme = pp->p_mapping; sfhme; sfhme = tmphme) {
6992                        tmphme = sfhme->hme_next;
6993                        if (IS_PAHME(sfhme)) {
6994                                struct pa_hment *pahmep = sfhme->hme_data;
6995                                sfmmu_pahment_leaked(pahmep);
6996                                HME_SUB(sfhme, pp);
6997                                kmem_cache_free(pa_hment_cache, pahmep);
6998                        }
6999                }

7001                ASSERT(!PP_ISMAPPED(origpp));
7261                ASSERT(!PP_ISMAPPED(origpp) || xhme_blks);
7002        }

7004        sfmmu_mlist_exit(pml);

7266        /*
7267         * XHAT may not have finished unloading pages
7268         * because some other thread was waiting for
7269         * mlist lock and XHAT_PAGEUNLOAD let it do
7270         * the job.
7271         */
7272        if (xhme_blks) {
7273                pp = origpp;
7274                goto retry_xhat;
7275        }

7006        return (0);
7007 }
```

_____***unchanged_portion_omitted_***

```
7231 uint_t
7232 hat_pagesync(struct page *pp, uint_t clearflag)
7233 {
7234        struct sf_hment *sfhme, *tmphme = NULL;
7235        struct hme_blk *hmeblkp;
7236        kmutex_t *pml;
7237        cpuset_t cpuset, tset;
7238        int     index, cons;
7239        extern  ulong_t po_share;
7240        page_t  *save_pp = pp;
7241        int     stop_on_sh = 0;
7242        uint_t  shcnt;

7244        CPUSET_ZERO(cpuset);

7246        if (PP_ISRO(pp) && (clearflag & HAT_SYNC_STOPON_MOD)) {
7247                return (PP_GENERIC_ATTR(pp));
7248        }

7250        if ((clearflag & HAT_SYNC_ZERORM) == 0) {
```



```
7251                if ((clearflag & HAT_SYNC_STOPON_REF) && PP_ISREF(pp)) {
7252                        return (PP_GENERIC_ATTR(pp));
7253                }
7254                if ((clearflag & HAT_SYNC_STOPON_MOD) && PP_ISMOD(pp)) {
7255                        return (PP_GENERIC_ATTR(pp));
7256                }
7257                if (clearflag & HAT_SYNC_STOPON_SHARED) {
7258                        if (pp->p_share > po_share) {
7259                                hat_page_setattr(pp, P_REF);
7260                                return (PP_GENERIC_ATTR(pp));
7261                        }
7262                        stop_on_sh = 1;
7263                        shcnt = 0;
7264                }
7265        }

7267        clearflag &= ~HAT_SYNC_STOPON_SHARED;
7268        pml = sfmmu_mlist_enter(pp);
7269        index = PP_MAPINDEX(pp);
7270        cons = TTE8K;
7271 retry:
7272        for (sfhme = pp->p_mapping; sfhme; sfhme = tmphme) {
7273                /*
7274                 * We need to save the next hment on the list since
7275                 * it is possible for pagesync to remove an invalid hment
7276                 * from the list.
7277                 */
7278                tmphme = sfhme->hme_next;
7279                if (IS_PAHME(sfhme))
7280                        continue;
7281                /*
7282                 * If we are looking for large mappings and this hme doesn't
7283                 * reach the range we are seeking, just ignore it.
7284                 */
7285                hmeblkp = sfmmu_hmetohblk(sfhme);
7557                if (hmeblkp->hblk_xhat_bit)
7558                        continue;

7287                if (hme_size(sfhme) < cons)
7288                        continue;

7290                if (stop_on_sh) {
7291                        if (hmeblkp->hblk_shared) {
7292                                sf_srd_t *srdp = hblktosrd(hmeblkp);
7293                                uint_t rid = hmeblkp->hblk_tag.htag_rid;
7294                                sf_region_t *rgnp;
7295                                ASSERT(SFMMU_IS_SHMERID_VALID(rid));
7296                                ASSERT(rid < SFMMU_MAX_HME_REGIONS);
7297                                ASSERT(srdp != NULL);
7298                                rgnp = srdp->srd_hmergnp[rid];
7299                                SFMMU_VALIDATE_SHAREDHBLK(hmeblkp, srdp,
7300                                    rgnp, rid);
7301                                shcnt += rgnp->rgn_refcnt;
7302                        } else {
7303                                shcnt++;
7304                        }
7305                        if (shcnt > po_share) {
7306                                /*
7307                                 * tell the pager to spare the page this time
7308                                 * around.
7309                                 */
7310                                hat_page_setattr(save_pp, P_REF);
7311                                index = 0;
7312                                break;
7313                        }
7314                }
```

```
7315                 tset = sfmmu_pagesync(pp, sfhme,
7316                     clearflag & ~HAT_SYNC_STOPON_RM);
7317                 CPUSET_OR(cpuset, tset);
7318
7319                 /*
7320                  * If clearflag is HAT_SYNC_DONTZERO, break out as soon
7321                  * as the "ref" or "mod" is set or share cnt exceeds po_share.
7322                  */
7323                 if ((clearflag & ~HAT_SYNC_STOPON_RM) == HAT_SYNC_DONTZERO &&
7324                     (((clearflag & HAT_SYNC_STOPON_MOD) && PP_ISMOD(save_pp)) ||
7325                     ((clearflag & HAT_SYNC_STOPON_REF) && PP_ISREF(save_pp)))) {
7326                         index = 0;
7327                         break;
7328                 }
7329         }
7330
7331         while (index) {
7332                 index = index >> 1;
7333                 cons++;
7334                 if (index & 0x1) {
7335                         /* Go to leading page */
7336                         pp = PP_GROUPLEADER(pp, cons);
7337                         goto retry;
7338                 }
7339         }
7340
7341         xt_sync(cpuset);
7342         sfmmu_mlist_exit(pml);
7343         return (PP_GENERIC_ATTR(save_pp));
7344 }
_____unchanged_portion_omitted_

7417 /*
7418  * Remove write permission from a mappings to a page, so that
7419  * we can detect the next modification of it. This requires modifying
7420  * the TTE then invalidating (demap) any TLB entry using that TTE.
7421  * This code is similar to sfmmu_pagesync().
7422  */
7423 static cpuset_t
7424 sfmmu_pageclrwrt(struct page *pp, struct sf_hment *sfhme)
7425 {
7426         caddr_t addr;
7427         tte_t tte;
7428         tte_t ttemod;
7429         struct hme_blk *hmeblkp;
7430         int ret;
7431         sfmmu_t *sfmmup;
7432         cpuset_t cpuset;
7433
7434         ASSERT(pp != NULL);
7435         ASSERT(sfmmu_mlist_held(pp));
7436
7437         CPUSET_ZERO(cpuset);
7438         SFMMU_STAT(sf_clrwrt);
7439
7440 retry:
7441
7442         sfmmu_copytte(&sfhme->hme_tte, &tte);
7443         if (TTE_IS_VALID(&tte) && TTE_IS_WRITABLE(&tte)) {
7444                 hmeblkp = sfmmu_hmetohblk(sfhme);
7445
7719                 /*
7720                  * xhat mappings should never be to a VMODSORT page.
7721                  */
7722                 ASSERT(hmeblkp->hblk_xhat_bit == 0);
```



```
7445                 sfmmup = hblktosfmmu(hmeblkp);
7446                 addr = tte_to_vaddr(hmeblkp, tte);
7447
7448                 ttemod = tte;
7449                 TTE_CLR_WRT(&ttemod);
7450                 TTE_CLR_MOD(&ttemod);
7451                 ret = sfmmu_modifytte_try(&tte, &ttemod, &sfhme->hme_tte);
7452
7453                 /*
7454                  * if cas failed and the new value is not what
7455                  * we want retry
7456                  */
7457                 if (ret < 0)
7458                         goto retry;
7459
7460                 /* we win the cas */
7461                 if (ret > 0) {
7462                         if (hmeblkp->hblk_shared) {
7463                                 sf_srd_t *srdp = (sf_srd_t *)sfmmup;
7464                                 uint_t rid = hmeblkp->hblk_tag.htag_rid;
7465                                 sf_region_t *rgnp;
7466                                 ASSERT(SFMMU_IS_SHMERID_VALID(rid));
7467                                 ASSERT(rid < SFMMU_MAX_HME_REGIONS);
7468                                 ASSERT(srdp != NULL);
7469                                 rgnp = srdp->srd_hmergnp[rid];
7470                                 SFMMU_VALIDATE_SHAREDHBLK(hmeblkp,
7471                                     srdp, rgnp, rid);
7472                                 cpuset = sfmmu_rgntlb_demap(addr,
7473                                     rgnp, hmeblkp, 1);
7474                         } else {
7475                                 sfmmu_tlb_demap(addr, sfmmup, hmeblkp, 0, 0);
7476                                 cpuset = sfmmup->sfmmu_cpusran;
7477                         }
7478                 }
7479         }
7480
7481         return (cpuset);
7482 }
_____unchanged_portion_omitted_
7686 #endif  /* DEBUG */
7687
7688 /*
7689  * Returns a page frame number for a given virtual address.
7690  * Returns PFN_INVALID to indicate an invalid mapping
7691  */
7692 pfn_t
7693 hat_getpfnum(struct hat *hat, caddr_t addr)
7694 {
7695         pfn_t pfn;
7696         tte_t tte;
7697
7698         /*
7699          * We would like to
7700          * ASSERT(AS_LOCK_HELD(as, &as->a_lock));
7701          * but we can't because the iommu driver will call this
7702          * routine at interrupt time and it can't grab the as lock
7703          * or it will deadlock: A thread could have the as lock
7704          * and be waiting for io.  The io can't complete
7705          * because the interrupt thread is blocked trying to grab
7706          * the as lock.
7707          */
7708
7988         ASSERT(hat->sfmmu_xhat_provider == NULL);
7989
7709         if (hat == ksfmmup) {
7710                 if (IS_KMEM_VA_LARGEPAGE(addr)) {
```

```
7711                        ASSERT(segkmem_lpszc > 0);
7712                        pfn = sfmmu_kvaszc2pfn(addr, segkmem_lpszc);
7713                        if (pfn != PFN_INVALID) {
7714                                sfmmu_check_kpfn(pfn);
7715                                return (pfn);
7716                        }
7717                } else if (segkpm && IS_KPM_ADDR(addr)) {
7718                        return (sfmmu_kpm_vatopfn(addr));
7719                }
7720                while ((pfn = sfmmu_vatopfn(addr, ksfmmup, &tte))
7721                    == PFN_SUSPENDED) {
7722                        sfmmu_vatopfn_suspended(addr, ksfmmup, &tte);
7723                }
7724                sfmmu_check_kpfn(pfn);
7725                return (pfn);
7726        } else {
7727                return (sfmmu_uvatopfn(addr, hat, NULL));
7728        }
7729 }
```

_____unchanged_portion_omitted_

```
7884 /*
7885  * For compatability with AT&T and later optimizations
7886  */
7887 /* ARGSUSED */
7888 void
7889 hat_map(struct hat *hat, caddr_t addr, size_t len, uint_t flags)
7890 {
7891         ASSERT(hat != NULL);
8173         ASSERT(hat->sfmmu_xhat_provider == NULL);
7892 }
```

_____unchanged_portion_omitted_

```
7945 /*
7946  * Return 1 if the number of mappings exceeds sh_thresh. Return 0
7947  * otherwise. Count shared hmeblks by region's refcnt.
7948  */
7949 int
7950 hat_page_checkshare(page_t *pp, ulong_t sh_thresh)
7951 {
7952         kmutex_t *pml;
7953         ulong_t cnt = 0;
7954         int index, sz = TTE8K;
7955         struct sf_hment *sfhme, *tmphme = NULL;
7956         struct hme_blk *hmeblkp;

7958         pml = sfmmu_mlist_enter(pp);

7960 #ifdef VAC
7961         if (kpm_enable)
7962                 cnt = pp->p_kpmref;
7963 #endif

7965         if (vpm_enable && pp->p_vpmref) {
7966                 cnt += 1;
7967         }

7969         if (pp->p_share + cnt > sh_thresh) {
7970                 sfmmu_mlist_exit(pml);
7971                 return (1);
7972         }

7974         index = PP_MAPINDEX(pp);

7976 again:
```



```
7977         for (sfhme = pp->p_mapping; sfhme; sfhme = tmphme) {
7978                 tmphme = sfhme->hme_next;
7979                 if (IS_PAHME(sfhme)) {
7980                         continue;
7981                 }

7983                 hmeblkp = sfmmu_hmetohblk(sfhme);
8266                 if (hmeblkp->hblk_xhat_bit) {
8267                         cnt++;
8268                         if (cnt > sh_thresh) {
8269                                 sfmmu_mlist_exit(pml);
8270                                 return (1);
8271                         }
8272                         continue;
8273                 }
7984                 if (hme_size(sfhme) != sz) {
7985                         continue;
7986                 }

7988                 if (hmeblkp->hblk_shared) {
7989                         sf_srd_t *srdp = hblktosrd(hmeblkp);
7990                         uint_t rid = hmeblkp->hblk_tag.htag_rid;
7991                         sf_region_t *rgnp;
7992                         ASSERT(SFMMU_IS_SHMERID_VALID(rid));
7993                         ASSERT(rid < SFMMU_MAX_HME_REGIONS);
7994                         ASSERT(srdp != NULL);
7995                         rgnp = srdp->srd_hmergnp[rid];
7996                         SFMMU_VALIDATE_SHAREDHBLK(hmeblkp, srdp,
7997                             rgnp, rid);
7998                         cnt += rgnp->rgn_refcnt;
7999                 } else {
8000                         cnt++;
8001                 }
8002                 if (cnt > sh_thresh) {
8003                         sfmmu_mlist_exit(pml);
8004                         return (1);
8005                 }
8006         }

8008         index >>= 1;
8009         sz++;
8010         while (index) {
8011                 pp = PP_GROUPLEADER(pp, sz);
8012                 ASSERT(sfmmu_mlist_held(pp));
8013                 if (index & 0x1) {
8014                         goto again;
8015                 }
8016                 index >>= 1;
8017                 sz++;
8018         }
8019         sfmmu_mlist_exit(pml);
8020         return (0);
8021 }

8023 /*
8024  * Unload all large mappings to the pp and reset the p_szc field of every
8025  * constituent page according to the remaining mappings.
8026  *
8027  * pp must be locked SE_EXCL. Even though no other constituent pages are
8028  * locked it's legal to unload the large mappings to the pp because all
8029  * constituent pages of large locked mappings have to be locked SE_SHARED.
8030  * This means if we have SE_EXCL lock on one of constituent pages none of the
8031  * large mappings to pp are locked.
8032  *
8033  * Decrease p_szc field starting from the last constituent page and ending
8034  * with the root page. This method is used because other threads rely on the
```

```
 * root's p_szc to find the lock to syncronize on. After a root page_t's p_szc
 * is demoted then other threads will succeed in sfmmu_mlspl_enter(). This
 * ensures that p_szc changes of the constituent pages appears atomic for all
 * threads that use sfmmu_mlspl_enter() to examine p_szc field.
 *
 * This mechanism is only used for file system pages where it's not always
 * possible to get SE_EXCL locks on all constituent pages to demote the size
 * code (as is done for anonymous or kernel large pages).
 *
 * See more comments in front of sfmmu_mlspl_enter().
 */
void
hat_page_demote(page_t *pp)
{
        int index;
        int sz;
        cpuset_t cpuset;
        int sync = 0;
        page_t *rootpp;
        struct sf_hment *sfhme;
        struct sf_hment *tmphme = NULL;
        struct hme_blk *hmeblkp;
        uint_t pszc;
        page_t *lastpp;
        cpuset_t tset;
        pgcnt_t npgs;
        kmutex_t *pml;
        kmutex_t *pmtx = NULL;

        ASSERT(PAGE_EXCL(pp));
        ASSERT(!PP_ISFREE(pp));
        ASSERT(!PP_ISKAS(pp));
        ASSERT(page_szc_lock_assert(pp));
        pml = sfmmu_mlist_enter(pp);

        pszc = pp->p_szc;
        if (pszc == 0) {
                goto out;
        }

        index = PP_MAPINDEX(pp) >> 1;

        if (index) {
                CPUSET_ZERO(cpuset);
                sz = TTE64K;
                sync = 1;
        }

        while (index) {
                if (!(index & 0x1)) {
                        index >>= 1;
                        sz++;
                        continue;
                }
                ASSERT(sz <= pszc);
                rootpp = PP_GROUPLEADER(pp, sz);
                for (sfhme = rootpp->p_mapping; sfhme; sfhme = tmphme) {
                        tmphme = sfhme->hme_next;
                        ASSERT(!IS_PAHME(sfhme));
                        hmeblkp = sfmmu_hmetohblk(sfhme);
                        if (hme_size(sfhme) != sz) {
                                continue;
                        }
                        if (hmeblkp->hblk_xhat_bit) {
                                cmn_err(CE_PANIC,
                                    "hat_page_demote: xhat hmeblk");
```



```
                        }
                        tset = sfmmu_pageunload(rootpp, sfhme, sz);
                        CPUSET_OR(cpuset, tset);
                }
                if (index >>= 1) {
                        sz++;
                }
        }

        ASSERT(!PP_ISMAPPED_LARGE(pp));

        if (sync) {
                xt_sync(cpuset);
#ifdef VAC
                if (PP_ISTNC(pp)) {
                        conv_tnc(rootpp, sz);
                }
#endif  /* VAC */
        }

        pmtx = sfmmu_page_enter(pp);

        ASSERT(pp->p_szc == pszc);
        rootpp = PP_PAGEROOT(pp);
        ASSERT(rootpp->p_szc == pszc);
        lastpp = PP_PAGENEXT_N(rootpp, TTEPAGES(pszc) - 1);

        while (lastpp != rootpp) {
                sz = PP_MAPINDEX(lastpp) ? fnd_mapping_sz(lastpp) : 0;
                ASSERT(sz < pszc);
                npgs = (sz == 0) ? 1 : TTEPAGES(sz);
                ASSERT(P2PHASE(lastpp->p_pagenum, npgs) == npgs - 1);
                while (--npgs > 0) {
                        lastpp->p_szc = (uchar_t)sz;
                        lastpp = PP_PAGEPREV(lastpp);
                }
                if (sz) {
                        /*
                         * make sure before current root's pszc
                         * is updated all updates to constituent pages pszc
                         * fields are globally visible.
                         */
                        membar_producer();
                }
                lastpp->p_szc = sz;
                ASSERT(IS_P2ALIGNED(lastpp->p_pagenum, TTEPAGES(sz)));
                if (lastpp != rootpp) {
                        lastpp = PP_PAGEPREV(lastpp);
                }
        }
        if (sz == 0) {
                /* the loop above doesn't cover this case */
                rootpp->p_szc = 0;
        }
out:
        ASSERT(pp->p_szc == 0);
        if (pmtx != NULL) {
                sfmmu_page_exit(pmtx);
        }
        sfmmu_mlist_exit(pml);
}
______unchanged_portion_omitted_

/*
 * Yield the memory claim requirement for an address space.
 *
```

```
 * This is currently implemented as the number of bytes that have active
 * hardware translations that have page structures.  Therefore, it can
 * underestimate the traditional resident set size, eg, if the
 * physical page is present and the hardware translation is missing;
 * and it can overestimate the rss, eg, if there are active
 * translations to a frame buffer with page structs.
 * Also, it does not take sharing into account.
 *
 * Note that we don't acquire locks here since this function is most often
 * called from the clock thread.
 */
size_t
hat_get_mapped_size(struct hat *hat)
{
        size_t          assize = 0;
        int             i;

        if (hat == NULL)
                return (0);

        ASSERT(hat->sfmmu_xhat_provider == NULL);

        for (i = 0; i < mmu_page_sizes; i++)
                assize += ((pgcnt_t)hat->sfmmu_ttecnt[i] +
                    (pgcnt_t)hat->sfmmu_scdrttecnt[i]) * TTEBYTES(i);

        if (hat->sfmmu_iblk == NULL)
                return (assize);

        for (i = 0; i < mmu_page_sizes; i++)
                assize += ((pgcnt_t)hat->sfmmu_ismttecnt[i] +
                    (pgcnt_t)hat->sfmmu_scdismttecnt[i]) * TTEBYTES(i);

        return (assize);
}

int
hat_stats_enable(struct hat *hat)
{
        hatlock_t       *hatlockp;

        ASSERT(hat->sfmmu_xhat_provider == NULL);

        hatlockp = sfmmu_hat_enter(hat);
        hat->sfmmu_rmstat++;
        sfmmu_hat_exit(hatlockp);
        return (1);
}

void
hat_stats_disable(struct hat *hat)
{
        hatlock_t       *hatlockp;

        ASSERT(hat->sfmmu_xhat_provider == NULL);

        hatlockp = sfmmu_hat_enter(hat);
        hat->sfmmu_rmstat--;
        sfmmu_hat_exit(hatlockp);
}
______unchanged_portion_omitted_

/*
 * Hat_share()/unshare() return an (non-zero) error
 * when saddr and daddr are not properly aligned.
 *
```



```
 * The top level mapping element determines the alignment
 * requirement for saddr and daddr, depending on different
 * architectures.
 *
 * When hat_share()/unshare() are not supported,
 * HATOP_SHARE()/UNSHARE() return 0
 */
int
hat_share(struct hat *sfmmup, caddr_t addr,
        struct hat *ism_hatid, caddr_t sptaddr, size_t len, uint_t ismszc)
{
        ism_blk_t       *ism_blkp;
        ism_blk_t       *new_iblk;
        ism_map_t       *ism_map;
        ism_ment_t      *ism_ment;
        int             i, added;
        hatlock_t       *hatlockp;
        int             reload_mmu = 0;
        uint_t          ismshift = page_get_shift(ismszc);
        size_t          ismpgsz = page_get_pagesize(ismszc);
        uint_t          ismmask = (uint_t)ismpgsz - 1;
        size_t          sh_size = ISM_SHIFT(ismshift, len);
        ushort_t        ismhatflag;
        hat_region_cookie_t rcookie;
        sf_scd_t        *old_scdp;

#ifdef DEBUG
        caddr_t         eaddr = addr + len;
#endif /* DEBUG */

        ASSERT(ism_hatid != NULL && sfmmup != NULL);
        ASSERT(sptaddr == ISMID_STARTADDR);
        /*
         * Check the alignment.
         */
        if (!ISM_ALIGNED(ismshift, addr) || !ISM_ALIGNED(ismshift, sptaddr))
                return (EINVAL);

        /*
         * Check size alignment.
         */
        if (!ISM_ALIGNED(ismshift, len))
                return (EINVAL);

        ASSERT(sfmmup->sfmmu_xhat_provider == NULL);

        /*
         * Allocate ism_ment for the ism_hat's mapping list, and an
         * ism map blk in case we need one.  We must do our
         * allocations before acquiring locks to prevent a deadlock
         * in the kmem allocator on the mapping list lock.
         */
        new_iblk = kmem_cache_alloc(ism_blk_cache, KM_SLEEP);
        ism_ment = kmem_cache_alloc(ism_ment_cache, KM_SLEEP);

        /*
         * Serialize ISM mappings with the ISM busy flag, and also the
         * trap handlers.
         */
        sfmmu_ismhat_enter(sfmmup, 0);

        /*
         * Allocate an ism map blk if necessary.
         */
        if (sfmmup->sfmmu_iblk == NULL) {
                sfmmup->sfmmu_iblk = new_iblk;
```

```
8376                 bzero(new_iblk, sizeof (*new_iblk));
8377                 new_iblk->iblk_nextpa = (uint64_t)-1;
8378                 membar_stst();  /* make sure next ptr visible to all CPUs */
8379                 sfmmup->sfmmu_ismblkpa = va_to_pa((caddr_t)new_iblk);
8380                 reload_mmu = 1;
8381                 new_iblk = NULL;
8382         }

8384 #ifdef DEBUG
8385         /*
8386          * Make sure mapping does not already exist.
8387          */
8388         ism_blkp = sfmmup->sfmmu_iblk;
8389         while (ism_blkp != NULL) {
8390                 ism_map = ism_blkp->iblk_maps;
8391                 for (i = 0; i < ISM_MAP_SLOTS && ism_map[i].imap_ismhat; i++) {
8392                         if ((addr >= ism_start(ism_map[i]) &&
8393                             addr < ism_end(ism_map[i])) ||
8394                             eaddr > ism_start(ism_map[i]) &&
8395                             eaddr <= ism_end(ism_map[i])) {
8396                                 panic("sfmmu_share: Already mapped!");
8397                         }
8398                 }
8399                 ism_blkp = ism_blkp->iblk_next;
8400         }
8401 #endif /* DEBUG */

8403         ASSERT(ismszc >= TTE4M);
8404         if (ismszc == TTE4M) {
8405                 ismhatflag = HAT_4M_FLAG;
8406         } else if (ismszc == TTE32M) {
8407                 ismhatflag = HAT_32M_FLAG;
8408         } else if (ismszc == TTE256M) {
8409                 ismhatflag = HAT_256M_FLAG;
8410         }
8411         /*
8412          * Add mapping to first available mapping slot.
8413          */
8414         ism_blkp = sfmmup->sfmmu_iblk;
8415         added = 0;
8416         while (!added) {
8417                 ism_map = ism_blkp->iblk_maps;
8418                 for (i = 0; i < ISM_MAP_SLOTS; i++)  {
8419                         if (ism_map[i].imap_ismhat == NULL) {

8421                                 ism_map[i].imap_ismhat = ism_hatid;
8422                                 ism_map[i].imap_vb_shift = (uchar_t)ismshift;
8423                                 ism_map[i].imap_rid = SFMMU_INVALID_ISMRID;
8424                                 ism_map[i].imap_hatflags = ismhatflag;
8425                                 ism_map[i].imap_sz_mask = ismmask;
8426                                 /*
8427                                  * imap_seg is checked in ISM_CHECK to see if
8428                                  * non-NULL, then other info assumed valid.
8429                                  */
8430                                 membar_stst();
8431                                 ism_map[i].imap_seg = (uintptr_t)addr | sh_size;
8432                                 ism_map[i].imap_ment = ism_ment;

8434                                 /*
8435                                  * Now add ourselves to the ism_hat's
8436                                  * mapping list.
8437                                  */
8438                                 ism_ment->iment_hat = sfmmup;
8439                                 ism_ment->iment_base_va = addr;
8440                                 ism_hatid->sfmmu_ismhat = 1;
8441                                 mutex_enter(&ism_mlist_lock);
```



```
8442                                 iment_add(ism_ment, ism_hatid);
8443                                 mutex_exit(&ism_mlist_lock);
8444                                 added = 1;
8445                                 break;
8446                         }
8447                 }
8448                 if (!added && ism_blkp->iblk_next == NULL) {
8449                         ism_blkp->iblk_next = new_iblk;
8450                         new_iblk = NULL;
8451                         bzero(ism_blkp->iblk_next,
8452                             sizeof (*ism_blkp->iblk_next));
8453                         ism_blkp->iblk_next->iblk_nextpa = (uint64_t)-1;
8454                         membar_stst();
8455                         ism_blkp->iblk_nextpa =
8456                             va_to_pa((caddr_t)ism_blkp->iblk_next);
8457                 }
8458                 ism_blkp = ism_blkp->iblk_next;
8459         }

8461         /*
8462          * After calling hat_join_region, sfmmup may join a new SCD or
8463          * move from the old scd to a new scd, in which case, we want to
8464          * shrink the sfmmup's private tsb size, i.e., pass shrink to
8465          * sfmmu_check_page_sizes at the end of this routine.
8466          */
8467         old_scdp = sfmmup->sfmmu_scdp;

8469         rcookie = hat_join_region(sfmmup, addr, len, (void *)ism_hatid, 0,
8470             PROT_ALL, ismszc, NULL, HAT_REGION_ISM);
8471         if (rcookie != HAT_INVALID_REGION_COOKIE) {
8472                 ism_map[i].imap_rid = (uchar_t)((uint64_t)rcookie);
8473         }
8474         /*
8475          * Update our counters for this sfmmup's ism mappings.
8476          */
8477         for (i = 0; i <= ismszc; i++) {
8478                 if (!(disable_ism_large_pages & (1 << i)))
8479                         (void) ism_tsb_entries(sfmmup, i);
8480         }

8482         /*
8483          * For ISM and DISM we do not support 512K pages, so we only only
8484          * search the 4M and 8K/64K hashes for 4 pagesize cpus, and search the
8485          * 256M or 32M, and 4M and 8K/64K hashes for 6 pagesize cpus.
8486          *
8487          * Need to set 32M/256M ISM flags to make sure
8488          * sfmmu_check_page_sizes() enables them on Panther.
8489          */
8490         ASSERT((disable_ism_large_pages & (1 << TTE512K)) != 0);

8492         switch (ismszc) {
8493         case TTE256M:
8494                 if (!SFMMU_FLAGS_ISSET(sfmmup, HAT_256M_ISM)) {
8495                         hatlockp = sfmmu_hat_enter(sfmmup);
8496                         SFMMU_FLAGS_SET(sfmmup, HAT_256M_ISM);
8497                         sfmmu_hat_exit(hatlockp);
8498                 }
8499                 break;
8500         case TTE32M:
8501                 if (!SFMMU_FLAGS_ISSET(sfmmup, HAT_32M_ISM)) {
8502                         hatlockp = sfmmu_hat_enter(sfmmup);
8503                         SFMMU_FLAGS_SET(sfmmup, HAT_32M_ISM);
8504                         sfmmu_hat_exit(hatlockp);
8505                 }
8506                 break;
8507         default:
```

```
8508                    break;
8509            }

8511            /*
8512             * If we updated the ismblkpa for this HAT we must make
8513             * sure all CPUs running this process reload their tsbmiss area.
8514             * Otherwise they will fail to load the mappings in the tsbmiss
8515             * handler and will loop calling pagefault().
8516             */
8517            if (reload_mmu) {
8518                    hatlockp = sfmmu_hat_enter(sfmmup);
8519                    sfmmu_sync_mmustate(sfmmup);
8520                    sfmmu_hat_exit(hatlockp);
8521            }

8523            sfmmu_ismhat_exit(sfmmup, 0);

8525            /*
8526             * Free up ismblk if we didn't use it.
8527             */
8528            if (new_iblk != NULL)
8529                    kmem_cache_free(ism_blk_cache, new_iblk);

8531            /*
8532             * Check TSB and TLB page sizes.
8533             */
8534            if (sfmmup->sfmmu_scdp != NULL && old_scdp != sfmmup->sfmmu_scdp) {
8535                    sfmmu_check_page_sizes(sfmmup, 0);
8536            } else {
8537                    sfmmu_check_page_sizes(sfmmup, 1);
8538            }
8539            return (0);
8540    }

8542    /*
8543     * hat_unshare removes exactly one ism_map from
8544     * this process's as.  It expects multiple calls
8545     * to hat_unshare for multiple shm segments.
8546     */
8547    void
8548    hat_unshare(struct hat *sfmmup, caddr_t addr, size_t len, uint_t ismszc)
8549    {
8550            ism_map_t       *ism_map;
8551            ism_ment_t      *free_ment = NULL;
8552            ism_blk_t       *ism_blkp;
8553            struct hat      *ism_hatid;
8554            int             found, i;
8555            hatlock_t       *hatlockp;
8556            struct tsb_info *tsbinfo;
8557            uint_t          ismshift = page_get_shift(ismszc);
8558            size_t          sh_size = ISM_SHIFT(ismshift, len);
8559            uchar_t         ism_rid;
8560            sf_scd_t        *old_scdp;

8562            ASSERT(ISM_ALIGNED(ismshift, addr));
8563            ASSERT(ISM_ALIGNED(ismshift, len));
8564            ASSERT(sfmmup != NULL);
8565            ASSERT(sfmmup != ksfmmup);

8869            if (sfmmup->sfmmu_xhat_provider) {
8870                    XHAT_UNSHARE(sfmmup, addr, len);
8871                    return;
8872            } else {
8873                    /*
8874                     * This must be a CPU HAT. If the address space has
8875                     * XHATs attached, inform all XHATs that ISM segment
```



```
8876                     * is going away
8877                     */
8567            ASSERT(sfmmup->sfmmu_as != NULL);
8879                    if (sfmmup->sfmmu_as->a_xhat != NULL)
8880                            xhat_unshare_all(sfmmup->sfmmu_as, addr, len);
8881            }

8569            /*
8570             * Make sure that during the entire time ISM mappings are removed,
8571             * the trap handlers serialize behind us, and that no one else
8572             * can be mucking with ISM mappings.  This also lets us get away
8573             * with not doing expensive cross calls to flush the TLB -- we
8574             * just discard the context, flush the entire TSB, and call it
8575             * a day.
8576             */
8577            sfmmu_ismhat_enter(sfmmup, 0);

8579            /*
8580             * Remove the mapping.
8581             *
8582             * We can't have any holes in the ism map.
8583             * The tsb miss code while searching the ism map will
8584             * stop on an empty map slot.  So we must move
8585             * everyone past the hole up 1 if any.
8586             *
8587             * Also empty ism map blks are not freed until the
8588             * process exits. This is to prevent a MT race condition
8589             * between sfmmu_unshare() and sfmmu_tsbmiss_exception().
8590             */
8591            found = 0;
8592            ism_blkp = sfmmup->sfmmu_iblk;
8593            while (!found && ism_blkp != NULL) {
8594                    ism_map = ism_blkp->iblk_maps;
8595                    for (i = 0; i < ISM_MAP_SLOTS; i++) {
8596                            if (addr == ism_start(ism_map[i]) &&
8597                                sh_size == (size_t)(ism_size(ism_map[i]))) {
8598                                    found = 1;
8599                                    break;
8600                            }
8601                    }
8602                    if (!found)
8603                            ism_blkp = ism_blkp->iblk_next;
8604            }

8606            if (found) {
8607                    ism_hatid = ism_map[i].imap_ismhat;
8608                    ism_rid = ism_map[i].imap_rid;
8609                    ASSERT(ism_hatid != NULL);
8610                    ASSERT(ism_hatid->sfmmu_ismhat == 1);

8612                    /*
8613                     * After hat_leave_region, the sfmmup may leave SCD,
8614                     * in which case, we want to grow the private tsb size when
8615                     * calling sfmmu_check_page_sizes at the end of the routine.
8616                     */
8617                    old_scdp = sfmmup->sfmmu_scdp;
8618                    /*
8619                     * Then remove ourselves from the region.
8620                     */
8621                    if (ism_rid != SFMMU_INVALID_ISMRID) {
8622                            hat_leave_region(sfmmup, (void *)((uint64_t)ism_rid),
8623                                HAT_REGION_ISM);
8624                    }

8626                    /*
8627                     * And now guarantee that any other cpu
```

```
8628                 * that tries to process an ISM miss
8629                 * will go to tl=0.
8630                 */
8631                hatlockp = sfmmu_hat_enter(sfmmup);
8632                sfmmu_invalidate_ctx(sfmmup);
8633                sfmmu_hat_exit(hatlockp);

8635                /*
8636                 * Remove ourselves from the ism mapping list.
8637                 */
8638                mutex_enter(&ism_mlist_lock);
8639                iment_sub(ism_map[i].imap_ment, ism_hatid);
8640                mutex_exit(&ism_mlist_lock);
8641                free_ment = ism_map[i].imap_ment;

8643                /*
8644                 * We delete the ism map by copying
8645                 * the next map over the current one.
8646                 * We will take the next one in the maps
8647                 * array or from the next ism_blk.
8648                 */
8649                while (ism_blkp != NULL) {
8650                        ism_map = ism_blkp->iblk_maps;
8651                        while (i < (ISM_MAP_SLOTS - 1)) {
8652                                ism_map[i] = ism_map[i + 1];
8653                                i++;
8654                        }
8655                        /* i == (ISM_MAP_SLOTS - 1) */
8656                        ism_blkp = ism_blkp->iblk_next;
8657                        if (ism_blkp != NULL) {
8658                                ism_map[i] = ism_blkp->iblk_maps[0];
8659                                i = 0;
8660                        } else {
8661                                ism_map[i].imap_seg = 0;
8662                                ism_map[i].imap_vb_shift = 0;
8663                                ism_map[i].imap_rid = SFMMU_INVALID_ISMRID;
8664                                ism_map[i].imap_hatflags = 0;
8665                                ism_map[i].imap_sz_mask = 0;
8666                                ism_map[i].imap_ismhat = NULL;
8667                                ism_map[i].imap_ment = NULL;
8668                        }
8669                }

8671                /*
8672                 * Now flush entire TSB for the process, since
8673                 * demapping page by page can be too expensive.
8674                 * We don't have to flush the TLB here anymore
8675                 * since we switch to a new TLB ctx instead.
8676                 * Also, there is no need to flush if the process
8677                 * is exiting since the TSB will be freed later.
8678                 */
8679                if (!sfmmup->sfmmu_free) {
8680                        hatlockp = sfmmu_hat_enter(sfmmup);
8681                        for (tsbinfo = sfmmup->sfmmu_tsb; tsbinfo != NULL;
8682                            tsbinfo = tsbinfo->tsb_next) {
8683                                if (tsbinfo->tsb_flags & TSB_SWAPPED)
8684                                        continue;
8685                                if (tsbinfo->tsb_flags & TSB_RELOC_FLAG) {
8686                                        tsbinfo->tsb_flags |=
8687                                            TSB_FLUSH_NEEDED;
8688                                        continue;
8689                                }

8691                                sfmmu_inv_tsb(tsbinfo->tsb_va,
8692                                    TSB_BYTES(tsbinfo->tsb_szc));
8693                        }
```



```
8694                            sfmmu_hat_exit(hatlockp);
8695                    }
8696            }

8698            /*
8699             * Update our counters for this sfmmup's ism mappings.
8700             */
8701            for (i = 0; i <= ismszc; i++) {
8702                    if (!(disable_ism_large_pages & (1 << i)))
8703                            (void) ism_tsb_entries(sfmmup, i);
8704            }

8706            sfmmu_ismhat_exit(sfmmup, 0);

8708            /*
8709             * We must do our freeing here after dropping locks
8710             * to prevent a deadlock in the kmem allocator on the
8711             * mapping list lock.
8712             */
8713            if (free_ment != NULL)
8714                    kmem_cache_free(ism_ment_cache, free_ment);

8716            /*
8717             * Check TSB and TLB page sizes if the process isn't exiting.
8718             */
8719            if (!sfmmup->sfmmu_free) {
8720                    if (found && old_scdp != NULL && sfmmup->sfmmu_scdp == NULL) {
8721                            sfmmu_check_page_sizes(sfmmup, 1);
8722                    } else {
8723                            sfmmu_check_page_sizes(sfmmup, 0);
8724                    }
8725            }
8726 }
_____unchanged_portion_omitted_

8935 #ifdef VAC
8936 /*
8937  * Check for conflicts.
8938  * A conflict exists if the new and existent mappings do not match in
8939  * their "shm_alignment fields. If conflicts exist, the existant mappings
8940  * are flushed unless one of them is locked. If one of them is locked, then
8941  * the mappings are flushed and converted to non-cacheable mappings.
8942  */
8943 static void
8944 sfmmu_vac_conflict(struct hat *hat, caddr_t addr, page_t *pp)
8945 {
8946         struct hat *tmphat;
8947         struct sf_hment *sfhmep, *tmphme = NULL;
8948         struct hme_blk *hmeblkp;
8949         int vcolor;
8950         tte_t tte;

8952         ASSERT(sfmmu_mlist_held(pp));
8953         ASSERT(!PP_ISNC(pp));           /* page better be cacheable */

8955         vcolor = addr_to_vcolor(addr);
8956         if (PP_NEWPAGE(pp)) {
8957                 PP_SET_VCOLOR(pp, vcolor);
8958                 return;
8959         }

8961         if (PP_GET_VCOLOR(pp) == vcolor) {
8962                 return;
8963         }

8965         if (!PP_ISMAPPED(pp) && !PP_ISMAPPED_KPM(pp)) {
```

```
8966                 /*
8967                  * Previous user of page had a different color
8968                  * but since there are no current users
8969                  * we just flush the cache and change the color.
8970                  */
8971                 SFMMU_STAT(sf_pgcolor_conflict);
8972                 sfmmu_cache_flush(pp->p_pagenum, PP_GET_VCOLOR(pp));
8973                 PP_SET_VCOLOR(pp, vcolor);
8974                 return;
8975         }

8977         /*
8978          * If we get here we have a vac conflict with a current
8979          * mapping.  VAC conflict policy is as follows.
8980          * - The default is to unload the other mappings unless:
8981          * - If we have a large mapping we uncache the page.
8982          * We need to uncache the rest of the large page too.
8983          * - If any of the mappings are locked we uncache the page.
8984          * - If the requested mapping is inconsistent
8985          * with another mapping and that mapping
8986          * is in the same address space we have to
8987          * make it non-cached.  The default thing
8988          * to do is unload the inconsistent mapping
8989          * but if they are in the same address space
8990          * we run the risk of unmapping the pc or the
8991          * stack which we will use as we return to the user,
8992          * in which case we can then fault on the thing
8993          * we just unloaded and get into an infinite loop.
8994          */
8995         if (PP_ISMAPPED_LARGE(pp)) {
8996                 int sz;

8998                 /*
8999                  * Existing mapping is for big pages. We don't unload
9000                  * existing big mappings to satisfy new mappings.
9001                  * Always convert all mappings to TNC.
9002                  */
9003                 sz = fnd_mapping_sz(pp);
9004                 pp = PP_GROUPLEADER(pp, sz);
9005                 SFMMU_STAT_ADD(sf_uncache_conflict, TTEPAGES(sz));
9006                 sfmmu_page_cache_array(pp, HAT_TMPNC, CACHE_FLUSH,
9007                     TTEPAGES(sz));

9009                 return;
9010         }

9012         /*
9013          * check if any mapping is in same as or if it is locked
9014          * since in that case we need to uncache.
9015          */
9016         for (sfhmep = pp->p_mapping; sfhmep; sfhmep = tmphme) {
9017                 tmphme = sfhmep->hme_next;
9018                 if (IS_PAHME(sfhmep))
9019                         continue;
9020                 hmeblkp = sfmmu_hmetohblk(sfhmep);
9335                 if (hmeblkp->hblk_xhat_bit)
9336                         continue;
9021                 tmphat = hblktosfmmu(hmeblkp);
9022                 sfmmu_copytte(&sfhmep->hme_tte, &tte);
9023                 ASSERT(TTE_IS_VALID(&tte));
9024                 if (hmeblkp->hblk_shared || tmphat == hat ||
9025                     hmeblkp->hblk_lckcnt) {
9026                         /*
9027                          * We have an uncache conflict
9028                          */
9029                         SFMMU_STAT(sf_uncache_conflict);
```



```
9030                         sfmmu_page_cache_array(pp, HAT_TMPNC, CACHE_FLUSH, 1);
9031                         return;
9032                 }
9033         }

9035         /*
9036          * We have an unload conflict
9037          * We have already checked for LARGE mappings, therefore
9038          * the remaining mapping(s) must be TTE8K.
9039          */
9040         SFMMU_STAT(sf_unload_conflict);

9042         for (sfhmep = pp->p_mapping; sfhmep; sfhmep = tmphme) {
9043                 tmphme = sfhmep->hme_next;
9044                 if (IS_PAHME(sfhmep))
9045                         continue;
9046                 hmeblkp = sfmmu_hmetohblk(sfhmep);
9363                 if (hmeblkp->hblk_xhat_bit)
9364                         continue;
9047                 ASSERT(!hmeblkp->hblk_shared);
9048                 (void) sfmmu_pageunload(pp, sfhmep, TTE8K);
9049         }

9051         if (PP_ISMAPPED_KPM(pp))
9052                 sfmmu_kpm_vac_unload(pp, addr);

9054         /*
9055          * Unloads only do TLB flushes so we need to flush the
9056          * cache here.
9057          */
9058         sfmmu_cache_flush(pp->p_pagenum, PP_GET_VCOLOR(pp));
9059         PP_SET_VCOLOR(pp, vcolor);
9060 }
______unchanged_portion_omitted_

9147 /*
9148  * Returns 1 if page(s) can be converted from TNC to cacheable setting,
9149  * returns 0 otherwise. Note that oaddr argument is valid for only
9150  * 8k pages.
9151  */
9152 int
9153 tst_tnc(page_t *pp, pgcnt_t npages)
9154 {
9155         struct  sf_hment *sfhme;
9156         struct  hme_blk *hmeblkp;
9157         tte_t   tte;
9158         caddr_t vaddr;
9159         int     clr_valid = 0;
9160         int     color, color1, bcolor;
9161         int     i, ncolors;

9163         ASSERT(pp != NULL);
9164         ASSERT(!(cache & CACHE_WRITEBACK));

9166         if (npages > 1) {
9167                 ncolors = CACHE_NUM_COLOR;
9168         }

9170         for (i = 0; i < npages; i++) {
9171                 ASSERT(sfmmu_mlist_held(pp));
9172                 ASSERT(PP_ISTNC(pp));
9173                 ASSERT(PP_GET_VCOLOR(pp) == NO_VCOLOR);

9175                 if (PP_ISPNC(pp)) {
9176                         return (0);
9177                 }
```

```
9179                clr_valid = 0;
9180                if (PP_ISMAPPED_KPM(pp)) {
9181                        caddr_t kpmvaddr;

9183                        ASSERT(kpm_enable);
9184                        kpmvaddr = hat_kpm_page2va(pp, 1);
9185                        ASSERT(!(npages > 1 && IS_KPM_ALIAS_RANGE(kpmvaddr)));
9186                        color1 = addr_to_vcolor(kpmvaddr);
9187                        clr_valid = 1;
9188                }

9190                for (sfhme = pp->p_mapping; sfhme; sfhme = sfhme->hme_next) {
9191                        if (IS_PAHME(sfhme))
9192                                continue;
9193                        hmeblkp = sfmmu_hmetohblk(sfhme);
9512                        if (hmeblkp->hblk_xhat_bit)
9513                                continue;

9195                        sfmmu_copytte(&sfhme->hme_tte, &tte);
9196                        ASSERT(TTE_IS_VALID(&tte));

9198                        vaddr = tte_to_vaddr(hmeblkp, tte);
9199                        color = addr_to_vcolor(vaddr);

9201                        if (npages > 1) {
9202                                /*
9203                                 * If there is a big mapping, make sure
9204                                 * 8K mapping is consistent with the big
9205                                 * mapping.
9206                                 */
9207                                bcolor = i % ncolors;
9208                                if (color != bcolor) {
9209                                        return (0);
9210                                }
9211                        }
9212                        if (!clr_valid) {
9213                                clr_valid = 1;
9214                                color1 = color;
9215                        }

9217                        if (color1 != color) {
9218                                return (0);
9219                        }
9220                }

9222                pp = PP_PAGENEXT(pp);
9223        }

9225        return (1);
9226 }
```

*_____unchanged_portion_omitted_*

```
9312 /*
9313  * This function changes the virtual cacheability of all mappings to a
9314  * particular page.  When changing from uncache to cacheable the mappings will
9315  * only be changed if all of them have the same virtual color.
9316  * We need to flush the cache in all cpus.  It is possible that
9317  * a process referenced a page as cacheable but has sinced exited
9318  * and cleared the mapping list.  We still to flush it but have no
9319  * state so all cpus is the only alternative.
9320  */
9321 static void
9322 sfmmu_page_cache(page_t *pp, int flags, int cache_flush_flag, int bcolor)
9323 {
9324         struct  sf_hment *sfhme;
```



```
9325         struct  hme_blk *hmeblkp;
9326         sfmmu_t *sfmmup;
9327         tte_t   tte, ttemod;
9328         caddr_t vaddr;
9329         int     ret, color;
9330         pfn_t   pfn;

9332         color = bcolor;
9333         pfn = pp->p_pagenum;

9335         for (sfhme = pp->p_mapping; sfhme; sfhme = sfhme->hme_next) {

9337                 if (IS_PAHME(sfhme))
9338                         continue;
9339                 hmeblkp = sfmmu_hmetohblk(sfhme);

9661                 if (hmeblkp->hblk_xhat_bit)
9662                         continue;

9341                 sfmmu_copytte(&sfhme->hme_tte, &tte);
9342                 ASSERT(TTE_IS_VALID(&tte));
9343                 vaddr = tte_to_vaddr(hmeblkp, tte);
9344                 color = addr_to_vcolor(vaddr);

9346 #ifdef DEBUG
9347                 if ((flags & HAT_CACHE) && bcolor != NO_VCOLOR) {
9348                         ASSERT(color == bcolor);
9349                 }
9350 #endif

9352                 ASSERT(flags != HAT_TMPNC || color == PP_GET_VCOLOR(pp));

9354                 ttemod = tte;
9355                 if (flags & (HAT_UNCACHE | HAT_TMPNC)) {
9356                         TTE_CLR_VCACHEABLE(&ttemod);
9357                 } else {        /* flags & HAT_CACHE */
9358                         TTE_SET_VCACHEABLE(&ttemod);
9359                 }
9360                 ret = sfmmu_modifytte_try(&tte, &ttemod, &sfhme->hme_tte);
9361                 if (ret < 0) {
9362                         /*
9363                          * Since all cpus are captured modifytte should not
9364                          * fail.
9365                          */
9366                         panic("sfmmu_page_cache: write to tte failed");
9367                 }

9369                 sfmmup = hblktosfmmu(hmeblkp);
9370                 if (cache_flush_flag == CACHE_FLUSH) {
9371                         /*
9372                          * Flush TSBs, TLBs and caches
9373                          */
9374                         if (hmeblkp->hblk_shared) {
9375                                 sf_srd_t *srdp = (sf_srd_t *)sfmmup;
9376                                 uint_t rid = hmeblkp->hblk_tag.htag_rid;
9377                                 sf_region_t *rgnp;
9378                                 ASSERT(SFMMU_IS_SHMERID_VALID(rid));
9379                                 ASSERT(rid < SFMMU_MAX_HME_REGIONS);
9380                                 ASSERT(srdp != NULL);
9381                                 rgnp = srdp->srd_hmergnp[rid];
9382                                 SFMMU_VALIDATE_SHAREDHBLK(hmeblkp,
9383                                     srdp, rgnp, rid);
9384                                 (void) sfmmu_rgntlb_demap(vaddr, rgnp,
9385                                     hmeblkp, 0);
9386                                 sfmmu_cache_flush(pfn, addr_to_vcolor(vaddr));
9387                         } else if (sfmmup->sfmmu_ismhat) {
```

```
9388                                if (flags & HAT_CACHE) {
9389                                        SFMMU_STAT(sf_ism_recache);
9390                                } else {
9391                                        SFMMU_STAT(sf_ism_uncache);
9392                                }
9393                                sfmmu_ismtlbcache_demap(vaddr, sfmmup, hmeblkp,
9394                                    pfn, CACHE_FLUSH);
9395                        } else {
9396                                sfmmu_tlbcache_demap(vaddr, sfmmup, hmeblkp,
9397                                    pfn, 0, FLUSH_ALL_CPUS, CACHE_FLUSH, 1);
9398                        }

9400                        /*
9401                         * all cache entries belonging to this pfn are
9402                         * now flushed.
9403                         */
9404                        cache_flush_flag = CACHE_NO_FLUSH;
9405                } else {
9406                        /*
9407                         * Flush only TSBs and TLBs.
9408                         */
9409                        if (hmeblkp->hblk_shared) {
9410                                sf_srd_t *srdp = (sf_srd_t *)sfmmup;
9411                                uint_t rid = hmeblkp->hblk_tag.htag_rid;
9412                                sf_region_t *rgnp;
9413                                ASSERT(SFMMU_IS_SHMERID_VALID(rid));
9414                                ASSERT(rid < SFMMU_MAX_HME_REGIONS);
9415                                ASSERT(srdp != NULL);
9416                                rgnp = srdp->srd_hmergnp[rid];
9417                                SFMMU_VALIDATE_SHAREDHBLK(hmeblkp,
9418                                    srdp, rgnp, rid);
9419                                (void) sfmmu_rgntlb_demap(vaddr, rgnp,
9420                                    hmeblkp, 0);
9421                        } else if (sfmmup->sfmmu_ismhat) {
9422                                if (flags & HAT_CACHE) {
9423                                        SFMMU_STAT(sf_ism_recache);
9424                                } else {
9425                                        SFMMU_STAT(sf_ism_uncache);
9426                                }
9427                                sfmmu_ismtlbcache_demap(vaddr, sfmmup, hmeblkp,
9428                                    pfn, CACHE_NO_FLUSH);
9429                        } else {
9430                                sfmmu_tlb_demap(vaddr, sfmmup, hmeblkp, 0, 1);
9431                        }
9432                }
9433        }

9435        if (PP_ISMAPPED_KPM(pp))
9436                sfmmu_kpm_page_cache(pp, flags, cache_flush_flag);

9438        switch (flags) {

9440                default:
9441                        panic("sfmmu_pagecache: unknown flags");
9442                        break;

9444                case HAT_CACHE:
9445                        PP_CLRTNC(pp);
9446                        PP_CLRPNC(pp);
9447                        PP_SET_VCOLOR(pp, color);
9448                        break;

9450                case HAT_TMPNC:
9451                        PP_SETTNC(pp);
9452                        PP_SET_VCOLOR(pp, NO_VCOLOR);
9453                        break;
```



```
9455                case HAT_UNCACHE:
9456                        PP_SETPNC(pp);
9457                        PP_CLRTNC(pp);
9458                        PP_SET_VCOLOR(pp, NO_VCOLOR);
9459                        break;
9460        }
9461 }
```

_____***unchanged_portion_omitted_***

```
9674 /*
9675  * Replace the specified TSB with a new TSB.  This function gets called when
9676  * we grow, or shrink a TSB.  When swapping in a TSB (TSB_SWAPIN), the
9999  * we grow, shrink or swapin a TSB.  When swapping in a TSB (TSB_SWAPIN), the
9677  * TSB_FORCEALLOC flag may be used to force allocation of a minimum-sized TSB
9678  * (8K).
9679  *
9680  * Caller must hold the HAT lock, but should assume any tsb_info
9681  * pointers it has are no longer valid after calling this function.
9682  *
9683  * Return values:
9684  *      TSB_ALLOCFAIL   Failed to allocate a TSB, due to memory constraints
9685  *      TSB_LOSTRACE    HAT is busy, i.e. another thread is already doing
9686  *                      something to this tsbinfo/TSB
9687  *      TSB_SUCCESS     Operation succeeded
9688  */
9689 static tsb_replace_rc_t
9690 sfmmu_replace_tsb(sfmmu_t *sfmmup, struct tsb_info *old_tsbinfo, uint_t szc,
9691     hatlock_t *hatlockp, uint_t flags)
9692 {
9693         struct tsb_info *new_tsbinfo = NULL;
9694         struct tsb_info *curtsb, *prevtsb;
9695         uint_t tte_sz_mask;
9696         int i;

9698         ASSERT(sfmmup != ksfmmup);
9699         ASSERT(sfmmup->sfmmu_ismhat == 0);
9700         ASSERT(sfmmu_hat_lock_held(sfmmup));
9701         ASSERT(szc <= tsb_max_growsize);

9703         if (SFMMU_FLAGS_ISSET(sfmmup, HAT_BUSY))
9704                 return (TSB_LOSTRACE);

9706         /*
9707          * Find the tsb_info ahead of this one in the list, and
9708          * also make sure that the tsb_info passed in really
9709          * exists!
9710          */
9711         for (prevtsb = NULL, curtsb = sfmmup->sfmmu_tsb;
9712             curtsb != old_tsbinfo && curtsb != NULL;
9713             prevtsb = curtsb, curtsb = curtsb->tsb_next)
9714                 ;
9715         ASSERT(curtsb != NULL);

9717         if (!(flags & TSB_SWAPIN) && SFMMU_FLAGS_ISSET(sfmmup, HAT_SWAPPED)) {
9718                 /*
9719                  * The process is swapped out, so just set the new size
9720                  * code.  When it swaps back in, we'll allocate a new one
9721                  * of the new chosen size.
9722                  */
9723                 curtsb->tsb_szc = szc;
9724                 return (TSB_SUCCESS);
9725         }
9726         SFMMU_FLAGS_SET(sfmmup, HAT_BUSY);
```

```
9728         tte_sz_mask = old_tsbinfo->tsb_ttesz_mask;

9730         /*
9731          * All initialization is done inside of sfmmu_tsbinfo_alloc().
9732          * If we fail to allocate a TSB, exit.
9733          *
9734          * If tsb grows with new tsb size > 4M and old tsb size < 4M,
9735          * then try 4M slab after the initial alloc fails.
9736          *
9737          * If tsb swapin with tsb size > 4M, then try 4M after the
9738          * initial alloc fails.
9739          */
9740         sfmmu_hat_exit(hatlockp);
9741         if (sfmmu_tsbinfo_alloc(&new_tsbinfo, szc,
9742             tte_sz_mask, flags, sfmmup) &&
9743             (!(flags & (TSB_GROW | TSB_SWAPIN)) || (szc <= TSB_4M_SZCODE) ||
9744             (!(flags & TSB_SWAPIN) &&
9745             (old_tsbinfo->tsb_szc >= TSB_4M_SZCODE)) ||
9746             sfmmu_tsbinfo_alloc(&new_tsbinfo, TSB_4M_SZCODE,
9747             tte_sz_mask, flags, sfmmup))) {
9748                 (void) sfmmu_hat_enter(sfmmup);
9749                 if (!(flags & TSB_SWAPIN))
9750                         SFMMU_STAT(sf_tsb_resize_failures);
9751                 SFMMU_FLAGS_CLEAR(sfmmup, HAT_BUSY);
9752                 return (TSB_ALLOCFAIL);
9753         }
9754         (void) sfmmu_hat_enter(sfmmup);

9756         /*
9757          * Re-check to make sure somebody else didn't muck with us while we
9758          * didn't hold the HAT lock.  If the process swapped out, fine, just
9759          * exit; this can happen if we try to shrink the TSB from the context
9760          * of another process (such as on an ISM unmap), though it is rare.
9761          */
9762         if (!(flags & TSB_SWAPIN) && SFMMU_FLAGS_ISSET(sfmmup, HAT_SWAPPED)) {
9763                 SFMMU_STAT(sf_tsb_resize_failures);
9764                 SFMMU_FLAGS_CLEAR(sfmmup, HAT_BUSY);
9765                 sfmmu_hat_exit(hatlockp);
9766                 sfmmu_tsbinfo_free(new_tsbinfo);
9767                 (void) sfmmu_hat_enter(sfmmup);
9768                 return (TSB_LOSTRACE);
9769         }

9771 #ifdef  DEBUG
9772         /* Reverify that the tsb_info still exists.. for debugging only */
9773         for (prevtsb = NULL, curtsb = sfmmup->sfmmu_tsb;
9774             curtsb != old_tsbinfo && curtsb != NULL;
9775             prevtsb = curtsb, curtsb = curtsb->tsb_next)
9776                 ;
9777         ASSERT(curtsb != NULL);
9778 #endif  /* DEBUG */

9780         /*
9781          * Quiesce any CPUs running this process on their next TLB miss
9782          * so they atomically see the new tsb_info.  We temporarily set the
9783          * context to invalid context so new threads that come on processor
9784          * after we do the xcall to cpusran will also serialize behind the
9785          * HAT lock on TLB miss and will see the new TSB.  Since this short
9786          * race with a new thread coming on processor is relatively rare,
9787          * this synchronization mechanism should be cheaper than always
9788          * pausing all CPUs for the duration of the setup, which is what
9789          * the old implementation did.  This is particularly true if we are
9790          * copying a huge chunk of memory around during that window.
9791          *
9792          * The memory barriers are to make sure things stay consistent
9793          * with resume() since it does not hold the HAT lock while
```



```
9794          * walking the list of tsb_info structures.
9795          */
9796         if ((flags & TSB_SWAPIN) != TSB_SWAPIN) {
9797                 /* The TSB is either growing or shrinking. */
9798                 sfmmu_invalidate_ctx(sfmmup);
9799         } else {
9800                 /*
9801                  * It is illegal to swap in TSBs from a process other
9802                  * than a process being swapped in.  This in turn
9803                  * implies we do not have a valid MMU context here
9804                  * since a process needs one to resolve translation
9805                  * misses.
9806                  */
9807                 ASSERT(curthread->t_procp->p_as->a_hat == sfmmup);
9808         }

9810 #ifdef DEBUG
9811         ASSERT(max_mmu_ctxdoms > 0);

9813         /*
9814          * Process should have INVALID_CONTEXT on all MMUs
9815          */
9816         for (i = 0; i < max_mmu_ctxdoms; i++) {

9818                 ASSERT(sfmmup->sfmmu_ctxs[i].cnum == INVALID_CONTEXT);
9819         }
9820 #endif

9822         new_tsbinfo->tsb_next = old_tsbinfo->tsb_next;
9823         membar_stst();  /* strict ordering required */
9824         if (prevtsb)
9825                 prevtsb->tsb_next = new_tsbinfo;
9826         else
9827                 sfmmup->sfmmu_tsb = new_tsbinfo;
9828         membar_enter(); /* make sure new TSB globally visible */

9830         /*
9831          * We need to migrate TSB entries from the old TSB to the new TSB
9832          * if tsb_remap_ttes is set and the TSB is growing.
9833          */
9834         if (tsb_remap_ttes && ((flags & TSB_GROW) == TSB_GROW))
9835                 sfmmu_copy_tsb(old_tsbinfo, new_tsbinfo);

9837         SFMMU_FLAGS_CLEAR(sfmmup, HAT_BUSY);

9839         /*
9840          * Drop the HAT lock to free our old tsb_info.
9841          */
9842         sfmmu_hat_exit(hatlockp);

9844         if ((flags & TSB_GROW) == TSB_GROW) {
9845                 SFMMU_STAT(sf_tsb_grow);
9846         } else if ((flags & TSB_SHRINK) == TSB_SHRINK) {
9847                 SFMMU_STAT(sf_tsb_shrink);
9848         }

9850         sfmmu_tsbinfo_free(old_tsbinfo);

9852         (void) sfmmu_hat_enter(sfmmup);
9853         return (TSB_SUCCESS);
9854 }
```

_____***unchanged_portion_omitted_***

```
13079 /*
13080  * This function is currently not supported on this platform. For what
13081  * it's supposed to do, see hat.c and hat_srmmu.c
```

```
13082  */
13083 /* ARGSUSED */
13084 faultcode_t
13085 hat_softlock(struct hat *hat, caddr_t addr, size_t *lenp, page_t **ppp,
13086     uint_t flags)
13087 {
13411         ASSERT(hat->sfmmu_xhat_provider == NULL);
13088         return (FC_NOSUPPORT);
13089 }

13091 /*
13092  * Searchs the mapping list of the page for a mapping of the same size. If not
13093  * found the corresponding bit is cleared in the p_index field. When large
13094  * pages are more prevalent in the system, we can maintain the mapping list
13095  * in order and we don't have to traverse the list each time. Just check the
13096  * next and prev entries, and if both are of different size, we clear the bit.
13097  */
13098 static void
13099 sfmmu_rm_large_mappings(page_t *pp, int ttesz)
13100 {
13101         struct sf_hment *sfhmep;
13102         struct hme_blk *hmeblkp;
13103         int     index;
13104         pgcnt_t npgs;

13106         ASSERT(ttesz > TTE8K);

13108         ASSERT(sfmmu_mlist_held(pp));

13110         ASSERT(PP_ISMAPPED_LARGE(pp));

13112         /*
13113          * Traverse mapping list looking for another mapping of same size.
13114          * since we only want to clear index field if all mappings of
13115          * that size are gone.
13116          */

13118         for (sfhmep = pp->p_mapping; sfhmep; sfhmep = sfhmep->hme_next) {
13119                 if (IS_PAHME(sfhmep))
13120                         continue;
13121                 hmeblkp = sfmmu_hmetohblk(sfhmep);
13446                 if (hmeblkp->hblk_xhat_bit)
13447                         continue;
13122                 if (hme_size(sfhmep) == ttesz) {
13123                         /*
13124                          * another mapping of the same size. don't clear index.
13125                          */
13126                         return;
13127                 }
13128         }

13130         /*
13131          * Clear the p_index bit for large page.
13132          */
13133         index = PAGESZ_TO_INDEX(ttesz);
13134         npgs = TTEPAGES(ttesz);
13135         while (npgs-- > 0) {
13136                 ASSERT(pp->p_index & index);
13137                 pp->p_index &= ~index;
13138                 pp = PP_PAGENEXT(pp);
13139         }
13140 }
```

unchanged_portion_omitted_

```
13655 /*
13656  * The caller makes sure hat_join_region()/hat_leave_region() can't be called
```



```
13657  * at the same time for the same process and address range. This is ensured by
13658  * the fact that address space is locked as writer when a process joins the
13659  * regions. Therefore there's no need to hold an srd lock during the entire
13660  * execution of hat_join_region()/hat_leave_region().
13661  */

13663 #define RGN_HASH_FUNCTION(obj)  (((((uintptr_t)(obj)) >> 4) ^ \
13664                                   (((uintptr_t)(obj)) >> 11)) & \
13665                                       srd_rgn_hashmask)
13666 /*
13667  * This routine implements the shared context functionality required when
13668  * attaching a segment to an address space. It must be called from
13669  * hat_share() for D(ISM) segments and from segvn_create() for segments
13670  * with the MAP_PRIVATE and MAP_TEXT flags set. It returns a region_cookie
13671  * which is saved in the private segment data for hme segments and
13672  * the ism_map structure for ism segments.
13673  */
13674 hat_region_cookie_t
13675 hat_join_region(struct hat *sfmmup,
13676         caddr_t r_saddr,
13677         size_t r_size,
13678         void *r_obj,
13679         u_offset_t r_objoff,
13680         uchar_t r_perm,
13681         uchar_t r_pgszc,
13682         hat_rgn_cb_func_t r_cb_function,
13683         uint_t flags)
13684 {
13685         sf_srd_t *srdp = sfmmup->sfmmu_srdp;
13686         uint_t rhash;
13687         uint_t rid;
13688         hatlock_t *hatlockp;
13689         sf_region_t *rgnp;
13690         sf_region_t *new_rgnp = NULL;
13691         int i;
13692         uint16_t *nextidp;
13693         sf_region_t **freelistp;
13694         int maxids;
13695         sf_region_t **rarrp;
13696         uint16_t *busyrgnsp;
13697         ulong_t rttecnt;
13698         uchar_t tteflag;
13699         uchar_t r_type = flags & HAT_REGION_TYPE_MASK;
13700         int text = (r_type == HAT_REGION_TEXT);

13702         if (srdp == NULL || r_size == 0) {
13703                 return (HAT_INVALID_REGION_COOKIE);
13704         }

14032         ASSERT(sfmmup->sfmmu_xhat_provider == NULL);
13706         ASSERT(sfmmup != ksfmmup);
13707         ASSERT(AS_WRITE_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));
13708         ASSERT(srdp->srd_refcnt > 0);
13709         ASSERT(!(flags & ~HAT_REGION_TYPE_MASK));
13710         ASSERT(flags == HAT_REGION_TEXT || flags == HAT_REGION_ISM);
13711         ASSERT(r_pgszc < mmu_page_sizes);
13712         if (!IS_P2ALIGNED(r_saddr, TTEBYTES(r_pgszc)) ||
13713             !IS_P2ALIGNED(r_size, TTEBYTES(r_pgszc))) {
13714                 panic("hat_join_region: region addr or size is not aligned\n");
13715         }


13718         r_type = (r_type == HAT_REGION_ISM) ? SFMMU_REGION_ISM :
13719             SFMMU_REGION_HME;
13720         /*
13721          * Currently only support shared hmes for the read only main text
```

```
13722          * region.
13723          */
13724         if (r_type == SFMMU_REGION_HME && ((r_obj != srdp->srd_evp) ||
13725             (r_perm & PROT_WRITE))) {
13726                 return (HAT_INVALID_REGION_COOKIE);
13727         }

13729         rhash = RGN_HASH_FUNCTION(r_obj);

13731         if (r_type == SFMMU_REGION_ISM) {
13732                 nextidp = &srdp->srd_next_ismrid;
13733                 freelistp = &srdp->srd_ismrgnfree;
13734                 maxids = SFMMU_MAX_ISM_REGIONS;
13735                 rarrp = srdp->srd_ismrgnp;
13736                 busyrgnsp = &srdp->srd_ismbusyrgns;
13737         } else {
13738                 nextidp = &srdp->srd_next_hmerid;
13739                 freelistp = &srdp->srd_hmergnfree;
13740                 maxids = SFMMU_MAX_HME_REGIONS;
13741                 rarrp = srdp->srd_hmergnp;
13742                 busyrgnsp = &srdp->srd_hmebusyrgns;
13743         }

13745         mutex_enter(&srdp->srd_mutex);

13747         for (rgnp = srdp->srd_rgnhash[rhash]; rgnp != NULL;
13748             rgnp = rgnp->rgn_hash) {
13749                 if (rgnp->rgn_saddr == r_saddr && rgnp->rgn_size == r_size &&
13750                     rgnp->rgn_obj == r_obj && rgnp->rgn_objoff == r_objoff &&
13751                     rgnp->rgn_perm == r_perm && rgnp->rgn_pgszc == r_pgszc) {
13752                         break;
13753                 }
13754         }

13756 rfound:
13757         if (rgnp != NULL) {
13758                 ASSERT((rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK) == r_type);
13759                 ASSERT(rgnp->rgn_cb_function == r_cb_function);
13760                 ASSERT(rgnp->rgn_refcnt >= 0);
13761                 rid = rgnp->rgn_id;
13762                 ASSERT(rid < maxids);
13763                 ASSERT(rarrp[rid] == rgnp);
13764                 ASSERT(rid < *nextidp);
13765                 atomic_inc_32((volatile uint_t *)&rgnp->rgn_refcnt);
13766                 mutex_exit(&srdp->srd_mutex);
13767                 if (new_rgnp != NULL) {
13768                         kmem_cache_free(region_cache, new_rgnp);
13769                 }
13770                 if (r_type == SFMMU_REGION_HME) {
13771                         int myjoin =
13772                             (sfmmup == astosfmmu(curthread->t_procp->p_as));

13774                         sfmmu_link_to_hmeregion(sfmmup, rgnp);
13775                         /*
13776                          * bitmap should be updated after linking sfmmu on
13777                          * region list so that pageunload() doesn't skip
13778                          * TSB/TLB flush. As soon as bitmap is updated another
13779                          * thread in this process can already start accessing
13780                          * this region.
13781                          */
13782                         /*
13783                          * Normally ttecnt accounting is done as part of
13784                          * pagefault handling. But a process may not take any
13785                          * pagefaults on shared hmeblks created by some other
13786                          * process. To compensate for this assume that the
13787                          * entire region will end up faulted in using
```



```
13788                          * the region's pagesize.
13789                          *
13790                          */
13791                         if (r_pgszc > TTE8K) {
13792                                 tteflag = 1 << r_pgszc;
13793                                 if (disable_large_pages & tteflag) {
13794                                         tteflag = 0;
13795                                 }
13796                         } else {
13797                                 tteflag = 0;
13798                         }
13799                         if (tteflag && !(sfmmup->sfmmu_rtteflags & tteflag)) {
13800                                 hatlockp = sfmmu_hat_enter(sfmmup);
13801                                 sfmmup->sfmmu_rtteflags |= tteflag;
13802                                 sfmmu_hat_exit(hatlockp);
13803                         }
13804                         hatlockp = sfmmu_hat_enter(sfmmup);

13806                         /*
13807                          * Preallocate 1/4 of ttecnt's in 8K TSB for >= 4M
13808                          * region to allow for large page allocation failure.
13809                          */
13810                         if (r_pgszc >= TTE4M) {
13811                                 sfmmup->sfmmu_tsb0_4minflcnt +=
13812                                     r_size >> (TTE_PAGE_SHIFT(TTE8K) + 2);
13813                         }

13815                         /* update sfmmu_ttecnt with the shme rgn ttecnt */
13816                         rttecnt = r_size >> TTE_PAGE_SHIFT(r_pgszc);
13817                         atomic_add_long(&sfmmup->sfmmu_ttecnt[r_pgszc],
13818                             rttecnt);

13820                         if (text && r_pgszc >= TTE4M &&
13821                             (tteflag || ((disable_large_pages >> TTE4M) &
13822                             ((1 << (r_pgszc - TTE4M + 1)) - 1))) &&
13823                             !SFMMU_FLAGS_ISSET(sfmmup, HAT_4MTEXT_FLAG)) {
13824                                 SFMMU_FLAGS_SET(sfmmup, HAT_4MTEXT_FLAG);
13825                         }

13827                         sfmmu_hat_exit(hatlockp);
13828                         /*
13829                          * On Panther we need to make sure TLB is programmed
13830                          * to accept 32M/256M pages.  Call
13831                          * sfmmu_check_page_sizes() now to make sure TLB is
13832                          * setup before making hmeregions visible to other
13833                          * threads.
13834                          */
13835                         sfmmu_check_page_sizes(sfmmup, 1);
13836                         hatlockp = sfmmu_hat_enter(sfmmup);
13837                         SF_RGNMAP_ADD(sfmmup->sfmmu_hmeregion_map, rid);

13839                         /*
13840                          * if context is invalid tsb miss exception code will
13841                          * call sfmmu_check_page_sizes() and update tsbmiss
13842                          * area later.
13843                          */
13844                         kpreempt_disable();
13845                         if (myjoin &&
13846                             (sfmmup->sfmmu_ctxs[CPU_MMU_IDX(CPU)].cnum
13847                             != INVALID_CONTEXT)) {
13848                                 struct tsbmiss *tsbmp;

13850                                 tsbmp = &tsbmiss_area[CPU->cpu_id];
13851                                 ASSERT(sfmmup == tsbmp->usfmmup);
13852                                 BT_SET(tsbmp->shmermap, rid);
13853                                 if (r_pgszc > TTE64K) {
```

```
13854                                       tsbmp->uhat_rtteflags |= tteflag;
13855                                   }
13856
13857                           }
13858                           kpreempt_enable();
13859
13860                           sfmmu_hat_exit(hatlockp);
13861                           ASSERT((hat_region_cookie_t)((uint64_t)rid) !=
13862                               HAT_INVALID_REGION_COOKIE);
13863                   } else {
13864                           hatlockp = sfmmu_hat_enter(sfmmup);
13865                           SF_RGNMAP_ADD(sfmmup->sfmmu_ismregion_map, rid);
13866                           sfmmu_hat_exit(hatlockp);
13867                   }
13868                   ASSERT(rid < maxids);
13869
13870                   if (r_type == SFMMU_REGION_ISM) {
13871                           sfmmu_find_scd(sfmmup);
13872                   }
13873                   return ((hat_region_cookie_t)((uint64_t)rid));
13874           }
13875
13876           ASSERT(new_rgnp == NULL);
13877
13878           if (*busyrgnsp >= maxids) {
13879                   mutex_exit(&srdp->srd_mutex);
13880                   return (HAT_INVALID_REGION_COOKIE);
13881           }
13882
13883           ASSERT(MUTEX_HELD(&srdp->srd_mutex));
13884           if (*freelistp != NULL) {
13885                   rgnp = *freelistp;
13886                   *freelistp = rgnp->rgn_next;
13887                   ASSERT(rgnp->rgn_id < *nextidp);
13888                   ASSERT(rgnp->rgn_id < maxids);
13889                   ASSERT(rgnp->rgn_flags & SFMMU_REGION_FREE);
13890                   ASSERT((rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK)
13891                       == r_type);
13892                   ASSERT(rarrp[rgnp->rgn_id] == rgnp);
13893                   ASSERT(rgnp->rgn_hmeflags == 0);
13894           } else {
13895                   /*
13896                    * release local locks before memory allocation.
13897                    */
13898                   mutex_exit(&srdp->srd_mutex);
13899
13900                   new_rgnp = kmem_cache_alloc(region_cache, KM_SLEEP);
13901
13902                   mutex_enter(&srdp->srd_mutex);
13903                   for (rgnp = srdp->srd_rgnhash[rhash]; rgnp != NULL;
13904                       rgnp = rgnp->rgn_hash) {
13905                           if (rgnp->rgn_saddr == r_saddr &&
13906                               rgnp->rgn_size == r_size &&
13907                               rgnp->rgn_obj == r_obj &&
13908                               rgnp->rgn_objoff == r_objoff &&
13909                               rgnp->rgn_perm == r_perm &&
13910                               rgnp->rgn_pgszc == r_pgszc) {
13911                                   break;
13912                           }
13913                   }
13914                   if (rgnp != NULL) {
13915                           goto rfound;
13916                   }
13917
13918                   if (*nextidp >= maxids) {
13919                           mutex_exit(&srdp->srd_mutex);
```



```
13920                           goto fail;
13921                   }
13922                   rgnp = new_rgnp;
13923                   new_rgnp = NULL;
13924                   rgnp->rgn_id = (*nextidp)++;
13925                   ASSERT(rgnp->rgn_id < maxids);
13926                   ASSERT(rarrp[rgnp->rgn_id] == NULL);
13927                   rarrp[rgnp->rgn_id] = rgnp;
13928           }
13929
13930           ASSERT(rgnp->rgn_sfmmu_head == NULL);
13931           ASSERT(rgnp->rgn_hmeflags == 0);
13932   #ifdef DEBUG
13933           for (i = 0; i < MMU_PAGE_SIZES; i++) {
13934                   ASSERT(rgnp->rgn_ttecnt[i] == 0);
13935           }
13936   #endif
13937           rgnp->rgn_saddr = r_saddr;
13938           rgnp->rgn_size = r_size;
13939           rgnp->rgn_obj = r_obj;
13940           rgnp->rgn_objoff = r_objoff;
13941           rgnp->rgn_perm = r_perm;
13942           rgnp->rgn_pgszc = r_pgszc;
13943           rgnp->rgn_flags = r_type;
13944           rgnp->rgn_refcnt = 0;
13945           rgnp->rgn_cb_function = r_cb_function;
13946           rgnp->rgn_hash = srdp->srd_rgnhash[rhash];
13947           srdp->srd_rgnhash[rhash] = rgnp;
13948           (*busyrgnsp)++;
13949           ASSERT(*busyrgnsp <= maxids);
13950           goto rfound;
13951
13952   fail:
13953           ASSERT(new_rgnp != NULL);
13954           kmem_cache_free(region_cache, new_rgnp);
13955           return (HAT_INVALID_REGION_COOKIE);
13956   }
13957
13958   /*
13959    * This function implements the shared context functionality required
13960    * when detaching a segment from an address space. It must be called
13961    * from hat_unshare() for all D(ISM) segments and from segvn_unmap(),
13962    * for segments with a valid region_cookie.
13963    * It will also be called from all seg_vn routines which change a
13964    * segment's attributes such as segvn_setprot(), segvn_setpagesize(),
13965    * segvn_clrszc() & segvn_advise(), as well as in the case of COW fault
13966    * from segvn_fault().
13967    */
13968   void
13969   hat_leave_region(struct hat *sfmmup, hat_region_cookie_t rcookie, uint_t flags)
13970   {
13971           sf_srd_t *srdp = sfmmup->sfmmu_srdp;
13972           sf_scd_t *scdp;
13973           uint_t rhash;
13974           uint_t rid = (uint_t)((uint64_t)rcookie);
13975           hatlock_t *hatlockp = NULL;
13976           sf_region_t *rgnp;
13977           sf_region_t **prev_rgnpp;
13978           sf_region_t *cur_rgnp;
13979           void *r_obj;
13980           int i;
13981           caddr_t r_saddr;
13982           caddr_t r_eaddr;
13983           size_t  r_size;
13984           uchar_t r_pgszc;
13985           uchar_t r_type = flags & HAT_REGION_TYPE_MASK;
```

```
13987         ASSERT(sfmmup != ksfmmup);
13988         ASSERT(srdp != NULL);
13989         ASSERT(srdp->srd_refcnt > 0);
13990         ASSERT(!(flags & ~HAT_REGION_TYPE_MASK));
13991         ASSERT(flags == HAT_REGION_TEXT || flags == HAT_REGION_ISM);
13992         ASSERT(!sfmmup->sfmmu_free || sfmmup->sfmmu_scdp == NULL);

13994         r_type = (r_type == HAT_REGION_ISM) ? SFMMU_REGION_ISM :
13995             SFMMU_REGION_HME;

13997         if (r_type == SFMMU_REGION_ISM) {
13998                 ASSERT(SFMMU_IS_ISMRID_VALID(rid));
13999                 ASSERT(rid < SFMMU_MAX_ISM_REGIONS);
14000                 rgnp = srdp->srd_ismrgnp[rid];
14001         } else {
14002                 ASSERT(SFMMU_IS_SHMERID_VALID(rid));
14003                 ASSERT(rid < SFMMU_MAX_HME_REGIONS);
14004                 rgnp = srdp->srd_hmergnp[rid];
14005         }
14006         ASSERT(rgnp != NULL);
14007         ASSERT(rgnp->rgn_id == rid);
14008         ASSERT((rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK) == r_type);
14009         ASSERT(!(rgnp->rgn_flags & SFMMU_REGION_FREE));
14010         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         }

14012         if (sfmmup->sfmmu_free) {
14013                 ulong_t rttecnt;
14014                 r_pgszc = rgnp->rgn_pgszc;
14015                 r_size = rgnp->rgn_size;

14017                 ASSERT(sfmmup->sfmmu_scdp == NULL);
14018                 if (r_type == SFMMU_REGION_ISM) {
14019                         SF_RGNMAP_DEL(sfmmup->sfmmu_ismregion_map, rid);
14020                 } else {
14021                         /* update shme rgns ttecnt in sfmmu_ttecnt */
14022                         rttecnt = r_size >> TTE_PAGE_SHIFT(r_pgszc);
14023                         ASSERT(sfmmup->sfmmu_ttecnt[r_pgszc] >= rttecnt);

14025                         atomic_add_long(&sfmmup->sfmmu_ttecnt[r_pgszc],
14026                             -rttecnt);

14028                         SF_RGNMAP_DEL(sfmmup->sfmmu_hmeregion_map, rid);
14029                 }
14030         } else if (r_type == SFMMU_REGION_ISM) {
14031                 hatlockp = sfmmu_hat_enter(sfmmup);
14032                 ASSERT(rid < srdp->srd_next_ismrid);
14033                 SF_RGNMAP_DEL(sfmmup->sfmmu_ismregion_map, rid);
14034                 scdp = sfmmup->sfmmu_scdp;
14035                 if (scdp != NULL &&
14036                     SF_RGNMAP_TEST(scdp->scd_ismregion_map, rid)) {
14037                         sfmmu_leave_scd(sfmmup, r_type);
14038                         ASSERT(sfmmu_hat_lock_held(sfmmup));
14039                 }
14040                 sfmmu_hat_exit(hatlockp);
14041         } else {
14042                 ulong_t rttecnt;
14043                 r_pgszc = rgnp->rgn_pgszc;
14044                 r_saddr = rgnp->rgn_saddr;
14045                 r_size = rgnp->rgn_size;
```



```
14046                 r_eaddr = r_saddr + r_size;

14048                 ASSERT(r_type == SFMMU_REGION_HME);
14049                 hatlockp = sfmmu_hat_enter(sfmmup);
14050                 ASSERT(rid < srdp->srd_next_hmerid);
14051                 SF_RGNMAP_DEL(sfmmup->sfmmu_hmeregion_map, rid);

14053                 /*
14054                  * If region is part of an SCD call sfmmu_leave_scd().
14055                  * Otherwise if process is not exiting and has valid context
14056                  * just drop the context on the floor to lose stale TLB
14057                  * entries and force the update of tsb miss area to reflect
14058                  * the new region map. After that clean our TSB entries.
14059                  */
14060                 scdp = sfmmup->sfmmu_scdp;
14061                 if (scdp != NULL &&
14062                     SF_RGNMAP_TEST(scdp->scd_hmeregion_map, rid)) {
14063                         sfmmu_leave_scd(sfmmup, r_type);
14064                         ASSERT(sfmmu_hat_lock_held(sfmmup));
14065                 }
14066                 sfmmu_invalidate_ctx(sfmmup);

14068                 i = TTE8K;
14069                 while (i < mmu_page_sizes) {
14070                         if (rgnp->rgn_ttecnt[i] != 0) {
14071                                 sfmmu_unload_tsb_range(sfmmup, r_saddr,
14072                                     r_eaddr, i);
14073                                 if (i < TTE4M) {
14074                                         i = TTE4M;
14075                                         continue;
14076                                 } else {
14077                                         break;
14078                                 }
14079                         }
14080                         i++;
14081                 }
14082                 /* Remove the preallocated 1/4 8k ttecnt for 4M regions. */
14083                 if (r_pgszc >= TTE4M) {
14084                         rttecnt = r_size >> (TTE_PAGE_SHIFT(TTE8K) + 2);
14085                         ASSERT(sfmmup->sfmmu_tsb0_4minflcnt >=
14086                             rttecnt);
14087                         sfmmup->sfmmu_tsb0_4minflcnt -= rttecnt;
14088                 }

14090                 /* update shme rgns ttecnt in sfmmu_ttecnt */
14091                 rttecnt = r_size >> TTE_PAGE_SHIFT(r_pgszc);
14092                 ASSERT(sfmmup->sfmmu_ttecnt[r_pgszc] >= rttecnt);
14093                 atomic_add_long(&sfmmup->sfmmu_ttecnt[r_pgszc], -rttecnt);

14095                 sfmmu_hat_exit(hatlockp);
14096                 if (scdp != NULL && sfmmup->sfmmu_scdp == NULL) {
14097                         /* sfmmup left the scd, grow private tsb */
14098                         sfmmu_check_page_sizes(sfmmup, 1);
14099                 } else {
14100                         sfmmu_check_page_sizes(sfmmup, 0);
14101                 }
14102         }

14104         if (r_type == SFMMU_REGION_HME) {
14105                 sfmmu_unlink_from_hmeregion(sfmmup, rgnp);
14106         }

14108         r_obj = rgnp->rgn_obj;
14109         if (atomic_dec_32_nv((volatile uint_t *)&rgnp->rgn_refcnt)) {
14110                 return;
14111         }
```

```
14113         /*
14114          * looks like nobody uses this region anymore. Free it.
14115          */
14116         rhash = RGN_HASH_FUNCTION(r_obj);
14117         mutex_enter(&srdp->srd_mutex);
14118         for (prev_rgnpp = &srdp->srd_rgnhash[rhash];
14119             (cur_rgnp = *prev_rgnpp) != NULL;
14120             prev_rgnpp = &cur_rgnp->rgn_hash) {
14121                 if (cur_rgnp == rgnp && cur_rgnp->rgn_refcnt == 0) {
14122                         break;
14123                 }
14124         }

14126         if (cur_rgnp == NULL) {
14127                 mutex_exit(&srdp->srd_mutex);
14128                 return;
14129         }

14131         ASSERT((rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK) == r_type);
14132         *prev_rgnpp = rgnp->rgn_hash;
14133         if (r_type == SFMMU_REGION_ISM) {
14134                 rgnp->rgn_flags |= SFMMU_REGION_FREE;
14135                 ASSERT(rid < srdp->srd_next_ismrid);
14136                 rgnp->rgn_next = srdp->srd_ismrgnfree;
14137                 srdp->srd_ismrgnfree = rgnp;
14138                 ASSERT(srdp->srd_ismbusyrgns > 0);
14139                 srdp->srd_ismbusyrgns--;
14140                 mutex_exit(&srdp->srd_mutex);
14141                 return;
14142         }
14143         mutex_exit(&srdp->srd_mutex);

14145         /*
14146          * Destroy region's hmeblks.
14147          */
14148         sfmmu_unload_hmeregion(srdp, rgnp);

14150         rgnp->rgn_hmeflags = 0;

14152         ASSERT(rgnp->rgn_sfmmu_head == NULL);
14153         ASSERT(rgnp->rgn_id == rid);
14154         for (i = 0; i < MMU_PAGE_SIZES; i++) {
14155                 rgnp->rgn_ttecnt[i] = 0;
14156         }
14157         rgnp->rgn_flags |= SFMMU_REGION_FREE;
14158         mutex_enter(&srdp->srd_mutex);
14159         ASSERT(rid < srdp->srd_next_hmerid);
14160         rgnp->rgn_next = srdp->srd_hmergnfree;
14161         srdp->srd_hmergnfree = rgnp;
14162         ASSERT(srdp->srd_hmebusyrgns > 0);
14163         srdp->srd_hmebusyrgns--;
14164         mutex_exit(&srdp->srd_mutex);
14165 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   86609 Fri May  8 18:10:41 2015
new/usr/src/uts/sfmmu/vm/hat_sfmmu.h
remove xhat
The xhat infrastructure was added to support hardware such as the zulu
graphics card - hardware which had on-board MMUs.  The VM used the xhat code
to keep the CPU's and Zulu's page tables in-sync.  Since the only xhat user
was zulu (which is gone), we can safely remove it simplifying the whole VM
subsystem.
Assorted notes:
- AS_BUSY flag was used solely by xhat
**********************************************************
______unchanged_portion_omitted_


 644 /*
 645  * The platform dependent hat structure.
 646  * tte counts should be protected by cas.
 647  * cpuset is protected by cas.
 648  *
 649  * ttecnt accounting for mappings which do not use shared hme is carried out
 650  * during pagefault handling. In the shared hme case, only the first process
 651  * to access a mapping generates a pagefault, subsequent processes simply
 652  * find the shared hme entry during trap handling and therefore there is no
 653  * corresponding event to initiate ttecnt accounting. Currently, as shared
 654  * hmes are only used for text segments, when joining a region we assume the
 655  * worst case and add the the number of ttes required to map the entire region
 656  * to the ttecnt corresponding to the region pagesize. However, if the region
 657  * has a 4M pagesize, and memory is low, the allocation of 4M pages may fail
 658  * then 8K pages will be allocated instead and the first TSB which stores 8K
 659  * mappings will potentially be undersized. To compensate for the potential
 660  * underaccounting in this case we always add 1/4 of the region size to the 8K
 661  * ttecnt.
 662  *
 663  * Note that sfmmu_xhat_provider MUST be the first element.
 662  */

 664 struct hat {
 667         void            *sfmmu_xhat_provider;   /* NULL for CPU hat */
 665         cpuset_t        sfmmu_cpusran;  /* cpu bit mask for efficient xcalls */
 666         struct  as      *sfmmu_as;      /* as this hat provides mapping for */
 667         /* per pgsz private ttecnt + shme rgns ttecnt for rgns not in SCD */
 668         ulong_t         sfmmu_ttecnt[MMU_PAGE_SIZES];
 669         /* shme rgns ttecnt for rgns in SCD */
 670         ulong_t         sfmmu_scdrttecnt[MMU_PAGE_SIZES];
 671         /* est. ism ttes that are NOT in a SCD */
 672         ulong_t         sfmmu_ismttecnt[MMU_PAGE_SIZES];
 673         /* ttecnt for isms that are in a SCD */
 674         ulong_t         sfmmu_scdismttecnt[MMU_PAGE_SIZES];
 675         /* inflate tsb0 to allow for large page alloc failure in region */
 676         ulong_t         sfmmu_tsb0_4minflcnt;
 677         union _h_un {
 678                 ism_blk_t       *sfmmu_iblkp;  /* maps to ismhat(s) */
 679                 ism_ment_t      *sfmmu_imentp; /* ism hat's mapping list */
 680         } h_un;
 681         uint_t          sfmmu_free:1;   /* hat to be freed - set on as_free */
 682         uint_t          sfmmu_ismhat:1; /* hat is dummy ism hatid */
 683         uint_t          sfmmu_scdhat:1; /* hat is dummy scd hatid */
 684         uchar_t         sfmmu_rmstat;   /* refmod stats refcnt */
 685         ushort_t        sfmmu_clrstart; /* start color bin for page coloring */
 686         ushort_t        sfmmu_clrbin;   /* per as phys page coloring bin */
 687         ushort_t        sfmmu_flags;    /* flags */
 688         uchar_t         sfmmu_tteflags; /* pgsz flags */
 689         uchar_t         sfmmu_rtteflags; /* pgsz flags for SRD hmes */
 690         struct tsb_info *sfmmu_tsb;     /* list of per as tsbs */
 691         uint64_t        sfmmu_ismblkpa; /* pa of sfmmu_iblkp, or -1 */
```



```
 692         lock_t          sfmmu_ctx_lock; /* sync ctx alloc and invalidation */
 693         kcondvar_t      sfmmu_tsb_cv;   /* signals TSB swapin or relocation */
 694         uchar_t         sfmmu_cext;     /* context page size encoding */
 695         uint8_t         sfmmu_pgsz[MMU_PAGE_SIZES];  /* ranking for MMU */
 696         sf_srd_t        *sfmmu_srdp;
 697         sf_scd_t        *sfmmu_scdp;    /* scd this address space belongs to */
 698         sf_region_map_t sfmmu_region_map;
 699         sf_rgn_link_t   *sfmmu_hmeregion_links[SFMMU_L1_HMERLINKS];
 700         sf_rgn_link_t   sfmmu_scd_link; /* link to scd or pending queue */
 701 #ifdef sun4v
 702         struct hv_tsb_block sfmmu_hvblock;
 703 #endif
 704         /*
 705          * sfmmu_ctxs is a variable length array of max_mmu_ctxdoms # of
 706          * elements. max_mmu_ctxdoms is determined at run-time.
 707          * sfmmu_ctxs[1] is just the fist element of an array, it always
 708          * has to be the last field to ensure that the memory allocated
 709          * for sfmmu_ctxs is consecutive with the memory of the rest of
 710          * the hat data structure.
 711          */
 712         sfmmu_ctx_t     sfmmu_ctxs[1];

 714 };
______unchanged_portion_omitted_

1218 #endif  /* HBLK_TRACE */


1221 /*
1222  * Hment block structure.
1223  * The hme_blk is the node data structure which the hash structure
1224  * mantains. An hme_blk can have 2 different sizes depending on the
1225  * number of hments it implicitly contains.  When dealing with 64K, 512K,
1226  * or 4M hments there is one hment per hme_blk.  When dealing with
1227  * 8k hments we allocate an hme_blk plus an additional 7 hments to
1228  * give us a total of 8 (NHMENTS) hments that can be referenced through a
1229  * hme_blk.
1230  *
1231  * The hmeblk structure contains 2 tte reference counters used to determine if
1232  * it is ok to free up the hmeblk.  Both counters have to be zero in order
1233  * to be able to free up hmeblk.  They are protected by cas.
1234  * hblk_hmecnt is the number of hments present on pp mapping lists.
1235  * hblk_vcnt reflects number of valid ttes in hmeblk.
1236  *
1237  * The hmeblk now also has per tte lock cnts.  This is required because
1238  * the counts can be high and there are not enough bits in the tte. When
1239  * physio is fixed to not lock the translations we should be able to move
1240  * the lock cnt back to the tte.  See bug id 1198554.
1244  *
1245  * Note that xhat_hme_blk's layout follows this structure: hme_blk_misc
1246  * and sf_hment are at the same offsets in both structures. Whenever
1247  * hme_blk is changed, xhat_hme_blk may need to be updated as well.
1241  */

1243 struct hme_blk_misc {
1244         uint_t  notused:26;
1251         uint_t  notused:25;
1245         uint_t  shared_bit:1;   /* set for SRD shared hmeblk */
1253         uint_t  xhat_bit:1;     /* set for an xhat hme_blk */
1246         uint_t  shadow_bit:1;   /* set for a shadow hme_blk */
1247         uint_t  nucleus_bit:1;  /* set for a nucleus hme_blk */
1248         uint_t  ttesize:3;      /* contains ttesz of hmeblk */
1249 };
______unchanged_portion_omitted_

1284 #define hblk_shared     hblk_misc.shared_bit
```

```
#define hblk_xhat_bit   hblk_misc.xhat_bit
#define hblk_shw_bit    hblk_misc.shadow_bit
#define hblk_nuc_bit    hblk_misc.nucleus_bit
#define hblk_ttesz      hblk_misc.ttesize
#define hblk_hmecnt     hblk_un.hblk_counts.hblk_hmecount
#define hblk_vcnt       hblk_un.hblk_counts.hblk_validcnt
#define hblk_shw_mask   hblk_un.hblk_shadow_mask

#define MAX_HBLK_LCKCNT 0xFFFFFFFF
#define HMEBLK_ALIGN    0x8             /* hmeblk has to be double aligned */

#ifdef  HBLK_TRACE

#define HBLK_STACK_TRACE(hmeblkp, lock)                                 \
{                                                                       \
        int flag = lock;        /* to pacify lint */                    \
        int audit_index;                                                \
                                                                        \
        mutex_enter(&hmeblkp->hblk_audit_lock);                         \
        audit_index = hmeblkp->hblk_audit_index;                        \
        hmeblkp->hblk_audit_index = ((hmeblkp->hblk_audit_index + 1) &  \
            (HBLK_AUDIT_CACHE_SIZE - 1));                               \
        mutex_exit(&hmeblkp->hblk_audit_lock);                          \
                                                                        \
        if (flag)                                                       \
                hmeblkp->hblk_audit_cache[audit_index].flag =           \
                    HBLK_LOCK_PATTERN;                                  \
        else                                                            \
                hmeblkp->hblk_audit_cache[audit_index].flag =           \
                    HBLK_UNLOCK_PATTERN;                                \
                                                                        \
        hmeblkp->hblk_audit_cache[audit_index].thread = curthread;      \
        hmeblkp->hblk_audit_cache[audit_index].depth =                  \
            getpcstack(hmeblkp->hblk_audit_cache[audit_index].stack,    \
            HBLK_STACK_DEPTH);                                          \
}
```

unchanged_portion_omitted_

```
extern size_t   tsb_slab_size;
extern uint_t   tsb_slab_shift;
extern size_t   tsb_slab_mask;

#endif /* !_ASM */

/*
 * Flags for TL kpm tsbmiss handler
 */
#define KPMTSBM_ENABLE_FLAG     0x01    /* bit copy of kpm_enable */
#define KPMTSBM_TLTSBM_FLAG     0x02    /* use TL tsbmiss handler */
#define KPMTSBM_TSBPHYS_FLAG    0x04    /* use ASI_MEM for TSB update */

/*
 * The TSB
 * All TSB sizes supported by the hardware are now supported (8K - 1M).
 * For kernel TSBs we may go beyond the hardware supported sizes and support
 * larger TSBs via software.
 * All TTE sizes are supported in the TSB; the manner in which this is
 * done is cpu dependent.
 */
#define TSB_MIN_SZCODE          TSB_8K_SZCODE   /* min. supported TSB size */
#define TSB_MIN_OFFSET_MASK     (TSB_OFFSET_MASK(TSB_MIN_SZCODE))

#ifdef sun4v
#define UTSB_MAX_SZCODE         TSB_256M_SZCODE /* max. supported TSB size */
#else /* sun4u */
#define UTSB_MAX_SZCODE         TSB_1M_SZCODE   /* max. supported TSB size */
```



```
#endif /* sun4v */

#define UTSB_MAX_OFFSET_MASK    (TSB_OFFSET_MASK(UTSB_MAX_SZCODE))

#define TSB_FREEMEM_MIN         0x1000          /* 32 mb */
#define TSB_FREEMEM_LARGE       0x10000         /* 512 mb */
#define TSB_8K_SZCODE           0               /* 512 entries */
#define TSB_16K_SZCODE          1               /* 1k entries */
#define TSB_32K_SZCODE          2               /* 2k entries */
#define TSB_64K_SZCODE          3               /* 4k entries */
#define TSB_128K_SZCODE         4               /* 8k entries */
#define TSB_256K_SZCODE         5               /* 16k entries */
#define TSB_512K_SZCODE         6               /* 32k entries */
#define TSB_1M_SZCODE           7               /* 64k entries */
#define TSB_2M_SZCODE           8               /* 128k entries */
#define TSB_4M_SZCODE           9               /* 256k entries */
#define TSB_8M_SZCODE           10              /* 512k entries */
#define TSB_16M_SZCODE          11              /* 1M entries */
#define TSB_32M_SZCODE          12              /* 2M entries */
#define TSB_64M_SZCODE          13              /* 4M entries */
#define TSB_128M_SZCODE         14              /* 8M entries */
#define TSB_256M_SZCODE         15              /* 16M entries */
#define TSB_ENTRY_SHIFT         4       /* each entry = 128 bits = 16 bytes */
#define TSB_ENTRY_SIZE          (1 << 4)
#define TSB_START_SIZE          9
#define TSB_ENTRIES(tsbsz)      (1 << (TSB_START_SIZE + tsbsz))
#define TSB_BYTES(tsbsz)        (TSB_ENTRIES(tsbsz) << TSB_ENTRY_SHIFT)
#define TSB_OFFSET_MASK(tsbsz)  (TSB_ENTRIES(tsbsz) - 1)
#define TSB_BASEADDR_MASK       ((1 << 12) - 1)

/*
 * sun4u platforms
 * ---------------
 * We now support two user TSBs with one TSB base register.
 * Hence the TSB base register is split up as follows:
 *
 * When only one TSB present:
 *   [63  62..42  41..13  12..4  3..0]
 *     ^   ^       ^       ^     ^
 *     |   |       |       |     |_ TSB size code
 *     |   |       |       |
 *     |   |       |       |_ Reserved 0
 *     |   |       |
 *     |   |       |_ TSB VA[41..13]
 *     |   |
 *     |   |_ VA hole (Spitfire), zeros (Cheetah and beyond)
 *     |
 *     |_ 0
 *
 * When second TSB present:
 *   [63  62..42  41..33  32..29  28..22  21..13  12..4  3..0]
 *     ^   ^       ^       ^       ^       ^       ^     ^
 *     |   |       |       |       |       |       |     |_ First TSB size code
 *     |   |       |       |       |       |       |
 *     |   |       |       |       |       |       |_ Reserved 0
 *     |   |       |       |       |       |
 *     |   |       |       |       |       |_ First TSB's VA[21..13]
 *     |   |       |       |       |
 *     |   |       |       |       |_ Reserved for future use
 *     |   |       |       |
 *     |   |       |       |_ Second TSB's size code
 *     |   |       |
 *     |   |       |_ Second TSB's VA[21..13]
 *     |   |
```

```
 *     |   |_ VA hole (Spitfire) / ones (Cheetah and beyond)
 *     |
 *     |_ 1
 *
 * Note that since we store 21..13 of each TSB's VA, TSBs and their slabs
 * may be up to 4M in size.  For now, only hardware supported TSB sizes
 * are supported, though the slabs are usually 4M in size.
 *
 * sun4u platforms that define UTSB_PHYS use physical addressing to access
 * the user TSBs at TL>0.  The first user TSB base is in the MMU I/D TSB Base
 * registers.  The second TSB base uses a dedicated scratchpad register which
 * requires a definition of SCRATCHPAD_UTSBREG2 in mach_sfmmu.h.  The layout for
 * both registers is equivalent to sun4v below, except the TSB PA range is
 * [46..13] for sun4u.
 *
 * sun4v platforms
 * ---------------
 * On sun4v platforms, we use two dedicated scratchpad registers as pseudo
 * hardware TSB base registers to hold up to two different user TSBs.
 *
 * Each register contains TSB's physical base and size code information
 * as follows:
 *
 *   [63..56  55..13  12..4  3..0]
 *      ^       ^       ^     ^
 *      |       |       |     |
 *      |       |       |     |_ TSB size code
 *      |       |       |
 *      |       |       |_ Reserved 0
 *      |       |
 *      |       |_ TSB PA[55..13]
 *      |
 *      |
 *      |
 *      |_ 0 for valid TSB
 *
 * Absence of a user TSB (primarily the second user TSB) is indicated by
 * storing a negative value in the TSB base register. This allows us to
 * check for presence of a user TSB by simply checking bit# 63.
 */
#define TSBREG_MSB_SHIFT        32              /* set upper bits */
#define TSBREG_MSB_CONST        0xfffff800      /* set bits 63..43 */
#define TSBREG_FIRTSB_SHIFT     42              /* to clear bits 63:22 */
#define TSBREG_SECTSB_MKSHIFT   20              /* 21:13 --> 41:33 */
#define TSBREG_SECTSB_LSHIFT    22              /* to clear bits 63:42 */
#define TSBREG_SECTSB_RSHIFT    (TSBREG_SECTSB_MKSHIFT + TSBREG_SECTSB_LSHIFT)
                                                /* sectsb va -> bits 21:13 */
                                                /* after clearing upper bits */
#define TSBREG_SECSZ_SHIFT      29              /* to get sectsb szc to 3:0 */
#define TSBREG_VAMASK_SHIFT     13              /* set up VA mask */

#define BIGKTSB_SZ_MASK         0xf
#define TSB_SOFTSZ_MASK         BIGKTSB_SZ_MASK
#define MIN_BIGKTSB_SZCODE      9       /* 256k entries */
#define MAX_BIGKTSB_SZCODE      11      /* 1024k entries */
#define MAX_BIGKTSB_TTES        (TSB_BYTES(MAX_BIGKTSB_SZCODE) / MMU_PAGESIZE4M)

#define TAG_VALO_SHIFT          22              /* tag's va are bits 63-22 */
/*
 * sw bits used on tsb_tag - bit masks used only in assembly
 * use only a sethi for these fields.
 */
#define TSBTAG_INVALID  0x00008000              /* tsb_tag.tag_invalid */
#define TSBTAG_LOCKED   0x00004000              /* tsb_tag.tag_locked */

#ifdef  _ASM
```



```
/*
 * Marker to indicate that this instruction will be hot patched at runtime
 * to some other value.
 * This value must be zero since it fills in the imm bits of the target
 * instructions to be patched
 */
#define RUNTIME_PATCH   (0)

/*
 * V9 defines nop instruction as the following, which we use
 * at runtime to nullify some instructions we don't want to
 * execute in the trap handlers on certain platforms.
 */
#define MAKE_NOP_INSTR(reg)     \
        sethi   %hi(0x1000000), reg

/*
 * This macro constructs a SPARC V9 "jmpl <source reg>, %g0"
 * instruction, with the source register specified by the jump_reg_number.
 * The jmp opcode [24:19] = 11 1000 and source register is bits [18:14].
 * The instruction is returned in reg. The macro is used to patch in a jmpl
 * instruction at runtime.
 */
#define MAKE_JMP_INSTR(jump_reg_number, reg, tmp)       \
        sethi   %hi(0x81c00000), reg;                   \
        mov     jump_reg_number, tmp;                   \
        sll     tmp, 14, tmp;                           \
        or      reg, tmp, reg

/*
 * Macro to get hat per-MMU cnum on this CPU.
 * sfmmu - In, pass in "sfmmup" from the caller.
 * cnum - Out, return 'cnum' to the caller
 * scr  - scratch
 */
#define SFMMU_CPU_CNUM(sfmmu, cnum, scr)                                        \
        CPU_ADDR(scr, cnum);    /* scr = load CPU struct addr */                \
        ld      [scr + CPU_MMU_IDX], cnum;      /* cnum = mmuid */              \
        add     sfmmu, SFMMU_CTXS, scr; /* scr = sfmmup->sfmmu_ctxs[] */        \
        sllx    cnum, SFMMU_MMU_CTX_SHIFT, cnum;                                \
        add     scr, cnum, scr;         /* scr = sfmmup->sfmmu_ctxs[id] */      \
        ldx     [scr + SFMMU_MMU_GC_NUM], scr;  /* sfmmu_ctxs[id].gcnum */      \
        sllx    scr, SFMMU_MMU_CNUM_LSHIFT, scr;                                \
        srlx    scr, SFMMU_MMU_CNUM_LSHIFT, cnum;       /* cnum = sfmmu cnum */

/*
 * Macro to get hat gnum & cnum assocaited with sfmmu_ctx[mmuid] entry
 * entry - In,  pass in (&sfmmu_ctxs[mmuid] - SFMMU_CTXS) from the caller.
 * gnum - Out, return sfmmu gnum
 * cnum - Out, return sfmmu cnum
 * reg  - scratch
 */
#define SFMMU_MMUID_GNUM_CNUM(entry, gnum, cnum, reg)                           \
        ldx     [entry + SFMMU_CTXS], reg;  /* reg = sfmmu (gnum | cnum) */     \
        srlx    reg, SFMMU_MMU_GNUM_RSHIFT, gnum;    /* gnum = sfmmu gnum */    \
        sllx    reg, SFMMU_MMU_CNUM_LSHIFT, cnum;                               \
        srlx    cnum, SFMMU_MMU_CNUM_LSHIFT, cnum;   /* cnum = sfmmu cnum */

/*
 * Macro to get this CPU's tsbmiss area.
 */
#define CPU_TSBMISS_AREA(tsbmiss, tmp1)                                         \
        CPU_INDEX(tmp1, tsbmiss);               /* tmp1 = cpu idx */        \
        sethi   %hi(tsbmiss_area), tsbmiss;     /* tsbmiss base ptr */      \
        mulx    tmp1, TSBMISS_SIZE, tmp1;       /* byte offset */           \
```

```
2052         or      tsbmiss, %lo(tsbmiss_area), tsbmiss;                    \
2053         add     tsbmiss, tmp1, tsbmiss          /* tsbmiss area of CPU */


2056 /*
2057  * Macro to set kernel context + page size codes in DMMU primary context
2058  * register. It is only necessary for sun4u because sun4v does not need
2059  * page size codes
2060  */
2061 #ifdef sun4v

2063 #define SET_KCONTEXTREG(reg0, reg1, reg2, reg3, reg4, label1, label2, label3)

2065 #else

2067 #define SET_KCONTEXTREG(reg0, reg1, reg2, reg3, reg4, label1, label2, label3) \
2068         sethi   %hi(kcontextreg), reg0;                                 \
2069         ldx     [reg0 + %lo(kcontextreg)], reg0;                        \
2070         mov     MMU_PCONTEXT, reg1;                                     \
2071         ldxa    [reg1]ASI_MMU_CTX, reg2;                                \
2072         xor     reg0, reg2, reg2;                                       \
2073         brz     reg2, label3;                                           \
2074         srlx    reg2, CTXREG_NEXT_SHIFT, reg2;                          \
2075         rdpr    %pstate, reg3;          /* disable interrupts */        \
2076         btst    PSTATE_IE, reg3;                                        \
2077 /*CSTYLED*/                                                             \
2078         bnz,a,pt %icc, label1;                                          \
2079         wrpr    reg3, PSTATE_IE, %pstate;                               \
2080 /*CSTYLED*/                                                             \
2081 label1:;                                                                \
2082         brz     reg2, label2;       /* need demap if N_pgsz0/1 change */ \
2083         sethi   %hi(FLUSH_ADDR), reg4;                                  \
2084         mov     DEMAP_ALL_TYPE, reg2;                                   \
2085         stxa    %g0, [reg2]ASI_DTLB_DEMAP;                              \
2086         stxa    %g0, [reg2]ASI_ITLB_DEMAP;                              \
2087 /*CSTYLED*/                                                             \
2088 label2:;                                                                \
2089         stxa    reg0, [reg1]ASI_MMU_CTX;                                \
2090         flush   reg4;                                                   \
2091         btst    PSTATE_IE, reg3;                                        \
2092 /*CSTYLED*/                                                             \
2093         bnz,a,pt %icc, label3;                                          \
2094         wrpr    %g0, reg3, %pstate;     /* restore interrupt state */   \
2095 label3:;

2097 #endif

2099 /*
2100  * Macro to setup arguments with kernel sfmmup context + page size before
2101  * calling sfmmu_setctx_sec()
2102  */
2103 #ifdef sun4v
2104 #define SET_KAS_CTXSEC_ARGS(sfmmup, arg0, arg1)                 \
2105         set     KCONTEXT, arg0;                                 \
2106         set     0, arg1;
2107 #else
2108 #define SET_KAS_CTXSEC_ARGS(sfmmup, arg0, arg1)                 \
2109         ldub    [sfmmup + SFMMU_CEXT], arg1;                    \
2110         set     KCONTEXT, arg0;                                 \
2111         sll     arg1, CTXREG_EXT_SHIFT, arg1;
2112 #endif

2114 #define PANIC_IF_INTR_DISABLED_PSTR(pstatereg, label, scr)              \
2115         andcc   pstatereg, PSTATE_IE, %g0;      /* panic if intrs */    \
2116 /*CSTYLED*/                                                             \
2117         bnz,pt  %icc, label;                    /* already disabled */  \
```



```
2118         nop;                                                            \
2119                                                                         \
2120         sethi   %hi(panicstr), scr;                                     \
2121         ldx     [scr + %lo(panicstr)], scr;                             \
2122         tst     scr;                                                    \
2123 /*CSTYLED*/                                                             \
2124         bnz,pt  %xcc, label;                                            \
2125         nop;                                                            \
2126                                                                         \
2127         save    %sp, -SA(MINFRAME), %sp;                                \
2128         sethi   %hi(sfmmu_panic1), %o0;                                 \
2129         call    panic;                                                  \
2130         or      %o0, %lo(sfmmu_panic1), %o0;                            \
2131 /*CSTYLED*/                                                             \
2132 label:

2134 #define PANIC_IF_INTR_ENABLED_PSTR(label, scr)                          \
2135         /*                                                              \
2136          * The caller must have disabled interrupts.                    \
2137          * If interrupts are not disabled, panic                        \
2138          */                                                             \
2139         rdpr    %pstate, scr;                                           \
2140         andcc   scr, PSTATE_IE, %g0;                                    \
2141 /*CSTYLED*/                                                             \
2142         bz,pt   %icc, label;                                            \
2143         nop;                                                            \
2144                                                                         \
2145         sethi   %hi(panicstr), scr;                                     \
2146         ldx     [scr + %lo(panicstr)], scr;                             \
2147         tst     scr;                                                    \
2148 /*CSTYLED*/                                                             \
2149         bnz,pt  %xcc, label;                                            \
2150         nop;                                                            \
2151                                                                         \
2152         sethi   %hi(sfmmu_panic6), %o0;                                 \
2153         call    panic;                                                  \
2154         or      %o0, %lo(sfmmu_panic6), %o0;                            \
2155 /*CSTYLED*/                                                             \
2156 label:

2158 #endif  /* _ASM */

2160 #ifndef _ASM

2162 #ifdef VAC
2163 /*
2164  * Page coloring
2165  * The p_vcolor field of the page struct (1 byte) is used to store the
2166  * virtual page color.  This provides for 255 colors.  The value zero is
2167  * used to mean the page has no color - never been mapped or somehow
2168  * purified.
2169  */

2171 #define PP_GET_VCOLOR(pp)       (((pp)->p_vcolor) - 1)
2172 #define PP_NEWPAGE(pp)          (!(pp)->p_vcolor)
2173 #define PP_SET_VCOLOR(pp, color)                                        \
2174         ((pp)->p_vcolor = ((color) + 1))

2176 /*
2177  * As mentioned p_vcolor == 0 means there is no color for this page.
2178  * But PP_SET_VCOLOR(pp, color) expects 'color' to be real color minus
2179  * one so we define this constant.
2180  */
2181 #define NO_VCOLOR       (-1)

2183 #define addr_to_vcolor(addr) \
```

```
2184         (((uint_t)(uintptr_t)(addr) >> MMU_PAGESHIFT) & vac_colors_mask)
2185 #else   /* VAC */
2186 #define addr_to_vcolor(addr)    (0)
2187 #endif  /* VAC */

2189 /*
2190  * The field p_index in the psm page structure is for large pages support.
2191  * P_index is a bit-vector of the different mapping sizes that a given page
2192  * is part of. An hme structure for a large mapping is only added in the
2193  * group leader page (first page). All pages covered by a given large mapping
2194  * have the corrosponding mapping bit set in their p_index field. This allows
2195  * us to only store an explicit hme structure in the leading page which
2196  * simplifies the mapping link list management. Furthermore, it provides us
2197  * a fast mechanism for determining the largest mapping a page is part of. For
2198  * exmaple, a page with a 64K and a 4M mappings has a p_index value of 0x0A.
2199  *
2200  * Implementation note: even though the first bit in p_index is reserved
2201  * for 8K mappings, it is NOT USED by the code and SHOULD NOT be set.
2202  * In addition, the upper four bits of the p_index field are used by the
2203  * code as temporaries
2204  */

2206 /*
2207  * Defines for psm page struct fields and large page support
2208  */
2209 #define SFMMU_INDEX_SHIFT               6
2210 #define SFMMU_INDEX_MASK                ((1 << SFMMU_INDEX_SHIFT) - 1)

2212 /* Return the mapping index */
2213 #define PP_MAPINDEX(pp) ((pp)->p_index & SFMMU_INDEX_MASK)

2215 /*
2216  * These macros rely on the following property:
2217  * All pages constituting a large page are covered by a virtually
2218  * contiguous set of page_t's.
2219  */

2221 /* Return the leader for this mapping size */
2222 #define PP_GROUPLEADER(pp, sz) \
2223         (&(pp)[-(int)(pp->p_pagenum & (TTEPAGES(sz)-1))])

2225 /* Return the root page for this page based on p_szc */
2226 #define PP_PAGEROOT(pp) ((pp)->p_szc == 0 ? (pp) : \
2227         PP_GROUPLEADER((pp), (pp)->p_szc))

2229 #define PP_PAGENEXT_N(pp, n)    ((pp) + (n))
2230 #define PP_PAGENEXT(pp)         PP_PAGENEXT_N((pp), 1)

2232 #define PP_PAGEPREV_N(pp, n)    ((pp) - (n))
2233 #define PP_PAGEPREV(pp)         PP_PAGEPREV_N((pp), 1)

2235 #define PP_ISMAPPED_LARGE(pp)   (PP_MAPINDEX(pp) != 0)

2237 /* Need function to test the page mappping which takes p_index into account */
2238 #define PP_ISMAPPED(pp) ((pp)->p_mapping || PP_ISMAPPED_LARGE(pp))

2240 /*
2241  * Don't call this macro with sz equal to zero. 8K mappings SHOULD NOT
2242  * set p_index field.
2243  */
2244 #define PAGESZ_TO_INDEX(sz)     (1 << (sz))


2247 /*
2248  * prototypes for hat assembly routines.  Some of these are
2249  * known to machine dependent VM code.
```



```
2250  */
2251 extern uint64_t sfmmu_make_tsbtag(caddr_t);
2252 extern struct tsbe *
2253                 sfmmu_get_tsbe(uint64_t, caddr_t, int, int);
2254 extern void     sfmmu_load_tsbe(struct tsbe *, uint64_t, tte_t *, int);
2255 extern void     sfmmu_unload_tsbe(struct tsbe *, uint64_t, int);
2256 extern void     sfmmu_load_mmustate(sfmmu_t *);
2257 extern void     sfmmu_raise_tsb_exception(uint64_t, uint64_t);
2258 #ifndef sun4v
2259 extern void     sfmmu_itlb_ld_kva(caddr_t, tte_t *);
2260 extern void     sfmmu_dtlb_ld_kva(caddr_t, tte_t *);
2261 #endif /* sun4v */
2262 extern void     sfmmu_copytte(tte_t *, tte_t *);
2263 extern int      sfmmu_modifytte(tte_t *, tte_t *, tte_t *);
2264 extern int      sfmmu_modifytte_try(tte_t *, tte_t *, tte_t *);
2265 extern pfn_t    sfmmu_ttetopfn(tte_t *, caddr_t);
2266 extern uint_t   sfmmu_disable_intrs(void);
2267 extern void     sfmmu_enable_intrs(uint_t);
2268 /*
2269  * functions exported to machine dependent VM code
2270  */
2271 extern void     sfmmu_patch_ktsb(void);
2272 #ifndef UTSB_PHYS
2273 extern void     sfmmu_patch_utsb(void);
2274 #endif /* UTSB_PHYS */
2275 extern pfn_t    sfmmu_vatopfn(caddr_t, sfmmu_t *, tte_t *);
2276 extern void     sfmmu_vatopfn_suspended(caddr_t, sfmmu_t *, tte_t *);
2277 extern pfn_t    sfmmu_kvaszc2pfn(caddr_t, int);
2278 #ifdef  DEBUG
2279 extern void     sfmmu_check_kpfn(pfn_t);
2280 #else
2281 #define         sfmmu_check_kpfn(pfn)   /* disabled */
2282 #endif  /* DEBUG */
2283 extern void     sfmmu_memtte(tte_t *, pfn_t, uint_t, int);
2284 extern void     sfmmu_tteload(struct hat *, tte_t *, caddr_t, page_t *, uint_t);
2285 extern void     sfmmu_tsbmiss_exception(struct regs *, uintptr_t, uint_t);
2286 extern void     sfmmu_init_tsbs(void);
2287 extern caddr_t  sfmmu_ktsb_alloc(caddr_t);
2288 extern int      sfmmu_getctx_pri(void);
2289 extern int      sfmmu_getctx_sec(void);
2290 extern void     sfmmu_setctx_sec(uint_t);
2291 extern void     sfmmu_inv_tsb(caddr_t, uint_t);
2292 extern void     sfmmu_init_ktsbinfo(void);
2293 extern int      sfmmu_setup_4lp(void);
2294 extern void     sfmmu_patch_mmu_asi(int);
2295 extern void     sfmmu_init_nucleus_hblks(caddr_t, size_t, int, int);
2296 extern void     sfmmu_cache_flushall(void);
2297 extern pgcnt_t  sfmmu_tte_cnt(sfmmu_t *, uint_t);
2298 extern void     *sfmmu_tsb_segkmem_alloc(vmem_t *, size_t, int);
2299 extern void     sfmmu_tsb_segkmem_free(vmem_t *, void *, size_t);
2300 extern void     sfmmu_reprog_pgsz_arr(sfmmu_t *, uint8_t *);

2302 extern void     hat_kern_setup(void);
2303 extern int      hat_page_relocate(page_t **, page_t **, spgcnt_t *);
2304 extern int      sfmmu_get_ppvcolor(struct page *);
2305 extern int      sfmmu_get_addrvcolor(caddr_t);
2306 extern int      sfmmu_hat_lock_held(sfmmu_t *);
2307 extern int      sfmmu_alloc_ctx(sfmmu_t *, int, struct cpu *, int);

2318 /*
2319  * Functions exported to xhat_sfmmu.c
2320  */
2309 extern kmutex_t *sfmmu_mlist_enter(page_t *);
2310 extern void     sfmmu_mlist_exit(kmutex_t *);
2311 extern int      sfmmu_mlist_held(struct page *);
2312 extern struct hme_blk *sfmmu_hmetohblk(struct sf_hment *);
```

```
2314 /*
2315  * MMU-specific functions optionally imported from the CPU module
2316  */
2317 #pragma weak mmu_init_scd
2318 #pragma weak mmu_large_pages_disabled
2319 #pragma weak mmu_set_ctx_page_sizes
2320 #pragma weak mmu_check_page_sizes

2322 extern void mmu_init_scd(sf_scd_t *);
2323 extern uint_t mmu_large_pages_disabled(uint_t);
2324 extern void mmu_set_ctx_page_sizes(sfmmu_t *);
2325 extern void mmu_check_page_sizes(sfmmu_t *, uint64_t *);

2327 extern sfmmu_t          *ksfmmup;
2328 extern caddr_t          ktsb_base;
2329 extern uint64_t         ktsb_pbase;
2330 extern int              ktsb_sz;
2331 extern int              ktsb_szcode;
2332 extern caddr_t          ktsb4m_base;
2333 extern uint64_t         ktsb4m_pbase;
2334 extern int              ktsb4m_sz;
2335 extern int              ktsb4m_szcode;
2336 extern uint64_t         kpm_tsbbase;
2337 extern int              kpm_tsbsz;
2338 extern int              ktsb_phys;
2339 extern int              enable_bigktsb;
2340 #ifndef sun4v
2341 extern int              utsb_dtlb_ttenum;
2342 extern int              utsb4m_dtlb_ttenum;
2343 #endif /* sun4v */
2344 extern int              uhmehash_num;
2345 extern int              khmehash_num;
2346 extern struct hmehash_bucket *uhme_hash;
2347 extern struct hmehash_bucket *khme_hash;
2348 extern uint_t           hblk_alloc_dynamic;
2349 extern struct tsbmiss   tsbmiss_area[NCPU];
2350 extern struct kpmtsbm   kpmtsbm_area[NCPU];

2352 #ifndef sun4v
2353 extern int              dtlb_resv_ttenum;
2354 extern caddr_t          utsb_vabase;
2355 extern caddr_t          utsb4m_vabase;
2356 #endif /* sun4v */
2357 extern vmem_t           *kmem_tsb_default_arena[];
2358 extern int              tsb_lgrp_affinity;

2360 extern uint_t           disable_large_pages;
2361 extern uint_t           disable_ism_large_pages;
2362 extern uint_t           disable_auto_data_large_pages;
2363 extern uint_t           disable_auto_text_large_pages;

2365 /* kpm externals */
2366 extern pfn_t            sfmmu_kpm_vatopfn(caddr_t);
2367 extern void             sfmmu_kpm_patch_tlbm(void);
2368 extern void             sfmmu_kpm_patch_tsbm(void);
2369 extern void             sfmmu_patch_shctx(void);
2370 extern void             sfmmu_kpm_load_tsb(caddr_t, tte_t *, int);
2371 extern void             sfmmu_kpm_unload_tsb(caddr_t, int);
2372 extern void             sfmmu_kpm_tsbmtl(short *, uint_t *, int);
2373 extern int              sfmmu_kpm_stsbmtl(uchar_t *, uint_t *, int);
2374 extern caddr_t          kpm_vbase;
2375 extern size_t           kpm_size;
2376 extern struct memseg    *memseg_hash[];
2377 extern uint64_t         memseg_phash[];
2378 extern kpm_hlk_t        *kpmp_table;
```



```
2379 extern kpm_shlk_t       *kpmp_stable;
2380 extern uint_t           kpmp_table_sz;
2381 extern uint_t           kpmp_stable_sz;
2382 extern uchar_t          kpmp_shift;

2384 #define PP_ISMAPPED_KPM(pp)     ((pp)->p_kpmref > 0)

2386 #define IS_KPM_ALIAS_RANGE(vaddr)                                       \
2387         (((vaddr) - kpm_vbase) >> (uintptr_t)kpm_size_shift > 0)

2389 #endif /* !_ASM */

2391 /* sfmmu_kpm_tsbmtl flags */
2392 #define KPMTSBM_STOP            0
2393 #define KPMTSBM_START           1

2395 /*
2396  * For kpm_smallpages, the state about how a kpm page is mapped and whether
2397  * it is ready to go is indicated by the two 4-bit fields defined in the
2398  * kpm_spage structure as follows:
2399  * kp_mapped_flag bit[0:3] - the page is mapped cacheable or not
2400  * kp_mapped_flag bit[4:7] - the mapping is ready to go or not
2401  * If the bit KPM_MAPPED_GO is on, it indicates that the assembly tsb miss
2402  * handler can drop the mapping in regardless of the caching state of the
2403  * mapping. Otherwise, we will have C handler resolve the VAC conflict no
2404  * matter the page is currently mapped cacheable or non-cacheable.
2405  */
2406 #define KPM_MAPPEDS             0x1     /* small mapping valid, no conflict */
2407 #define KPM_MAPPEDSC            0x2     /* small mapping valid, conflict */
2408 #define KPM_MAPPED_GO           0x10    /* the mapping is ready to go */
2409 #define KPM_MAPPED_MASK         0xf

2411 /* Physical memseg address NULL marker */
2412 #define MSEG_NULLPTR_PA         -1

2414 /*
2415  * Memseg hash defines for kpm trap level tsbmiss handler.
2416  * Must be in sync w/ page.h .
2417  */
2418 #define SFMMU_MEM_HASH_SHIFT            0x9
2419 #define SFMMU_N_MEM_SLOTS               0x200
2420 #define SFMMU_MEM_HASH_ENTRY_SHIFT      3

2422 #ifndef _ASM
2423 #if (SFMMU_MEM_HASH_SHIFT != MEM_HASH_SHIFT)
2424 #error SFMMU_MEM_HASH_SHIFT != MEM_HASH_SHIFT
2425 #endif
2426 #if (SFMMU_N_MEM_SLOTS != N_MEM_SLOTS)
2427 #error SFMMU_N_MEM_SLOTS != N_MEM_SLOTS
2428 #endif

2430 /* Physical memseg address NULL marker */
2431 #define SFMMU_MEMSEG_NULLPTR_PA         -1

2433 /*
2434  * Check KCONTEXT to be zero, asm parts depend on that assumption.
2435  */
2436 #if (KCONTEXT != 0)
2437 #error KCONTEXT != 0
2438 #endif
2439 #endif  /* !_ASM */


2442 #endif /* _KERNEL */

2444 #ifndef _ASM
```

```
2445 /*
2446  * ctx, hmeblk, mlistlock and other stats for sfmmu
2447  */
2448 struct sfmmu_global_stat {
2449         int             sf_tsb_exceptions;      /* # of tsb exceptions */
2450         int             sf_tsb_raise_exception; /* # tsb exc. w/o TLB flush */

2452         int             sf_pagefaults;          /* # of pagefaults */

2454         int             sf_uhash_searches;      /* # of user hash searches */
2455         int             sf_uhash_links;         /* # of user hash links */
2456         int             sf_khash_searches;      /* # of kernel hash searches */
2457         int             sf_khash_links;         /* # of kernel hash links */

2459         int             sf_swapout;             /* # times hat swapped out */

2461         int             sf_tsb_alloc;           /* # TSB allocations */
2462         int             sf_tsb_allocfail;       /* # times TSB alloc fail */
2463         int             sf_tsb_sectsb_create;   /* # times second TSB added */

2465         int             sf_scd_1sttsb_alloc;    /* # SCD 1st TSB allocations */
2466         int             sf_scd_2ndtsb_alloc;    /* # SCD 2nd TSB allocations */
2467         int             sf_scd_1sttsb_allocfail; /* # SCD 1st TSB alloc fail */
2468         int             sf_scd_2ndtsb_allocfail; /* # SCD 2nd TSB alloc fail */


2471         int             sf_tteload8k;           /* calls to sfmmu_tteload */
2472         int             sf_tteload64k;          /* calls to sfmmu_tteload */
2473         int             sf_tteload512k;         /* calls to sfmmu_tteload */
2474         int             sf_tteload4m;           /* calls to sfmmu_tteload */
2475         int             sf_tteload32m;          /* calls to sfmmu_tteload */
2476         int             sf_tteload256m;         /* calls to sfmmu_tteload */

2478         int             sf_tsb_load8k;          /* # times loaded 8K tsbent */
2479         int             sf_tsb_load4m;          /* # times loaded 4M tsbent */

2481         int             sf_hblk_hit;            /* found hblk during tteload */
2482         int             sf_hblk8_ncreate;       /* static hblk8's created */
2483         int             sf_hblk8_nalloc;        /* static hblk8's allocated */
2484         int             sf_hblk1_ncreate;       /* static hblk1's created */
2485         int             sf_hblk1_nalloc;        /* static hblk1's allocated */
2486         int             sf_hblk_slab_cnt;       /* sfmmu8_cache slab creates */
2487         int             sf_hblk_reserve_cnt;    /* hblk_reserve usage */
2488         int             sf_hblk_recurse_cnt;    /* hblk_reserve owner reqs */
2489         int             sf_hblk_reserve_hit;    /* hblk_reserve hash hits */
2490         int             sf_get_free_success;    /* reserve list allocs */
2491         int             sf_get_free_throttle;   /* fails due to throttling */
2492         int             sf_get_free_fail;       /* fails due to empty list */
2493         int             sf_put_free_success;    /* reserve list frees */
2494         int             sf_put_free_fail;       /* fails due to full list */

2496         int             sf_pgcolor_conflict;    /* VAC conflict resolution */
2497         int             sf_uncache_conflict;    /* VAC conflict resolution */
2498         int             sf_unload_conflict;     /* VAC unload resolution */
2499         int             sf_ism_uncache;         /* VAC conflict resolution */
2500         int             sf_ism_recache;         /* VAC conflict resolution */
2501         int             sf_recache;             /* VAC conflict resolution */

2503         int             sf_steal_count;         /* # of hblks stolen */

2505         int             sf_pagesync;            /* # of pagesyncs */
2506         int             sf_clrwrt;              /* # of clear write perms */
2507         int             sf_pagesync_invalid;    /* pagesync with inv tte */

2509         int             sf_kernel_xcalls;       /* # of kernel cross calls */
2510         int             sf_user_xcalls;         /* # of user cross calls */
```



```
2512         int             sf_tsb_grow;            /* # of user tsb grows */
2513         int             sf_tsb_shrink;          /* # of user tsb shrinks */
2514         int             sf_tsb_resize_failures; /* # of user tsb resize */
2515         int             sf_tsb_reloc;           /* # of user tsb relocations */

2517         int             sf_user_vtop;           /* # of user vatopfn calls */

2519         int             sf_ctx_inv;             /* #times invalidate MMU ctx */

2521         int             sf_tlb_reprog_pgsz;     /* # times switch TLB pgsz */

2523         int             sf_region_remap_demap;  /* # times shme remap demap */

2525         int             sf_create_scd;          /* # times SCD is created */
2526         int             sf_join_scd;            /* # process joined scd */
2527         int             sf_leave_scd;           /* # process left scd */
2528         int             sf_destroy_scd;         /* # times SCD is destroyed */
2529 };
______unchanged_portion_omitted_
```

```
**********************************************************
   31077 Fri May  8 18:10:42 2015
new/usr/src/uts/sparc/os/syscall.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

342 /*
343  * Perform pre-system-call processing, including stopping for tracing,
344  * auditing, microstate-accounting, etc.
345  *
346  * This routine is called only if the t_pre_sys flag is set.  Any condition
347  * requiring pre-syscall handling must set the t_pre_sys flag.  If the
348  * condition is persistent, this routine will repost t_pre_sys.
349  */
350 int
351 pre_syscall(int arg0)
352 {
353         unsigned int code;
354         kthread_t *t = curthread;
355         proc_t *p = ttoproc(t);
356         klwp_t *lwp = ttolwp(t);
357         struct regs *rp = lwptoregs(lwp);
358         int     repost;

360         t->t_pre_sys = repost = 0;      /* clear pre-syscall processing flag */

362         ASSERT(t->t_schedflag & TS_DONT_SWAP);

362         syscall_mstate(LMS_USER, LMS_SYSTEM);

364         /*
365          * The syscall arguments in the out registers should be pointed to
366          * by lwp_ap.  If the args need to be copied so that the outs can
367          * be changed without losing the ability to get the args for /proc,
368          * they can be saved by save_syscall_args(), and lwp_ap will be
369          * restored by post_syscall().
370          */
371         ASSERT(lwp->lwp_ap == (long *)&rp->r_o0);

373         /*
374          * Make sure the thread is holding the latest credentials for the
375          * process.  The credentials in the process right now apply to this
376          * thread for the entire system call.
377          */
378         if (t->t_cred != p->p_cred) {
379                 cred_t *oldcred = t->t_cred;
380                 /*
381                  * DTrace accesses t_cred in probe context.  t_cred must
382                  * always be either NULL, or point to a valid, allocated cred
383                  * structure.
384                  */
385                 t->t_cred = crgetcred();
386                 crfree(oldcred);
387         }

389         /*
390          * Undo special arrangements to single-step the lwp
391          * so that a debugger will see valid register contents.
392          * Also so that the pc is valid for syncfpu().
```



```
393          * Also so that a syscall like exec() can be stepped.
394          */
395         if (lwp->lwp_pcb.pcb_step != STEP_NONE) {
396                 (void) prundostep();
397                 repost = 1;
398         }

400         /*
401          * Check for indirect system call in case we stop for tracing.
402          * Don't allow multiple indirection.
403          */
404         code = t->t_sysnum;
405         if (code == 0 && arg0 != 0) {           /* indirect syscall */
406                 code = arg0;
407                 t->t_sysnum = arg0;
408         }

410         /*
411          * From the proc(4) manual page:
412          * When entry to a system call is being traced, the traced process
413          * stops after having begun the call to the system but before the
414          * system call arguments have been fetched from the process.
415          * If proc changes the args we must refetch them after starting.
416          */
417         if (PTOU(p)->u_systrap) {
418                 if (prismember(&PTOU(p)->u_entrymask, code)) {
419                         /*
420                          * Recheck stop condition, now that lock is held.
421                          */
422                         mutex_enter(&p->p_lock);
423                         if (PTOU(p)->u_systrap &&
424                             prismember(&PTOU(p)->u_entrymask, code)) {
425                                 stop(PR_SYSENTRY, code);
426                                 /*
427                                  * Must refetch args since they were
428                                  * possibly modified by /proc.  Indicate
429                                  * that the valid copy is in the
430                                  * registers.
431                                  */
432                                 lwp->lwp_argsaved = 0;
433                                 lwp->lwp_ap = (long *)&rp->r_o0;
434                         }
435                         mutex_exit(&p->p_lock);
436                 }
437                 repost = 1;
438         }

440         if (lwp->lwp_sysabort) {
441                 /*
442                  * lwp_sysabort may have been set via /proc while the process
443                  * was stopped on PR_SYSENTRY.  If so, abort the system call.
444                  * Override any error from the copyin() of the arguments.
445                  */
446                 lwp->lwp_sysabort = 0;
447                 (void) set_errno(EINTR); /* sets post-sys processing */
448                 t->t_pre_sys = 1;       /* repost anyway */
449                 return (1);             /* don't do system call, return EINTR */
450         }

452         /* begin auditing for this syscall */
453         if (audit_active == C2AUDIT_LOADED) {
454                 uint32_t auditing = au_zone_getstate(NULL);

456                 if (auditing & AU_AUDIT_MASK) {
457                         int error;
458                         if (error = audit_start(T_SYSCALL, code, auditing, \
```

```
459                             0, lwp)) {
460                                 t->t_pre_sys = 1;       /* repost anyway */
461                                 lwp->lwp_error = 0;     /* for old drivers */
462                                 return (error);
463                         }
464                         repost = 1;
465                 }
466         }

468 #ifndef NPROBE
469         /* Kernel probe */
470         if (tnf_tracing_active) {
471                 TNF_PROBE_1(syscall_start, "syscall thread", /* CSTYLED */,
472                         tnf_sysnum,     sysnum,         t->t_sysnum);
473                 t->t_post_sys = 1;      /* make sure post_syscall runs */
474                 repost = 1;
475         }
476 #endif /* NPROBE */

478 #ifdef SYSCALLTRACE
479         if (syscalltrace) {
480                 int i;
481                 long *ap;
482                 char *cp;
483                 char *sysname;
484                 struct sysent *callp;

486                 if (code >= NSYSCALL)
487                         callp = &nosys_ent;     /* nosys has no args */
488                 else
489                         callp = LWP_GETSYSENT(lwp) + code;
490                 (void) save_syscall_args();
491                 mutex_enter(&systrace_lock);
492                 printf("%d: ", p->p_pid);
493                 if (code >= NSYSCALL)
494                         printf("0x%x", code);
495                 else {
496                         sysname = mod_getsysname(code);
497                         printf("%s[0x%x]", sysname == NULL ? "NULL" :
498                             sysname, code);
499                 }
500                 cp = "(";
501                 for (i = 0, ap = lwp->lwp_ap; i < callp->sy_narg; i++, ap++) {
502                         printf("%s%lx", cp, *ap);
503                         cp = ", ";
504                 }
505                 if (i)
506                         printf(")");
507                 printf(" %s id=0x%p\n", PTOU(p)->u_comm, curthread);
508                 mutex_exit(&systrace_lock);
509         }
510 #endif /* SYSCALLTRACE */

512         /*
513          * If there was a continuing reason for pre-syscall processing,
514          * set the t_pre_sys flag for the next system call.
515          */
516         if (repost)
517                 t->t_pre_sys = 1;
518         lwp->lwp_error = 0;     /* for old drivers */
519         lwp->lwp_badpriv = PRIV_NONE;   /* for privilege tracing */
520         return (0);
521 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   50033 Fri May  8 18:10:42 2015
new/usr/src/uts/sparc/v9/os/v9dep.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

863 void
864 lwp_swapin(kthread_t *tp)
865 {
866         struct machpcb *mpcb = lwptompcb(ttolwp(tp));

868         mpcb->mpcb_pa = va_to_pa(mpcb);
869         mpcb->mpcb_wbuf_pa = va_to_pa(mpcb->mpcb_wbuf);
870 }

863 /*
864  * Construct the execution environment for the user's signal
865  * handler and arrange for control to be given to it on return
866  * to userland.  The library code now calls setcontext() to
867  * clean up after the signal handler, so sigret() is no longer
868  * needed.
869  */
870 int
871 sendsig(int sig, k_siginfo_t *sip, void (*hdlr)())
872 {
873         /*
874          * 'volatile' is needed to ensure that values are
875          * correct on the error return from on_fault().
876          */
877         volatile int minstacksz; /* min stack required to catch signal */
878         int newstack = 0;       /* if true, switching to altstack */
879         label_t ljb;
880         caddr_t sp;
881         struct regs *volatile rp;
882         klwp_t *lwp = ttolwp(curthread);
883         proc_t *volatile p = ttoproc(curthread);
884         int fpq_size = 0;
885         struct sigframe {
886                 struct frame frwin;
887                 ucontext_t uc;
888         };
889         siginfo_t *sip_addr;
890         struct sigframe *volatile fp;
891         ucontext_t *volatile tuc = NULL;
892         char *volatile xregs = NULL;
893         volatile size_t xregs_size = 0;
894         gwindows_t *volatile gwp = NULL;
895         volatile int gwin_size = 0;
896         kfpu_t *fpp;
897         struct machpcb *mpcb;
898         volatile int watched = 0;
899         volatile int watched2 = 0;
900         caddr_t tos;

902         /*
903          * Make sure the current last user window has been flushed to
904          * the stack save area before we change the sp.
905          * Restore register window if a debugger modified it.
906          */
```



```
907         (void) flush_user_windows_to_stack(NULL);
908         if (lwp->lwp_pcb.pcb_xregstat != XREGNONE)
909                 xregrestore(lwp, 0);

911         mpcb = lwptompcb(lwp);
912         rp = lwptoregs(lwp);

914         /*
915          * Clear the watchpoint return stack pointers.
916          */
917         mpcb->mpcb_rsp[0] = NULL;
918         mpcb->mpcb_rsp[1] = NULL;

920         minstacksz = sizeof (struct sigframe);

922         /*
923          * We know that sizeof (siginfo_t) is stack-aligned:
924          * 128 bytes for ILP32, 256 bytes for LP64.
925          */
926         if (sip != NULL)
927                 minstacksz += sizeof (siginfo_t);

929         /*
930          * These two fields are pointed to by ABI structures and may
931          * be of arbitrary length. Size them now so we know how big
932          * the signal frame has to be.
933          */
934         fpp = lwptofpu(lwp);
935         fpp->fpu_fprs = _fp_read_fprs();
936         if ((fpp->fpu_en) || (fpp->fpu_fprs & FPRS_FEF)) {
937                 fpq_size = fpp->fpu_q_entrysize * fpp->fpu_qcnt;
938                 minstacksz += SA(fpq_size);
939         }

941         mpcb = lwptompcb(lwp);
942         if (mpcb->mpcb_wbcnt != 0) {
943                 gwin_size = (mpcb->mpcb_wbcnt * sizeof (struct rwindow)) +
944                     (SPARC_MAXREGWINDOW * sizeof (caddr_t)) + sizeof (long);
945                 minstacksz += SA(gwin_size);
946         }

948         /*
949          * Extra registers, if support by this platform, may be of arbitrary
950          * length. Size them now so we know how big the signal frame has to be.
951          * For sparcv9 _LP64 user programs, use asrs instead of the xregs.
952          */
953         minstacksz += SA(xregs_size);

955         /*
956          * Figure out whether we will be handling this signal on
957          * an alternate stack specified by the user. Then allocate
958          * and validate the stack requirements for the signal handler
959          * context. on_fault will catch any faults.
960          */
961         newstack = (sigismember(&PTOU(curproc)->u_sigonstack, sig) &&
962             !(lwp->lwp_sigaltstack.ss_flags & (SS_ONSTACK|SS_DISABLE)));

964         tos = (caddr_t)rp->r_sp + STACK_BIAS;
965         /*
966          * Force proper stack pointer alignment, even in the face of a
967          * misaligned stack pointer from user-level before the signal.
968          * Don't use the SA() macro because that rounds up, not down.
969          */
970         tos = (caddr_t)((uintptr_t)tos & ~(STACK_ALIGN - 1ul));

972         if (newstack != 0) {
```

```
 973                 fp = (struct sigframe *)
 974                     (SA((uintptr_t)lwp->lwp_sigaltstack.ss_sp) +
 975                     SA((int)lwp->lwp_sigaltstack.ss_size) - STACK_ALIGN -
 976                     SA(minstacksz));
 977         } else {
 978                 /*
 979                  * If we were unable to flush all register windows to
 980                  * the stack and we are not now on an alternate stack,
 981                  * just dump core with a SIGSEGV back in psig().
 982                  */
 983                 if (sig == SIGSEGV &&
 984                     mpcb->mpcb_wbcnt != 0 &&
 985                     !(lwp->lwp_sigaltstack.ss_flags & SS_ONSTACK))
 986                         return (0);
 987                 fp = (struct sigframe *)(tos - SA(minstacksz));
 988                 /*
 989                  * Could call grow here, but stack growth now handled below
 990                  * in code protected by on_fault().
 991                  */
 992         }
 993         sp = (caddr_t)fp + sizeof (struct sigframe);

 995         /*
 996          * Make sure process hasn't trashed its stack.
 997          */
 998         if ((caddr_t)fp >= p->p_usrstack ||
 999             (caddr_t)fp + SA(minstacksz) >= p->p_usrstack) {
1000 #ifdef DEBUG
1001                 printf("sendsig: bad signal stack cmd=%s, pid=%d, sig=%d\n",
1002                     PTOU(p)->u_comm, p->p_pid, sig);
1003                 printf("sigsp = 0x%p, action = 0x%p, upc = 0x%lx\n",
1004                     (void *)fp, (void *)hdlr, rp->r_pc);
1005                 printf("fp above USRSTACK\n");
1006 #endif
1007                 return (0);
1008         }

1010         watched = watch_disable_addr((caddr_t)fp, SA(minstacksz), S_WRITE);
1011         if (on_fault(&ljb))
1012                 goto badstack;

1014         tuc = kmem_alloc(sizeof (ucontext_t), KM_SLEEP);
1015         savecontext(tuc, &lwp->lwp_sigoldmask);

1017         /*
1018          * save extra register state if it exists
1019          */
1020         if (xregs_size != 0) {
1021                 xregs_setptr(lwp, tuc, sp);
1022                 xregs = kmem_alloc(xregs_size, KM_SLEEP);
1023                 xregs_get(lwp, xregs);
1024                 copyout_noerr(xregs, sp, xregs_size);
1025                 kmem_free(xregs, xregs_size);
1026                 xregs = NULL;
1027                 sp += SA(xregs_size);
1028         }

1030         copyout_noerr(tuc, &fp->uc, sizeof (*tuc));
1031         kmem_free(tuc, sizeof (*tuc));
1032         tuc = NULL;

1034         if (sip != NULL) {
1035                 zoneid_t zoneid;

1037                 uzero(sp, sizeof (siginfo_t));
1038                 if (SI_FROMUSER(sip) &&
```



```
1039                     (zoneid = p->p_zone->zone_id) != GLOBAL_ZONEID &&
1040                     zoneid != sip->si_zoneid) {
1041                         k_siginfo_t sani_sip = *sip;
1042                         sani_sip.si_pid = p->p_zone->zone_zsched->p_pid;
1043                         sani_sip.si_uid = 0;
1044                         sani_sip.si_ctid = -1;
1045                         sani_sip.si_zoneid = zoneid;
1046                         copyout_noerr(&sani_sip, sp, sizeof (sani_sip));
1047                 } else {
1048                         copyout_noerr(sip, sp, sizeof (*sip));
1049                 }
1050                 sip_addr = (siginfo_t *)sp;
1051                 sp += sizeof (siginfo_t);

1053                 if (sig == SIGPROF &&
1054                     curthread->t_rprof != NULL &&
1055                     curthread->t_rprof->rp_anystate) {
1056                         /*
1057                          * We stand on our head to deal with
1058                          * the real time profiling signal.
1059                          * Fill in the stuff that doesn't fit
1060                          * in a normal k_siginfo structure.
1061                          */
1062                         int i = sip->si_nsysarg;
1063                         while (--i >= 0) {
1064                                 sulword_noerr(
1065                                     (ulong_t *)&sip_addr->si_sysarg[i],
1066                                     (ulong_t)lwp->lwp_arg[i]);
1067                         }
1068                         copyout_noerr(curthread->t_rprof->rp_state,
1069                             sip_addr->si_mstate,
1070                             sizeof (curthread->t_rprof->rp_state));
1071                 }
1072         } else {
1073                 sip_addr = (siginfo_t *)NULL;
1074         }

1076         /*
1077          * When flush_user_windows_to_stack() can't save all the
1078          * windows to the stack, it puts them in the lwp's pcb.
1079          */
1080         if (gwin_size != 0) {
1081                 gwp = kmem_alloc(gwin_size, KM_SLEEP);
1082                 getgwins(lwp, gwp);
1083                 sulword_noerr(&fp->uc.uc_mcontext.gwins, (ulong_t)sp);
1084                 copyout_noerr(gwp, sp, gwin_size);
1085                 kmem_free(gwp, gwin_size);
1086                 gwp = NULL;
1087                 sp += SA(gwin_size);
1088         } else
1089                 sulword_noerr(&fp->uc.uc_mcontext.gwins, (ulong_t)NULL);

1091         if (fpq_size != 0) {
1092                 struct fq *fqp = (struct fq *)sp;
1093                 sulword_noerr(&fp->uc.uc_mcontext.fpregs.fpu_q, (ulong_t)fqp);
1094                 copyout_noerr(mpcb->mpcb_fpu_q, fqp, fpq_size);

1096                 /*
1097                  * forget the fp queue so that the signal handler can run
1098                  * without being harrassed--it will do a setcontext that will
1099                  * re-establish the queue if there still is one
1100                  *
1101                  * NOTE: fp_runq() relies on the qcnt field being zeroed here
1102                  *      to terminate its processing of the queue after signal
1103                  *      delivery.
1104                  */
```

```
1105                 mpcb->mpcb_fpu->fpu_qcnt = 0;
1106                 sp += SA(fpq_size);

1108                 /* Also, syscall needs to know about this */
1109                 mpcb->mpcb_flags |= FP_TRAPPED;

1111         } else {
1112                 sulword_noerr(&fp->uc.uc_mcontext.fpregs.fpu_q, (ulong_t)NULL);
1113                 suword8_noerr(&fp->uc.uc_mcontext.fpregs.fpu_qcnt, 0);
1114         }


1117         /*
1118          * Since we flushed the user's windows and we are changing his
1119          * stack pointer, the window that the user will return to will
1120          * be restored from the save area in the frame we are setting up.
1121          * We copy in save area for old stack pointer so that debuggers
1122          * can do a proper stack backtrace from the signal handler.
1123          */
1124         if (mpcb->mpcb_wbcnt == 0) {
1125                 watched2 = watch_disable_addr(tos, sizeof (struct rwindow),
1126                     S_READ);
1127                 ucopy(tos, &fp->frwin, sizeof (struct rwindow));
1128         }

1130         lwp->lwp_oldcontext = (uintptr_t)&fp->uc;

1132         if (newstack != 0) {
1133                 lwp->lwp_sigaltstack.ss_flags |= SS_ONSTACK;

1135                 if (lwp->lwp_ustack) {
1136                         copyout_noerr(&lwp->lwp_sigaltstack,
1137                             (stack_t *)lwp->lwp_ustack, sizeof (stack_t));
1138                 }
1139         }

1141         no_fault();
1142         mpcb->mpcb_wbcnt = 0;           /* let user go on */

1144         if (watched2)
1145                 watch_enable_addr(tos, sizeof (struct rwindow), S_READ);
1146         if (watched)
1147                 watch_enable_addr((caddr_t)fp, SA(minstacksz), S_WRITE);

1149         /*
1150          * Set up user registers for execution of signal handler.
1151          */
1152         rp->r_sp = (uintptr_t)fp - STACK_BIAS;
1153         rp->r_pc = (uintptr_t)hdlr;
1154         rp->r_npc = (uintptr_t)hdlr + 4;
1155         /* make sure %asi is ASI_PNF */
1156         rp->r_tstate &= ~((uint64_t)TSTATE_ASI_MASK << TSTATE_ASI_SHIFT);
1157         rp->r_tstate |= ((uint64_t)ASI_PNF << TSTATE_ASI_SHIFT);
1158         rp->r_o0 = sig;
1159         rp->r_o1 = (uintptr_t)sip_addr;
1160         rp->r_o2 = (uintptr_t)&fp->uc;
1161         /*
1162          * Don't set lwp_eosys here.  sendsig() is called via psig() after
1163          * lwp_eosys is handled, so setting it here would affect the next
1164          * system call.
1165          */
1166         return (1);

1168 badstack:
1169         no_fault();
1170         if (watched2)
```



```
1171                 watch_enable_addr(tos, sizeof (struct rwindow), S_READ);
1172         if (watched)
1173                 watch_enable_addr((caddr_t)fp, SA(minstacksz), S_WRITE);
1174         if (tuc)
1175                 kmem_free(tuc, sizeof (ucontext_t));
1176         if (xregs)
1177                 kmem_free(xregs, xregs_size);
1178         if (gwp)
1179                 kmem_free(gwp, gwin_size);
1180 #ifdef DEBUG
1181         printf("sendsig: bad signal stack cmd=%s, pid=%d, sig=%d\n",
1182             PTOU(p)->u_comm, p->p_pid, sig);
1183         printf("on fault, sigsp = %p, action = %p, upc = 0x%lx\n",
1184             (void *)fp, (void *)hdlr, rp->r_pc);
1185 #endif
1186         return (0);
1187 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   11466 Fri May  8 18:10:42 2015
new/usr/src/uts/sparc/v9/vm/seg_nf.c
use NULL dump segop as a shorthand for no-op
Instead of forcing every segment driver to implement a dummy function that
does nothing, handle NULL dump segop function pointer as a no-op shorthand.
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
use NULL setpagesize segop as a shorthand for ENOTSUP
Instead of forcing every segment driver to implement a dummp function to
return (hopefully) ENOTSUP, handle NULL setpagesize segop function pointer
as "return ENOTSUP" shorthand.
use NULL getmemid segop as a shorthand for ENODEV
Instead of forcing every segment driver to implement a dummy function to
return (hopefully) ENODEV, handle NULL getmemid segop function pointer as
"return ENODEV" shorthand.
segop_getpolicy already checks for a NULL op
no need for bad-op segment op functions
The segment drivers have a number of bad-op functions that simply panic.
Keeping the function pointer NULL will accomplish the same thing in most
cases.  In other cases, keeping the function pointer NULL will result in
proper error code being returned.
use C99 initializers in segment ops structures
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
```

```
   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 /* Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T */
  27 /* All Rights Reserved */

  29 /*
  30  * Portions of this source code were derived from Berkeley 4.3 BSD
  31  * under license from the Regents of the University of California.
  32  */

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



```
  34 /*
  35  * VM - segment for non-faulting loads.
  36  */

  38 #include <sys/types.h>
  39 #include <sys/t_lock.h>
  40 #include <sys/param.h>
  41 #include <sys/mman.h>
  42 #include <sys/errno.h>
  43 #include <sys/kmem.h>
  44 #include <sys/cmn_err.h>
  45 #include <sys/vnode.h>
  46 #include <sys/proc.h>
  47 #include <sys/conf.h>
  48 #include <sys/debug.h>
  49 #include <sys/archsystm.h>
  50 #include <sys/lgrp.h>

  52 #include <vm/page.h>
  53 #include <vm/hat.h>
  54 #include <vm/as.h>
  55 #include <vm/seg.h>
  56 #include <vm/vpage.h>

  58 /*
  59  * Private seg op routines.
  60  */
  61 static int      segnf_dup(struct seg *seg, struct seg *newseg);
  62 static int      segnf_unmap(struct seg *seg, caddr_t addr, size_t len);
  63 static void     segnf_free(struct seg *seg);
  64 static faultcode_t segnf_nomap(void);
  65 static int      segnf_setprot(struct seg *seg, caddr_t addr,
  66                     size_t len, uint_t prot);
  67 static int      segnf_checkprot(struct seg *seg, caddr_t addr,
  68                     size_t len, uint_t prot);
  71 static void     segnf_badop(void);
  69 static int      segnf_nop(void);
  70 static int      segnf_getprot(struct seg *seg, caddr_t addr,
  71                     size_t len, uint_t *protv);
  72 static u_offset_t segnf_getoffset(struct seg *seg, caddr_t addr);
  73 static int      segnf_gettype(struct seg *seg, caddr_t addr);
  74 static int      segnf_getvp(struct seg *seg, caddr_t addr, struct vnode **vpp);
  78 static void     segnf_dump(struct seg *seg);
  75 static int      segnf_pagelock(struct seg *seg, caddr_t addr, size_t len,
  76                     struct page ***ppp, enum lock_type type, enum seg_rw rw);


  79 const struct seg_ops segnf_ops = {
  80         .dup            = segnf_dup,
  81         .unmap          = segnf_unmap,
  82         .free           = segnf_free,
  83         .fault          = (faultcode_t (*)(struct hat *, struct seg *, caddr_t,
  84             size_t, enum fault_type, enum seg_rw))segnf_nomap,
  85         .faulta         = (faultcode_t (*)(struct seg *, caddr_t)) segnf_nomap,
  86         .setprot        = segnf_setprot,
  87         .checkprot      = segnf_checkprot,
  88         .sync           = (int (*)(struct seg *, caddr_t, size_t, int, uint_t))
  89                 segnf_nop,
  90         .incore         = (size_t (*)(struct seg *, caddr_t, size_t, char *))
  91                 segnf_nop,
  92         .lockop         = (int (*)(struct seg *, caddr_t, size_t, int, int,
  93             ulong_t *, size_t))segnf_nop,
  94         .getprot        = segnf_getprot,
  95         .getoffset      = segnf_getoffset,
  96         .gettype        = segnf_gettype,
  97         .getvp          = segnf_getvp,
```

```
 98         .advise         = (int (*)(struct seg *, caddr_t, size_t, uint_t))
 99                 segnf_nop,
100         .pagelock       = segnf_pagelock,
 81 static int      segnf_setpagesize(struct seg *seg, caddr_t addr, size_t len,
 82                     uint_t szc);
 83 static int      segnf_getmemid(struct seg *seg, caddr_t addr, memid_t *memidp);
 84 static lgrp_mem_policy_info_t   *segnf_getpolicy(struct seg *seg,
 85     caddr_t addr);


 88 struct seg_ops segnf_ops = {
 89         segnf_dup,
 90         segnf_unmap,
 91         segnf_free,
 92         (faultcode_t (*)(struct hat *, struct seg *, caddr_t, size_t,
 93             enum fault_type, enum seg_rw))
 94                 segnf_nomap,            /* fault */
 95         (faultcode_t (*)(struct seg *, caddr_t))
 96                 segnf_nomap,            /* faulta */
 97         segnf_setprot,
 98         segnf_checkprot,
 99         (int (*)())segnf_badop,         /* kluster */
100         (size_t (*)(struct seg *))NULL, /* swapout */
101         (int (*)(struct seg *, caddr_t, size_t, int, uint_t))
102                 segnf_nop,              /* sync */
103         (size_t (*)(struct seg *, caddr_t, size_t, char *))
104                 segnf_nop,              /* incore */
105         (int (*)(struct seg *, caddr_t, size_t, int, int, ulong_t *, size_t))
106                 segnf_nop,              /* lockop */
107         segnf_getprot,
108         segnf_getoffset,
109         segnf_gettype,
110         segnf_getvp,
111         (int (*)(struct seg *, caddr_t, size_t, uint_t))
112                 segnf_nop,              /* advise */
113         segnf_dump,
114         segnf_pagelock,
115         segnf_setpagesize,
116         segnf_getmemid,
117         segnf_getpolicy,
101 };
```

_____unchanged_portion_omitted_

```
409 static void
410 segnf_badop(void)
411 {
412         panic("segnf_badop");
413         /*NOTREACHED*/
414 }
```

```
392 static int
393 segnf_nop(void)
394 {
395         return (0);
396 }
```

_____unchanged_portion_omitted_

```
462 /*
463  * segnf pages are not dumped, so we just return
464  */
465 /* ARGSUSED */
466 static void
467 segnf_dump(struct seg *seg)
468 {}
```

```
438 /*ARGSUSED*/
```



```
439 static int
440 segnf_pagelock(struct seg *seg, caddr_t addr, size_t len,
441     struct page ***ppp, enum lock_type type, enum seg_rw rw)
442 {
443         return (ENOTSUP);
476 }

478 /*ARGSUSED*/
479 static int
480 segnf_setpagesize(struct seg *seg, caddr_t addr, size_t len,
481     uint_t szc)
482 {
483         return (ENOTSUP);
484 }

486 /*ARGSUSED*/
487 static int
488 segnf_getmemid(struct seg *seg, caddr_t addr, memid_t *memidp)
489 {
490         return (ENODEV);
491 }

493 /*ARGSUSED*/
494 static lgrp_mem_policy_info_t *
495 segnf_getpolicy(struct seg *seg, caddr_t addr)
496 {
497         return (NULL);
444 }
```

_____unchanged_portion_omitted_

****
23606 Fri May 8 18:10:42 2015
new/usr/src/uts/sun4/io/rootnex.c
const-ify make segment ops structures
There is no reason to keep the segment ops structures writable.
****
_____***unchanged_portion_omitted_***

```
 697 /*
 698  * Shorthand defines
 699  */

 701 #define DMAOBJ_PP_PP    dmao_obj.pp_obj.pp_pp
 702 #define DMAOBJ_PP_OFF   dmao_ogj.pp_obj.pp_offset
 703 #define ALO             dma_lim->dlim_addr_lo
 704 #define AHI             dma_lim->dlim_addr_hi
 705 #define OBJSIZE         dmareq->dmar_object.dmao_size
 706 #define ORIGVADDR       dmareq->dmar_object.dmao_obj.virt_obj.v_addr
 707 #define RED             ((mp->dmai_rflags & DDI_DMA_REDZONE)? 1 : 0)
 708 #define DIRECTION       (mp->dmai_rflags & DDI_DMA_RDWR)

 710 /*
 711  * rootnex_map_fault:
 712  *
 713  *      fault in mappings for requestors
 714  */

 716 /*ARGSUSED*/
 717 static int
 718 rootnex_map_fault(dev_info_t *dip, dev_info_t *rdip,
 719     struct hat *hat, struct seg *seg, caddr_t addr,
 720     struct devpage *dp, pfn_t pfn, uint_t prot, uint_t lock)
 721 {
 722         extern const struct seg_ops segdev_ops;
 722         extern struct seg_ops segdev_ops;

 724         DPRINTF(ROOTNEX_MAP_DEBUG, ("rootnex_map_fault: address <%p> "
 725             "pfn <%lx>", (void *)addr, pfn));
 726         DPRINTF(ROOTNEX_MAP_DEBUG, (" Seg <%s>\n",
 727             seg->s_ops == &segdev_ops ? "segdev" :
 728             seg == &kvseg ? "segkmem" : "NONE!"));

 730         /*
 731          * This is all terribly broken, but it is a start
 732          *
 733          * XXX  Note that this test means that segdev_ops
 734          *      must be exported from seg_dev.c.
 735          * XXX  What about devices with their own segment drivers?
 736          */
 737         if (seg->s_ops == &segdev_ops) {
 738                 register struct segdev_data *sdp =
 739                     (struct segdev_data *)seg->s_data;

 741                 if (hat == NULL) {
 742                         /*
 743                          * This is one plausible interpretation of
 744                          * a null hat i.e. use the first hat on the
 745                          * address space hat list which by convention is
 746                          * the hat of the system MMU.  At alternative
 747                          * would be to panic .. this might well be better ..
 748                          */
 749                         ASSERT(AS_READ_HELD(seg->s_as, &seg->s_as->a_lock));
 750                         hat = seg->s_as->a_hat;
 751                         cmn_err(CE_NOTE, "rootnex_map_fault: nil hat");
 752                 }
```



```
 753                 hat_devload(hat, addr, MMU_PAGESIZE, pfn, prot | sdp->hat_attr,
 754                     (lock ? HAT_LOAD_LOCK : HAT_LOAD));
 755         } else if (seg == &kvseg && dp == (struct devpage *)0) {
 756                 hat_devload(kas.a_hat, addr, MMU_PAGESIZE, pfn, prot,
 757                     HAT_LOAD_LOCK);
 758         } else
 759                 return (DDI_FAILURE);
 760         return (DDI_SUCCESS);
 761 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   14116 Fri May  8 18:10:42 2015
new/usr/src/uts/sun4/os/mlsetup.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
```

```
/*
 * 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/systm.h>
#include <sys/archsystm.h>
#include <sys/machsystm.h>
#include <sys/disp.h>
#include <sys/autoconf.h>
#include <sys/promif.h>
#include <sys/prom_plat.h>
#include <sys/promimpl.h>
#include <sys/platform_module.h>
#include <sys/clock.h>
#include <sys/pte.h>
#include <sys/scb.h>
#include <sys/cpu.h>
#include <sys/stack.h>
#include <sys/intreg.h>
#include <sys/ivintr.h>
#include <vm/as.h>
#include <vm/hat_sfmmu.h>
#include <sys/reboot.h>
#include <sys/sysmacros.h>
#include <sys/vtrace.h>
#include <sys/trap.h>
#include <sys/machtrap.h>
#include <sys/privregs.h>
#include <sys/machpcb.h>
#include <sys/proc.h>
#include <sys/cpupart.h>
#include <sys/pset.h>
#include <sys/cpu_module.h>
```



```
#include <sys/copyops.h>
#include <sys/panic.h>
#include <sys/bootconf.h>       /* for bootops */
#include <sys/pg.h>
#include <sys/kdi.h>
#include <sys/fpras.h>

#include <sys/prom_debug.h>
#include <sys/debug.h>

#include <sys/sunddi.h>
#include <sys/lgrp.h>
#include <sys/traptrace.h>

#include <sys/kobj_impl.h>
#include <sys/kdi_machimpl.h>

/*
 * External Routines:
 */
extern void map_wellknown_devices(void);
extern void hsvc_setup(void);
extern void mach_descrip_startup_init(void);
extern void mach_soft_state_init(void);

int     dcache_size;
int     dcache_linesize;
int     icache_size;
int     icache_linesize;
int     ecache_size;
int     ecache_alignsize;
int     ecache_associativity;
int     ecache_setsize;                 /* max possible e$ setsize */
int     cpu_setsize;                    /* max e$ setsize of configured cpus */
int     dcache_line_mask;               /* spitfire only */
int     vac_size;                       /* cache size in bytes */
uint_t  vac_mask;                       /* VAC alignment consistency mask */
int     vac_shift;                      /* log2(vac_size) for ppmapout() */
int     vac = 0;        /* virtual address cache type (none == 0) */

/*
 * fpRAS.  An individual sun4* machine class (or perhaps subclass,
 * eg sun4u/cheetah) must set fpras_implemented to indicate that it implements
 * the fpRAS feature.  The feature can be suppressed by setting fpras_disable
 * or the mechanism can be disabled for individual copy operations with
 * fpras_disableids.  All these are checked in post_startup() code so
 * fpras_disable and fpras_disableids can be set in /etc/system.
 * If/when fpRAS is implemented on non-sun4 architectures these
 * definitions will need to move up to the common level.
 */
int     fpras_implemented;
int     fpras_disable;
int     fpras_disableids;

/*
 * Static Routines:
 */
static void kern_splr_preprom(void);
static void kern_splx_postprom(void);

/*
 * Setup routine called right before main(). Interposing this function
 * before main() allows us to call it in a machine-independent fashion.
 */

void
```

```
122 mlsetup(struct regs *rp, kfpu_t *fp)
123 {
124         struct machpcb *mpcb;

126         extern char t0stack[];
127         extern struct classfuncs sys_classfuncs;
128         extern disp_t cpu0_disp;
129         unsigned long long pa;

131 #ifdef TRAPTRACE
132         TRAP_TRACE_CTL *ctlp;
133 #endif /* TRAPTRACE */

135         /* drop into kmdb on boot -d */
136         if (boothowto & RB_DEBUGENTER)
137                 kmdb_enter();

139         /*
140          * initialize cpu_self
141          */
142         cpu0.cpu_self = &cpu0;

144         /*
145          * initialize t0
146          */
147         t0.t_stk = (caddr_t)rp - REGOFF;
148         /* Can't use va_to_pa here - wait until prom_ initialized */
149         t0.t_stkbase = t0stack;
150         t0.t_pri = maxclsyspri - 3;
151         t0.t_schedflag = 0;
151         t0.t_schedflag = TS_LOAD | TS_DONT_SWAP;
152         t0.t_procp = &p0;
153         t0.t_plockp = &p0lock.pl_lock;
154         t0.t_lwp = &lwp0;
155         t0.t_forw = &t0;
156         t0.t_back = &t0;
157         t0.t_next = &t0;
158         t0.t_prev = &t0;
159         t0.t_cpu = &cpu0;                       /* loaded by _start */
160         t0.t_disp_queue = &cpu0_disp;
161         t0.t_bind_cpu = PBIND_NONE;
162         t0.t_bind_pset = PS_NONE;
163         t0.t_bindflag = (uchar_t)default_binding_mode;
164         t0.t_cpupart = &cp_default;
165         t0.t_clfuncs = &sys_classfuncs.thread;
166         t0.t_copyops = NULL;
167         THREAD_ONPROC(&t0, CPU);

169         lwp0.lwp_thread = &t0;
170         lwp0.lwp_procp = &p0;
171         lwp0.lwp_regs = (void *)rp;
172         t0.t_tid = p0.p_lwpcnt = p0.p_lwprcnt = p0.p_lwpid = 1;

174         mpcb = lwptompcb(&lwp0);
175         mpcb->mpcb_fpu = fp;
176         mpcb->mpcb_fpu->fpu_q = mpcb->mpcb_fpu_q;
177         mpcb->mpcb_thread = &t0;
178         lwp0.lwp_fpu = (void *)mpcb->mpcb_fpu;

180         p0.p_exec = NULL;
181         p0.p_stat = SRUN;
182         p0.p_flag = SSYS;
183         p0.p_tlist = &t0;
184         p0.p_stksize = 2*PAGESIZE;
185         p0.p_stkpageszc = 0;
186         p0.p_as = &kas;
```



```
187         p0.p_lockp = &p0lock;
188         p0.p_utraps = NULL;
189         p0.p_brkpageszc = 0;
190         p0.p_t1_lgrpid = LGRP_NONE;
191         p0.p_tr_lgrpid = LGRP_NONE;
192         sigorset(&p0.p_ignore, &ignoredefault);

194         CPU->cpu_thread = &t0;
195         CPU->cpu_dispthread = &t0;
196         bzero(&cpu0_disp, sizeof (disp_t));
197         CPU->cpu_disp = &cpu0_disp;
198         CPU->cpu_disp->disp_cpu = CPU;
199         CPU->cpu_idle_thread = &t0;
200         CPU->cpu_flags = CPU_RUNNING;
201         CPU->cpu_id = getprocessorid();
202         CPU->cpu_dispatch_pri = t0.t_pri;

204         /*
205          * Initialize thread/cpu microstate accounting
206          */
207         init_mstate(&t0, LMS_SYSTEM);
208         init_cpu_mstate(CPU, CMS_SYSTEM);

210         /*
211          * Initialize lists of available and active CPUs.
212          */
213         cpu_list_init(CPU);

215         cpu_vm_data_init(CPU);

217         pg_cpu_bootstrap(CPU);

219         (void) prom_set_preprom(kern_splr_preprom);
220         (void) prom_set_postprom(kern_splx_postprom);
221         PRM_INFO("mlsetup: now ok to call prom_printf");

223         mpcb->mpcb_pa = va_to_pa(t0.t_stk);

225         /*
226          * Claim the physical and virtual resources used by panicbuf,
227          * then map panicbuf.  This operation removes the phys and
228          * virtual addresses from the free lists.
229          */
230         if (prom_claim_virt(PANICBUFSIZE, panicbuf) != panicbuf)
231                 prom_panic("Can't claim panicbuf virtual address");

233         if (prom_retain("panicbuf", PANICBUFSIZE, MMU_PAGESIZE, &pa) != 0)
234                 prom_panic("Can't allocate retained panicbuf physical address");

236         if (prom_map_phys(-1, PANICBUFSIZE, panicbuf, pa) != 0)
237                 prom_panic("Can't map panicbuf");

239         PRM_DEBUG(panicbuf);
240         PRM_DEBUG(pa);

242         /*
243          * Negotiate hypervisor services, if any
244          */
245         hsvc_setup();
246         mach_soft_state_init();

248 #ifdef TRAPTRACE
249         /*
250          * initialize the trap trace buffer for the boot cpu
251          * XXX todo, dynamically allocate this buffer too
252          */
```

```
253         ctlp = &trap_trace_ctl[CPU->cpu_id];
254         ctlp->d.vaddr_base = trap_tr0;
255         ctlp->d.offset = ctlp->d.last_offset = 0;
256         ctlp->d.limit = TRAP_TSIZE;             /* XXX dynamic someday */
257         ctlp->d.paddr_base = va_to_pa(trap_tr0);
258 #endif /* TRAPTRACE */

260         /*
261          * Initialize the Machine Description kernel framework
262          */

264         mach_descrip_startup_init();

266         /*
267          * initialize HV trap trace buffer for the boot cpu
268          */
269         mach_htraptrace_setup(CPU->cpu_id);
270         mach_htraptrace_configure(CPU->cpu_id);

272         /*
273          * lgroup framework initialization. This must be done prior
274          * to devices being mapped.
275          */
276         lgrp_init(LGRP_INIT_STAGE1);

278         cpu_setup();

280         if (boothowto & RB_HALT) {
281                 prom_printf("unix: kernel halted by -h flag\n");
282                 prom_enter_mon();
283         }

285         setcputype();
286         map_wellknown_devices();
287         setcpudelay();
288 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   51350 Fri May  8 18:10:43 2015
new/usr/src/uts/sun4/os/trap.c
remove whole-process swapping
Long before Unix supported paging, it used process swapping to reclaim
memory.  The code is there and in theory it runs when we get *extremely* low
on memory.  In practice, it never runs since the definition of low-on-memory
is antiquated. (XXX: define what antiquated means)
You can check the number of swapout/swapin events with kstats:
$ kstat -p ::vm:swapin ::vm:swapout
**********************************************************
______unchanged_portion_omitted_

#if defined(SF_ERRATA_23) || defined(SF_ERRATA_30) /* call ... illegal-insn */
int     ill_calls;
#endif

/*
 * Currently, the only PREFETCH/PREFETCHA instructions which cause traps
 * are the "strong" prefetches (fcn=20-23).  But we check for all flavors of
 * PREFETCH, in case some future variant also causes a DATA_MMU_MISS.
 */
#define IS_PREFETCH(i)  (((i) & 0xc1780000) == 0xc1680000)

#define IS_FLUSH(i)     (((i) & 0xc1f80000) == 0x81d80000)
#define IS_SWAP(i)      (((i) & 0xc1f80000) == 0xc0780000)
#define IS_LDSTUB(i)    (((i) & 0xc1f80000) == 0xc0680000)
#define IS_FLOAT(i)     (((i) & 0x1000000) != 0)
#define IS_STORE(i)     (((i) >> 21) & 1)

/*
 * Called from the trap handler when a processor trap occurs.
 */
/*VARARGS2*/
void
trap(struct regs *rp, caddr_t addr, uint32_t type, uint32_t mmu_fsr)
{
        proc_t *p = ttoproc(curthread);
        klwp_id_t lwp = ttolwp(curthread);
        struct machpcb *mpcb = NULL;
        k_siginfo_t siginfo;
        uint_t op3, fault = 0;
        int stepped = 0;
        greg_t oldpc;
        int mstate;
        char *badaddr;
        faultcode_t res;
        enum fault_type fault_type;
        enum seg_rw rw;
        uintptr_t lofault;
        label_t *onfault;
        int instr;
        int iskernel;
        int watchcode;
        int watchpage;
        extern faultcode_t pagefault(caddr_t, enum fault_type,
            enum seg_rw, int);
#ifdef sun4v
        extern boolean_t tick_stick_emulation_active;
#endif  /* sun4v */

        CPU_STATS_ADDQ(CPU, sys, trap, 1);

#ifdef SF_ERRATA_23 /* call causes illegal-insn */
        ASSERT((curthread->t_schedflag & TS_DONT_SWAP) ||
            (type == T_UNIMP_INSTR));
```



```
#else
        ASSERT(curthread->t_schedflag & TS_DONT_SWAP);
#endif /* SF_ERRATA_23 */

        if (USERMODE(rp->r_tstate) || (type & T_USER)) {
                /*
                 * Set lwp_state before trying to acquire any
                 * adaptive lock
                 */
                ASSERT(lwp != NULL);
                lwp->lwp_state = LWP_SYS;
                /*
                 * Set up the current cred to use during this trap. u_cred
                 * no longer exists.  t_cred is used instead.
                 * The current process credential applies to the thread for
                 * the entire trap.  If trapping from the kernel, this
                 * should already be set up.
                 */
                if (curthread->t_cred != p->p_cred) {
                        cred_t *oldcred = curthread->t_cred;
                        /*
                         * DTrace accesses t_cred in probe context.  t_cred
                         * must always be either NULL, or point to a valid,
                         * allocated cred structure.
                         */
                        curthread->t_cred = crgetcred();
                        crfree(oldcred);
                }
                type |= T_USER;
                ASSERT((type == (T_SYS_RTT_PAGE | T_USER)) ||
                    (type == (T_SYS_RTT_ALIGN | T_USER)) ||
                    lwp->lwp_regs == rp);
                mpcb = lwptompcb(lwp);
                switch (type) {
                case T_WIN_OVERFLOW + T_USER:
                case T_WIN_UNDERFLOW + T_USER:
                case T_SYS_RTT_PAGE + T_USER:
                case T_DATA_MMU_MISS + T_USER:
                        mstate = LMS_DFAULT;
                        break;
                case T_INSTR_MMU_MISS + T_USER:
                        mstate = LMS_TFAULT;
                        break;
                default:
                        mstate = LMS_TRAP;
                        break;
                }
                /* Kernel probe */
                TNF_PROBE_1(thread_state, "thread", /* CSTYLED */,
                    tnf_microstate, state, (char)mstate);
                mstate = new_mstate(curthread, mstate);
                siginfo.si_signo = 0;
                stepped =
                    lwp->lwp_pcb.pcb_step != STEP_NONE &&
                    ((oldpc = rp->r_pc), prundostep()) &&
                    mmu_btop((uintptr_t)addr) == mmu_btop((uintptr_t)oldpc);
                /* this assignment must not precede call to prundostep() */
                oldpc = rp->r_pc;
        }

        TRACE_1(TR_FAC_TRAP, TR_C_TRAP_HANDLER_ENTER,
            "C_trap_handler_enter:type %x", type);

#ifdef  F_DEFERRED
        /*
         * Take any pending floating point exceptions now.
```

```
233          * If the floating point unit has an exception to handle,
234          * just return to user-level to let the signal handler run.
235          * The instruction that got us to trap() will be reexecuted on
236          * return from the signal handler and we will trap to here again.
237          * This is necessary to disambiguate simultaneous traps which
238          * happen when a floating-point exception is pending and a
239          * machine fault is incurred.
240          */
241         if (type & USER) {
242                 /*
243                  * FP_TRAPPED is set only by sendsig() when it copies
244                  * out the floating-point queue for the signal handler.
245                  * It is set there so we can test it here and in syscall().
246                  */
247                 mpcb->mpcb_flags &= ~FP_TRAPPED;
248                 syncfpu();
249                 if (mpcb->mpcb_flags & FP_TRAPPED) {
250                         /*
251                          * trap() has have been called recursively and may
252                          * have stopped the process, so do single step
253                          * support for /proc.
254                          */
255                         mpcb->mpcb_flags &= ~FP_TRAPPED;
256                         goto out;
257                 }
258         }
259 #endif
260         switch (type) {
261                 case T_DATA_MMU_MISS:
262                 case T_INSTR_MMU_MISS + T_USER:
263                 case T_DATA_MMU_MISS + T_USER:
264                 case T_DATA_PROT + T_USER:
265                 case T_AST + T_USER:
266                 case T_SYS_RTT_PAGE + T_USER:
267                 case T_FLUSH_PCB + T_USER:
268                 case T_FLUSHW + T_USER:
269                         break;

271                 default:
272                         FTRACE_3("trap(): type=0x%lx, regs=0x%lx, addr=0x%lx",
273                             (ulong_t)type, (ulong_t)rp, (ulong_t)addr);
274                         break;
275         }

277         switch (type) {

279         default:
280                 /*
281                  * Check for user software trap.
282                  */
283                 if (type & T_USER) {
284                         if (tudebug)
285                                 showregs(type, rp, (caddr_t)0, 0);
286                         if ((type & ~T_USER) >= T_SOFTWARE_TRAP) {
287                                 bzero(&siginfo, sizeof (siginfo));
288                                 siginfo.si_signo = SIGILL;
289                                 siginfo.si_code  = ILL_ILLTRP;
290                                 siginfo.si_addr  = (caddr_t)rp->r_pc;
291                                 siginfo.si_trapno = type &~ T_USER;
292                                 fault = FLTILL;
293                                 break;
294                         }
295                 }
296                 addr = (caddr_t)rp->r_pc;
297                 (void) die(type, rp, addr, 0);
298                 /*NOTREACHED*/
```



```
300         case T_ALIGNMENT:       /* supv alignment error */
301                 if (nfload(rp, NULL))
302                         goto cleanup;

304                 if (curthread->t_lofault) {
305                         if (lodebug) {
306                                 showregs(type, rp, addr, 0);
307                                 traceback((caddr_t)rp->r_sp);
308                         }
309                         rp->r_g1 = EFAULT;
310                         rp->r_pc = curthread->t_lofault;
311                         rp->r_npc = rp->r_pc + 4;
312                         goto cleanup;
313                 }
314                 (void) die(type, rp, addr, 0);
315                 /*NOTREACHED*/

317         case T_INSTR_EXCEPTION:         /* sys instruction access exception */
318                 addr = (caddr_t)rp->r_pc;
319                 (void) die(type, rp, addr, mmu_fsr);
320                 /*NOTREACHED*/

322         case T_INSTR_MMU_MISS:          /* sys instruction mmu miss */
323                 addr = (caddr_t)rp->r_pc;
324                 (void) die(type, rp, addr, 0);
325                 /*NOTREACHED*/

327         case T_DATA_EXCEPTION:          /* system data access exception */
328                 switch (X_FAULT_TYPE(mmu_fsr)) {
329                 case FT_RANGE:
330                         /*
331                          * This happens when we attempt to dereference an
332                          * address in the address hole.  If t_ontrap is set,
333                          * then break and fall through to T_DATA_MMU_MISS /
334                          * T_DATA_PROT case below.  If lofault is set, then
335                          * honour it (perhaps the user gave us a bogus
336                          * address in the hole to copyin from or copyout to?)
337                          */

339                         if (curthread->t_ontrap != NULL)
340                                 break;

342                         addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
343                         if (curthread->t_lofault) {
344                                 if (lodebug) {
345                                         showregs(type, rp, addr, 0);
346                                         traceback((caddr_t)rp->r_sp);
347                                 }
348                                 rp->r_g1 = EFAULT;
349                                 rp->r_pc = curthread->t_lofault;
350                                 rp->r_npc = rp->r_pc + 4;
351                                 goto cleanup;
352                         }
353                         (void) die(type, rp, addr, mmu_fsr);
354                         /*NOTREACHED*/

356                 case FT_PRIV:
357                         /*
358                          * This can happen if we access ASI_USER from a kernel
359                          * thread.  To support pxfs, we need to honor lofault if
360                          * we're doing a copyin/copyout from a kernel thread.
361                          */

363                         if (nfload(rp, NULL))
364                                 goto cleanup;
```

```
365                         addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
366                         if (curthread->t_lofault) {
367                                 if (lodebug) {
368                                         showregs(type, rp, addr, 0);
369                                         traceback((caddr_t)rp->r_sp);
370                                 }
371                                 rp->r_g1 = EFAULT;
372                                 rp->r_pc = curthread->t_lofault;
373                                 rp->r_npc = rp->r_pc + 4;
374                                 goto cleanup;
375                         }
376                         (void) die(type, rp, addr, mmu_fsr);
377                         /*NOTREACHED*/

379                 default:
380                         if (nfload(rp, NULL))
381                                 goto cleanup;
382                         addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
383                         (void) die(type, rp, addr, mmu_fsr);
384                         /*NOTREACHED*/

386                 case FT_NFO:
387                         break;
388                 }
389                 /* fall into ... */

391         case T_DATA_MMU_MISS:           /* system data mmu miss */
392         case T_DATA_PROT:               /* system data protection fault */
393                 if (nfload(rp, &instr))
394                         goto cleanup;

396                 /*
397                  * If we're under on_trap() protection (see <sys/ontrap.h>),
398                  * set ot_trap and return from the trap to the trampoline.
399                  */
400                 if (curthread->t_ontrap != NULL) {
401                         on_trap_data_t *otp = curthread->t_ontrap;

403                         TRACE_0(TR_FAC_TRAP, TR_C_TRAP_HANDLER_EXIT,
404                             "C_trap_handler_exit");
405                         TRACE_0(TR_FAC_TRAP, TR_TRAP_END, "trap_end");

407                         if (otp->ot_prot & OT_DATA_ACCESS) {
408                                 otp->ot_trap |= OT_DATA_ACCESS;
409                                 rp->r_pc = otp->ot_trampoline;
410                                 rp->r_npc = rp->r_pc + 4;
411                                 goto cleanup;
412                         }
413                 }
414                 lofault = curthread->t_lofault;
415                 onfault = curthread->t_onfault;
416                 curthread->t_lofault = 0;

418                 mstate = new_mstate(curthread, LMS_KFAULT);

420                 switch (type) {
421                 case T_DATA_PROT:
422                         fault_type = F_PROT;
423                         rw = S_WRITE;
424                         break;
425                 case T_INSTR_MMU_MISS:
426                         fault_type = F_INVAL;
427                         rw = S_EXEC;
428                         break;
429                 case T_DATA_MMU_MISS:
430                 case T_DATA_EXCEPTION:
```



```
431                         /*
432                          * The hardware doesn't update the sfsr on mmu
433                          * misses so it is not easy to find out whether
434                          * the access was a read or a write so we need
435                          * to decode the actual instruction.
436                          */
437                         fault_type = F_INVAL;
438                         rw = get_accesstype(rp);
439                         break;
440                 default:
441                         cmn_err(CE_PANIC, "trap: unknown type %x", type);
442                         break;
443                 }
444                 /*
445                  * We determine if access was done to kernel or user
446                  * address space.  The addr passed into trap is really the
447                  * tag access register.
448                  */
449                 iskernel = (((uintptr_t)addr & TAGACC_CTX_MASK) == KCONTEXT);
450                 addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);

452                 res = pagefault(addr, fault_type, rw, iskernel);
453                 if (!iskernel && res == FC_NOMAP &&
454                     addr < p->p_usrstack && grow(addr))
455                         res = 0;

457                 (void) new_mstate(curthread, mstate);

459                 /*
460                  * Restore lofault and onfault.  If we resolved the fault, exit.
461                  * If we didn't and lofault wasn't set, die.
462                  */
463                 curthread->t_lofault = lofault;
464                 curthread->t_onfault = onfault;

466                 if (res == 0)
467                         goto cleanup;

469                 if (IS_PREFETCH(instr)) {
470                         /* skip prefetch instructions in kernel-land */
471                         rp->r_pc = rp->r_npc;
472                         rp->r_npc += 4;
473                         goto cleanup;
474                 }

476                 if ((lofault == 0 || lodebug) &&
477                     (calc_memaddr(rp, &badaddr) == SIMU_SUCCESS))
478                         addr = badaddr;
479                 if (lofault == 0)
480                         (void) die(type, rp, addr, 0);
481                 /*
482                  * Cannot resolve fault.  Return to lofault.
483                  */
484                 if (lodebug) {
485                         showregs(type, rp, addr, 0);
486                         traceback((caddr_t)rp->r_sp);
487                 }
488                 if (FC_CODE(res) == FC_OBJERR)
489                         res = FC_ERRNO(res);
490                 else
491                         res = EFAULT;
492                 rp->r_g1 = res;
493                 rp->r_pc = curthread->t_lofault;
494                 rp->r_npc = curthread->t_lofault + 4;
495                 goto cleanup;
```

```
497         case T_INSTR_EXCEPTION + T_USER: /* user insn access exception */
498                 bzero(&siginfo, sizeof (siginfo));
499                 siginfo.si_addr = (caddr_t)rp->r_pc;
500                 siginfo.si_signo = SIGSEGV;
501                 siginfo.si_code = X_FAULT_TYPE(mmu_fsr) == FT_PRIV ?
502                     SEGV_ACCERR : SEGV_MAPERR;
503                 fault = FLTBOUNDS;
504                 break;

506         case T_WIN_OVERFLOW + T_USER:   /* window overflow in ??? */
507         case T_WIN_UNDERFLOW + T_USER:  /* window underflow in ??? */
508         case T_SYS_RTT_PAGE + T_USER:   /* window underflow in user_rtt */
509         case T_INSTR_MMU_MISS + T_USER: /* user instruction mmu miss */
510         case T_DATA_MMU_MISS + T_USER:  /* user data mmu miss */
511         case T_DATA_PROT + T_USER:      /* user data protection fault */
512                 switch (type) {
513                 case T_INSTR_MMU_MISS + T_USER:
514                         addr = (caddr_t)rp->r_pc;
515                         fault_type = F_INVAL;
516                         rw = S_EXEC;
517                         break;

519                 case T_DATA_MMU_MISS + T_USER:
520                         addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
521                         fault_type = F_INVAL;
522                         /*
523                          * The hardware doesn't update the sfsr on mmu misses
524                          * so it is not easy to find out whether the access
525                          * was a read or a write so we need to decode the
526                          * actual instruction.  XXX BUGLY HW
527                          */
528                         rw = get_accesstype(rp);
529                         break;

531                 case T_DATA_PROT + T_USER:
532                         addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
533                         fault_type = F_PROT;
534                         rw = S_WRITE;
535                         break;

537                 case T_WIN_OVERFLOW + T_USER:
538                         addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
539                         fault_type = F_INVAL;
540                         rw = S_WRITE;
541                         break;

543                 case T_WIN_UNDERFLOW + T_USER:
544                 case T_SYS_RTT_PAGE + T_USER:
545                         addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
546                         fault_type = F_INVAL;
547                         rw = S_READ;
548                         break;

550                 default:
551                         cmn_err(CE_PANIC, "trap: unknown type %x", type);
552                         break;
553                 }

555                 /*
556                  * If we are single stepping do not call pagefault
557                  */
558                 if (stepped) {
559                         res = FC_NOMAP;
560                 } else {
561                         caddr_t vaddr = addr;
562                         size_t sz;
```



```
563                         int ta;

565                         ASSERT(!(curthread->t_flag & T_WATCHPT));
566                         watchpage = (pr_watch_active(p) &&
567                             type != T_WIN_OVERFLOW + T_USER &&
568                             type != T_WIN_UNDERFLOW + T_USER &&
569                             type != T_SYS_RTT_PAGE + T_USER &&
570                             pr_is_watchpage(addr, rw));

572                         if (!watchpage ||
573                             (sz = instr_size(rp, &vaddr, rw)) <= 0)
574                                 /* EMPTY */;
575                         else if ((watchcode = pr_is_watchpoint(&vaddr, &ta,
576                             sz, NULL, rw)) != 0) {
577                                 if (ta) {
578                                         do_watch_step(vaddr, sz, rw,
579                                             watchcode, rp->r_pc);
580                                         fault_type = F_INVAL;
581                                 } else {
582                                         bzero(&siginfo, sizeof (siginfo));
583                                         siginfo.si_signo = SIGTRAP;
584                                         siginfo.si_code = watchcode;
585                                         siginfo.si_addr = vaddr;
586                                         siginfo.si_trapafter = 0;
587                                         siginfo.si_pc = (caddr_t)rp->r_pc;
588                                         fault = FLTWATCH;
589                                         break;
590                                 }
591                         } else {
592                                 if (rw != S_EXEC &&
593                                     pr_watch_emul(rp, vaddr, rw))
594                                         goto out;
595                                 do_watch_step(vaddr, sz, rw, 0, 0);
596                                 fault_type = F_INVAL;
597                         }

599                         if (pr_watch_active(p) &&
600                             (type == T_WIN_OVERFLOW + T_USER ||
601                             type == T_WIN_UNDERFLOW + T_USER ||
602                             type == T_SYS_RTT_PAGE + T_USER)) {
603                                 int dotwo = (type == T_WIN_UNDERFLOW + T_USER);
604                                 if (copy_return_window(dotwo))
605                                         goto out;
606                                 fault_type = F_INVAL;
607                         }

609                         res = pagefault(addr, fault_type, rw, 0);

611                         /*
612                          * If pagefault succeed, ok.
613                          * Otherwise grow the stack automatically.
614                          */
615                         if (res == 0 ||
616                             (res == FC_NOMAP &&
617                             type != T_INSTR_MMU_MISS + T_USER &&
618                             addr < p->p_usrstack &&
619                             grow(addr))) {
620                                 int ismem = prismember(&p->p_fltmask, FLTPAGE);

622                                 /*
623                                  * instr_size() is used to get the exact
624                                  * address of the fault, instead of the
625                                  * page of the fault. Unfortunately it is
626                                  * very slow, and this is an important
627                                  * code path. Don't call it unless
628                                  * correctness is needed. ie. if FLTPAGE
```

```
629                                  * is set, or we're profiling.
630                                  */
631
632                                 if (curthread->t_rprof != NULL || ismem)
633                                         (void) instr_size(rp, &addr, rw);
634
635                                 lwp->lwp_lastfault = FLTPAGE;
636                                 lwp->lwp_lastfaddr = addr;
637
638                                 if (ismem) {
639                                         bzero(&siginfo, sizeof (siginfo));
640                                         siginfo.si_addr = addr;
641                                         (void) stop_on_fault(FLTPAGE, &siginfo);
642                                 }
643                                 goto out;
644                         }
645
646                         if (type != (T_INSTR_MMU_MISS + T_USER)) {
647                                 /*
648                                  * check for non-faulting loads, also
649                                  * fetch the instruction to check for
650                                  * flush
651                                  */
652                                 if (nfload(rp, &instr))
653                                         goto out;
654
655                                 /* skip userland prefetch instructions */
656                                 if (IS_PREFETCH(instr)) {
657                                         rp->r_pc = rp->r_npc;
658                                         rp->r_npc += 4;
659                                         goto out;
660                                         /*NOTREACHED*/
661                                 }
662
663                                 /*
664                                  * check if the instruction was a
665                                  * flush.  ABI allows users to specify
666                                  * an illegal address on the flush
667                                  * instruction so we simply return in
668                                  * this case.
669                                  *
670                                  * NB: the hardware should set a bit
671                                  * indicating this trap was caused by
672                                  * a flush instruction.  Instruction
673                                  * decoding is bugly!
674                                  */
675                                 if (IS_FLUSH(instr)) {
676                                         /* skip the flush instruction */
677                                         rp->r_pc = rp->r_npc;
678                                         rp->r_npc += 4;
679                                         goto out;
680                                         /*NOTREACHED*/
681                                 }
682                         } else if (res == FC_PROT) {
683                                 report_stack_exec(p, addr);
684                         }
685
686                         if (tudebug)
687                                 showregs(type, rp, addr, 0);
688                 }
689
690                 /*
691                  * In the case where both pagefault and grow fail,
692                  * set the code to the value provided by pagefault.
693                  */
694                 (void) instr_size(rp, &addr, rw);
```



```
695                 bzero(&siginfo, sizeof (siginfo));
696                 siginfo.si_addr = addr;
697                 if (FC_CODE(res) == FC_OBJERR) {
698                         siginfo.si_errno = FC_ERRNO(res);
699                         if (siginfo.si_errno != EINTR) {
700                                 siginfo.si_signo = SIGBUS;
701                                 siginfo.si_code = BUS_OBJERR;
702                                 fault = FLTACCESS;
703                         }
704                 } else { /* FC_NOMAP || FC_PROT */
705                         siginfo.si_signo = SIGSEGV;
706                         siginfo.si_code = (res == FC_NOMAP) ?
707                             SEGV_MAPERR : SEGV_ACCERR;
708                         fault = FLTBOUNDS;
709                 }
710                 /*
711                  * If this is the culmination of a single-step,
712                  * reset the addr, code, signal and fault to
713                  * indicate a hardware trace trap.
714                  */
715                 if (stepped) {
716                         pcb_t *pcb = &lwp->lwp_pcb;
717
718                         siginfo.si_signo = 0;
719                         fault = 0;
720                         if (pcb->pcb_step == STEP_WASACTIVE) {
721                                 pcb->pcb_step = STEP_NONE;
722                                 pcb->pcb_tracepc = NULL;
723                                 oldpc = rp->r_pc - 4;
724                         }
725                         /*
726                          * If both NORMAL_STEP and WATCH_STEP are in
727                          * effect, give precedence to WATCH_STEP.
728                          * One or the other must be set at this point.
729                          */
730                         ASSERT(pcb->pcb_flags & (NORMAL_STEP|WATCH_STEP));
731                         if ((fault = undo_watch_step(&siginfo)) == 0 &&
732                             (pcb->pcb_flags & NORMAL_STEP)) {
733                                 siginfo.si_signo = SIGTRAP;
734                                 siginfo.si_code = TRAP_TRACE;
735                                 siginfo.si_addr = (caddr_t)rp->r_pc;
736                                 fault = FLTTRACE;
737                         }
738                         pcb->pcb_flags &= ~(NORMAL_STEP|WATCH_STEP);
739                 }
740                 break;
741
742         case T_DATA_EXCEPTION + T_USER: /* user data access exception */
743
744                 if (&vis1_partial_support != NULL) {
745                         bzero(&siginfo, sizeof (siginfo));
746                         if (vis1_partial_support(rp,
747                             &siginfo, &fault) == 0)
748                                 goto out;
749                 }
750
751                 if (nfload(rp, &instr))
752                         goto out;
753                 if (IS_FLUSH(instr)) {
754                         /* skip the flush instruction */
755                         rp->r_pc = rp->r_npc;
756                         rp->r_npc += 4;
757                         goto out;
758                         /*NOTREACHED*/
759                 }
760                 bzero(&siginfo, sizeof (siginfo));
```

```
761                 siginfo.si_addr = addr;
762                 switch (X_FAULT_TYPE(mmu_fsr)) {
763                 case FT_ATOMIC_NC:
764                         if ((IS_SWAP(instr) && swap_nc(rp, instr)) ||
765                             (IS_LDSTUB(instr) && ldstub_nc(rp, instr))) {
766                                 /* skip the atomic */
767                                 rp->r_pc = rp->r_npc;
768                                 rp->r_npc += 4;
769                                 goto out;
770                         }
771                         /* fall into ... */
772                 case FT_PRIV:
773                         siginfo.si_signo = SIGSEGV;
774                         siginfo.si_code = SEGV_ACCERR;
775                         fault = FLTBOUNDS;
776                         break;
777                 case FT_SPEC_LD:
778                 case FT_ILL_ALT:
779                         siginfo.si_signo = SIGILL;
780                         siginfo.si_code = ILL_ILLADR;
781                         fault = FLTILL;
782                         break;
783                 default:
784                         siginfo.si_signo = SIGSEGV;
785                         siginfo.si_code = SEGV_MAPERR;
786                         fault = FLTBOUNDS;
787                         break;
788                 }
789                 break;

791         case T_SYS_RTT_ALIGN + T_USER:  /* user alignment error */
792         case T_ALIGNMENT + T_USER:      /* user alignment error */
793                 if (tudebug)
794                         showregs(type, rp, addr, 0);
795                 /*
796                  * If the user has to do unaligned references
797                  * the ugly stuff gets done here.
798                  */
799                 alignfaults++;
800                 if (&vis1_partial_support != NULL) {
801                         bzero(&siginfo, sizeof (siginfo));
802                         if (vis1_partial_support(rp,
803                             &siginfo, &fault) == 0)
804                                 goto out;
805                 }

807                 bzero(&siginfo, sizeof (siginfo));
808                 if (type == T_SYS_RTT_ALIGN + T_USER) {
809                         if (nfload(rp, NULL))
810                                 goto out;
811                         /*
812                          * Can't do unaligned stack access
813                          */
814                         siginfo.si_signo = SIGBUS;
815                         siginfo.si_code = BUS_ADRALN;
816                         siginfo.si_addr = addr;
817                         fault = FLTACCESS;
818                         break;
819                 }

821                 /*
822                  * Try to fix alignment before non-faulting load test.
823                  */
824                 if (p->p_fixalignment) {
825                         if (do_unaligned(rp, &badaddr) == SIMU_SUCCESS) {
826                                 rp->r_pc = rp->r_npc;
```



```
827                                 rp->r_npc += 4;
828                                 goto out;
829                         }
830                         if (nfload(rp, NULL))
831                                 goto out;
832                         siginfo.si_signo = SIGSEGV;
833                         siginfo.si_code = SEGV_MAPERR;
834                         siginfo.si_addr = badaddr;
835                         fault = FLTBOUNDS;
836                 } else {
837                         if (nfload(rp, NULL))
838                                 goto out;
839                         siginfo.si_signo = SIGBUS;
840                         siginfo.si_code = BUS_ADRALN;
841                         if (rp->r_pc & 3) {     /* offending address, if pc */
842                                 siginfo.si_addr = (caddr_t)rp->r_pc;
843                         } else {
844                                 if (calc_memaddr(rp, &badaddr) == SIMU_UNALIGN)
845                                         siginfo.si_addr = badaddr;
846                                 else
847                                         siginfo.si_addr = (caddr_t)rp->r_pc;
848                         }
849                         fault = FLTACCESS;
850                 }
851                 break;

853         case T_PRIV_INSTR + T_USER:     /* privileged instruction fault */
854                 if (tudebug)
855                         showregs(type, rp, (caddr_t)0, 0);

857                 bzero(&siginfo, sizeof (siginfo));
858 #ifdef  sun4v
859                 /*
860                  * If this instruction fault is a non-privileged %tick
861                  * or %stick trap, and %tick/%stick user emulation is
862                  * enabled as a result of an OS suspend, then simulate
863                  * the register read. We rely on simulate_rdtick to fail
864                  * if the instruction is not a %tick or %stick read,
865                  * causing us to fall through to the normal privileged
866                  * instruction handling.
867                  */
868                 if (tick_stick_emulation_active &&
869                     (X_FAULT_TYPE(mmu_fsr) == FT_NEW_PRVACT) &&
870                     simulate_rdtick(rp) == SIMU_SUCCESS) {
871                         /* skip the successfully simulated instruction */
872                         rp->r_pc = rp->r_npc;
873                         rp->r_npc += 4;
874                         goto out;
875                 }
876 #endif
877                 siginfo.si_signo = SIGILL;
878                 siginfo.si_code = ILL_PRVOPC;
879                 siginfo.si_addr = (caddr_t)rp->r_pc;
880                 fault = FLTILL;
881                 break;

883         case T_UNIMP_INSTR:             /* priv illegal instruction fault */
884                 if (fpras_implemented) {
885                         /*
886                          * Call fpras_chktrap indicating that
887                          * we've come from a trap handler and pass
888                          * the regs.  That function may choose to panic
889                          * (in which case it won't return) or it may
890                          * determine that a reboot is desired.  In the
891                          * latter case it must alter pc/npc to skip
892                          * the illegal instruction and continue at
```

```
893                          * a controlled address.
894                          */
895                         if (&fpras_chktrap) {
896                                 if (fpras_chktrap(rp))
897                                         goto cleanup;
898                         }
899                 }
900 #if defined(SF_ERRATA_23) || defined(SF_ERRATA_30) /* call ... illegal-insn */
901                 instr = *(int *)rp->r_pc;
902                 if ((instr & 0xc0000000) == 0x40000000) {
903                         long pc;

905                         rp->r_o7 = (long long)rp->r_pc;
906                         pc = rp->r_pc + ((instr & 0x3fffffff) << 2);
907                         rp->r_pc = rp->r_npc;
908                         rp->r_npc = pc;
909                         ill_calls++;
910                         goto cleanup;
911                 }
912 #endif /* SF_ERRATA_23 || SF_ERRATA_30 */
913                 /*
914                  * It's not an fpras failure and it's not SF_ERRATA_23 - die
915                  */
916                 addr = (caddr_t)rp->r_pc;
917                 (void) die(type, rp, addr, 0);
918                 /*NOTREACHED*/

920         case T_UNIMP_INSTR + T_USER:    /* illegal instruction fault */
921 #if defined(SF_ERRATA_23) || defined(SF_ERRATA_30) /* call ... illegal-insn */
922                 instr = fetch_user_instr((caddr_t)rp->r_pc);
923                 if ((instr & 0xc0000000) == 0x40000000) {
924                         long pc;

926                         rp->r_o7 = (long long)rp->r_pc;
927                         pc = rp->r_pc + ((instr & 0x3fffffff) << 2);
928                         rp->r_pc = rp->r_npc;
929                         rp->r_npc = pc;
930                         ill_calls++;
931                         goto out;
932                 }
933 #endif /* SF_ERRATA_23 || SF_ERRATA_30 */
934                 if (tudebug)
935                         showregs(type, rp, (caddr_t)0, 0);
936                 bzero(&siginfo, sizeof (siginfo));
937                 /*
938                  * Try to simulate the instruction.
939                  */
940                 switch (simulate_unimp(rp, &badaddr)) {
941                 case SIMU_RETRY:
942                         goto out;       /* regs are already set up */
943                         /*NOTREACHED*/

945                 case SIMU_SUCCESS:
946                         /* skip the successfully simulated instruction */
947                         rp->r_pc = rp->r_npc;
948                         rp->r_npc += 4;
949                         goto out;
950                         /*NOTREACHED*/

952                 case SIMU_FAULT:
953                         siginfo.si_signo = SIGSEGV;
954                         siginfo.si_code = SEGV_MAPERR;
955                         siginfo.si_addr = badaddr;
956                         fault = FLTBOUNDS;
957                         break;
```



```
959                 case SIMU_DZERO:
960                         siginfo.si_signo = SIGFPE;
961                         siginfo.si_code = FPE_INTDIV;
962                         siginfo.si_addr = (caddr_t)rp->r_pc;
963                         fault = FLTIZDIV;
964                         break;

966                 case SIMU_UNALIGN:
967                         siginfo.si_signo = SIGBUS;
968                         siginfo.si_code = BUS_ADRALN;
969                         siginfo.si_addr = badaddr;
970                         fault = FLTACCESS;
971                         break;

973                 case SIMU_ILLEGAL:
974                 default:
975                         siginfo.si_signo = SIGILL;
976                         op3 = (instr >> 19) & 0x3F;
977                         if ((IS_FLOAT(instr) && (op3 == IOP_V8_STQFA) ||
978                             (op3 == IOP_V8_STDFA)))
979                                 siginfo.si_code = ILL_ILLADR;
980                         else
981                                 siginfo.si_code = ILL_ILLOPC;
982                         siginfo.si_addr = (caddr_t)rp->r_pc;
983                         fault = FLTILL;
984                         break;
985                 }
986                 break;

988         case T_UNIMP_LDD + T_USER:
989         case T_UNIMP_STD + T_USER:
990                 if (tudebug)
991                         showregs(type, rp, (caddr_t)0, 0);
992                 switch (simulate_lddstd(rp, &badaddr)) {
993                 case SIMU_SUCCESS:
994                         /* skip the successfully simulated instruction */
995                         rp->r_pc = rp->r_npc;
996                         rp->r_npc += 4;
997                         goto out;
998                         /*NOTREACHED*/

1000                case SIMU_FAULT:
1001                        if (nfload(rp, NULL))
1002                                goto out;
1003                        siginfo.si_signo = SIGSEGV;
1004                        siginfo.si_code = SEGV_MAPERR;
1005                        siginfo.si_addr = badaddr;
1006                        fault = FLTBOUNDS;
1007                        break;

1009                case SIMU_UNALIGN:
1010                        if (nfload(rp, NULL))
1011                                goto out;
1012                        siginfo.si_signo = SIGBUS;
1013                        siginfo.si_code = BUS_ADRALN;
1014                        siginfo.si_addr = badaddr;
1015                        fault = FLTACCESS;
1016                        break;

1018                case SIMU_ILLEGAL:
1019                default:
1020                        siginfo.si_signo = SIGILL;
1021                        siginfo.si_code = ILL_ILLOPC;
1022                        siginfo.si_addr = (caddr_t)rp->r_pc;
1023                        fault = FLTILL;
1024                        break;
```

```
1025                }
1026                break;

1028        case T_UNIMP_LDD:
1029        case T_UNIMP_STD:
1030                if (simulate_lddstd(rp, &badaddr) == SIMU_SUCCESS) {
1031                        /* skip the successfully simulated instruction */
1032                        rp->r_pc = rp->r_npc;
1033                        rp->r_npc += 4;
1034                        goto cleanup;
1035                        /*NOTREACHED*/
1036                }
1037                /*
1038                 * A third party driver executed an {LDD,STD,LDDA,STDA}
1039                 * that we couldn't simulate.
1040                 */
1041                if (nfload(rp, NULL))
1042                        goto cleanup;

1044                if (curthread->t_lofault) {
1045                        if (lodebug) {
1046                                showregs(type, rp, addr, 0);
1047                                traceback((caddr_t)rp->r_sp);
1048                        }
1049                        rp->r_g1 = EFAULT;
1050                        rp->r_pc = curthread->t_lofault;
1051                        rp->r_npc = rp->r_pc + 4;
1052                        goto cleanup;
1053                }
1054                (void) die(type, rp, addr, 0);
1055                /*NOTREACHED*/

1057        case T_IDIV0 + T_USER:          /* integer divide by zero */
1058        case T_DIV0 + T_USER:           /* integer divide by zero */
1059                if (tudebug && tudebugfpe)
1060                        showregs(type, rp, (caddr_t)0, 0);
1061                bzero(&siginfo, sizeof (siginfo));
1062                siginfo.si_signo = SIGFPE;
1063                siginfo.si_code = FPE_INTDIV;
1064                siginfo.si_addr = (caddr_t)rp->r_pc;
1065                fault = FLTIZDIV;
1066                break;

1068        case T_INT_OVERFLOW + T_USER:   /* integer overflow */
1069                if (tudebug && tudebugfpe)
1070                        showregs(type, rp, (caddr_t)0, 0);
1071                bzero(&siginfo, sizeof (siginfo));
1072                siginfo.si_signo = SIGFPE;
1073                siginfo.si_code  = FPE_INTOVF;
1074                siginfo.si_addr  = (caddr_t)rp->r_pc;
1075                fault = FLTIOVF;
1076                break;

1078        case T_BREAKPOINT + T_USER:     /* breakpoint trap (t 1) */
1079                if (tudebug && tudebugbpt)
1080                        showregs(type, rp, (caddr_t)0, 0);
1081                bzero(&siginfo, sizeof (siginfo));
1082                siginfo.si_signo = SIGTRAP;
1083                siginfo.si_code = TRAP_BRKPT;
1084                siginfo.si_addr = (caddr_t)rp->r_pc;
1085                fault = FLTBPT;
1086                break;

1088        case T_TAG_OVERFLOW + T_USER:   /* tag overflow (taddcctv, tsubcctv) */
1089                if (tudebug)
1090                        showregs(type, rp, (caddr_t)0, 0);
```



```
1091                bzero(&siginfo, sizeof (siginfo));
1092                siginfo.si_signo = SIGEMT;
1093                siginfo.si_code = EMT_TAGOVF;
1094                siginfo.si_addr = (caddr_t)rp->r_pc;
1095                fault = FLTACCESS;
1096                break;

1098        case T_FLUSH_PCB + T_USER:      /* finish user window overflow */
1099        case T_FLUSHW + T_USER:         /* finish user window flush */
1100                /*
1101                 * This trap is entered from sys_rtt in locore.s when,
1102                 * upon return to user is is found that there are user
1103                 * windows in pcb_wbuf.  This happens because they could
1104                 * not be saved on the user stack, either because it
1105                 * wasn't resident or because it was misaligned.
1106                 */
1107        {
1108                int error;
1109                caddr_t sp;

1111                error = flush_user_windows_to_stack(&sp);
1112                /*
1113                 * Possible errors:
1114                 *      error copying out
1115                 *      unaligned stack pointer
1116                 * The first is given to us as the return value
1117                 * from flush_user_windows_to_stack().  The second
1118                 * results in residual windows in the pcb.
1119                 */
1120                if (error != 0) {
1121                        /*
1122                         * EINTR comes from a signal during copyout;
1123                         * we should not post another signal.
1124                         */
1125                        if (error != EINTR) {
1126                                /*
1127                                 * Zap the process with a SIGSEGV - process
1128                                 * may be managing its own stack growth by
1129                                 * taking SIGSEGVs on a different signal stack.
1130                                 */
1131                                bzero(&siginfo, sizeof (siginfo));
1132                                siginfo.si_signo = SIGSEGV;
1133                                siginfo.si_code  = SEGV_MAPERR;
1134                                siginfo.si_addr  = sp;
1135                                fault = FLTBOUNDS;
1136                        }
1137                        break;
1138                } else if (mpcb->mpcb_wbcnt) {
1139                        bzero(&siginfo, sizeof (siginfo));
1140                        siginfo.si_signo = SIGILL;
1141                        siginfo.si_code  = ILL_BADSTK;
1142                        siginfo.si_addr  = (caddr_t)rp->r_pc;
1143                        fault = FLTILL;
1144                        break;
1145                }
1146        }

1148                /*
1149                 * T_FLUSHW is used when handling a ta 0x3 -- the old flush
1150                 * window trap -- which is implemented by executing the
1151                 * flushw instruction. The flushw can trap if any of the
1152                 * stack pages are not writable for whatever reason. In this
1153                 * case only, we advance the pc to the next instruction so
1154                 * that the user thread doesn't needlessly execute the trap
1155                 * again. Normally this wouldn't be a problem -- we'll
1156                 * usually only end up here if this is the first touch to a
```

```
1157                  * stack page -- since the second execution won't trap, but
1158                  * if there's a watchpoint on the stack page the user thread
1159                  * would spin, continuously executing the trap instruction.
1160                  */
1161                 if (type == T_FLUSHW + T_USER) {
1162                         rp->r_pc = rp->r_npc;
1163                         rp->r_npc += 4;
1164                 }
1165                 goto out;

1167         case T_AST + T_USER:            /* profiling or resched pseudo trap */
1168                 if (lwp->lwp_pcb.pcb_flags & CPC_OVERFLOW) {
1169                         lwp->lwp_pcb.pcb_flags &= ~CPC_OVERFLOW;
1170                         if (kcpc_overflow_ast()) {
1171                                 /*
1172                                  * Signal performance counter overflow
1173                                  */
1174                                 if (tudebug)
1175                                         showregs(type, rp, (caddr_t)0, 0);
1176                                 bzero(&siginfo, sizeof (siginfo));
1177                                 siginfo.si_signo = SIGEMT;
1178                                 siginfo.si_code = EMT_CPCOVF;
1179                                 siginfo.si_addr = (caddr_t)rp->r_pc;
1180                                 /* for trap_cleanup(), below */
1181                                 oldpc = rp->r_pc - 4;
1182                                 fault = FLTCPCOVF;
1183                         }
1184                 }

1186                 /*
1187                  * The CPC_OVERFLOW check above may already have populated
1188                  * siginfo and set fault, so the checks below must not
1189                  * touch these and the functions they call must use
1190                  * trapsig() directly.
1191                  */

1193                 if (lwp->lwp_pcb.pcb_flags & ASYNC_HWERR) {
1194                         lwp->lwp_pcb.pcb_flags &= ~ASYNC_HWERR;
1195                         trap_async_hwerr();
1196                 }

1198                 if (lwp->lwp_pcb.pcb_flags & ASYNC_BERR) {
1199                         lwp->lwp_pcb.pcb_flags &= ~ASYNC_BERR;
1200                         trap_async_berr_bto(ASYNC_BERR, rp);
1201                 }

1203                 if (lwp->lwp_pcb.pcb_flags & ASYNC_BTO) {
1204                         lwp->lwp_pcb.pcb_flags &= ~ASYNC_BTO;
1205                         trap_async_berr_bto(ASYNC_BTO, rp);
1206                 }

1208                 break;
1209         }

1211         if (fault) {
1212                 /* We took a fault so abort single step. */
1213                 lwp->lwp_pcb.pcb_flags &= ~(NORMAL_STEP|WATCH_STEP);
1214         }
1215         trap_cleanup(rp, fault, &siginfo, oldpc == rp->r_pc);

1217 out:    /* We can't get here from a system trap */
1218         ASSERT(type & T_USER);
1219         trap_rtt();
1220         (void) new_mstate(curthread, mstate);
1221         /* Kernel probe */
1222         TNF_PROBE_1(thread_state, "thread", /* CSTYLED */,
```



```
1223                 tnf_microstate, state, LMS_USER);

1225         TRACE_0(TR_FAC_TRAP, TR_C_TRAP_HANDLER_EXIT, "C_trap_handler_exit");
1226         return;

1228 cleanup:        /* system traps end up here */
1229         ASSERT(!(type & T_USER));

1231         TRACE_0(TR_FAC_TRAP, TR_C_TRAP_HANDLER_EXIT, "C_trap_handler_exit");
1232 }
______unchanged_portion_omitted_

1343 /*
1344  * Called from fp_traps when a floating point trap occurs.
1345  * Note that the T_DATA_EXCEPTION case does not use X_FAULT_TYPE(mmu_fsr),
1346  * because mmu_fsr (now changed to code) is always 0.
1347  * Note that the T_UNIMP_INSTR case does not call simulate_unimp(),
1348  * because the simulator only simulates multiply and divide instructions,
1349  * which would not cause floating point traps in the first place.
1350  * XXX - Supervisor mode floating point traps?
1351  */
1352 void
1353 fpu_trap(struct regs *rp, caddr_t addr, uint32_t type, uint32_t code)
1354 {
1355         proc_t *p = ttoproc(curthread);
1356         klwp_id_t lwp = ttolwp(curthread);
1357         k_siginfo_t siginfo;
1358         uint_t op3, fault = 0;
1359         int mstate;
1360         char *badaddr;
1361         kfpu_t *fp;
1362         struct fpq *pfpq;
1363         uint32_t inst;
1364         utrap_handler_t *utrapp;

1366         CPU_STATS_ADDQ(CPU, sys, trap, 1);

1375         ASSERT(curthread->t_schedflag & TS_DONT_SWAP);

1368         if (USERMODE(rp->r_tstate)) {
1369                 /*
1370                  * Set lwp_state before trying to acquire any
1371                  * adaptive lock
1372                  */
1373                 ASSERT(lwp != NULL);
1374                 lwp->lwp_state = LWP_SYS;
1375                 /*
1376                  * Set up the current cred to use during this trap. u_cred
1377                  * no longer exists.  t_cred is used instead.
1378                  * The current process credential applies to the thread for
1379                  * the entire trap.  If trapping from the kernel, this
1380                  * should already be set up.
1381                  */
1382                 if (curthread->t_cred != p->p_cred) {
1383                         cred_t *oldcred = curthread->t_cred;
1384                         /*
1385                          * DTrace accesses t_cred in probe context.  t_cred
1386                          * must always be either NULL, or point to a valid,
1387                          * allocated cred structure.
1388                          */
1389                         curthread->t_cred = crgetcred();
1390                         crfree(oldcred);
1391                 }
1392                 ASSERT(lwp->lwp_regs == rp);
1393                 mstate = new_mstate(curthread, LMS_TRAP);
1394                 siginfo.si_signo = 0;
```

```
1395                    type |= T_USER;
1396            }

1398            TRACE_1(TR_FAC_TRAP, TR_C_TRAP_HANDLER_ENTER,
1399                "C_fpu_trap_handler_enter:type %x", type);

1401            if (tudebug && tudebugfpe)
1402                    showregs(type, rp, addr, 0);

1404            bzero(&siginfo, sizeof (siginfo));
1405            siginfo.si_code = code;
1406            siginfo.si_addr = addr;

1408            switch (type) {

1410            case T_FP_EXCEPTION_IEEE + T_USER:      /* FPU arithmetic exception */
1411                    /*
1412                     * FPU arithmetic exception - fake up a fpq if we
1413                     *      came here directly from _fp_ieee_exception,
1414                     *      which is indicated by a zero fpu_qcnt.
1415                     */
1416                    fp = lwptofpu(curthread->t_lwp);
1417                    utrapp = curthread->t_procp->p_utraps;
1418                    if (fp->fpu_qcnt == 0) {
1419                            inst = fetch_user_instr((caddr_t)rp->r_pc);
1420                            lwp->lwp_state = LWP_SYS;
1421                            pfpq = &fp->fpu_q->FQu.fpq;
1422                            pfpq->fpq_addr = (uint32_t *)rp->r_pc;
1423                            pfpq->fpq_instr = inst;
1424                            fp->fpu_qcnt = 1;
1425                            fp->fpu_q_entrysize = sizeof (struct fpq);
1426 #ifdef SF_V9_TABLE_28
1427                            /*
1428                             * Spitfire and blackbird followed the SPARC V9 manual
1429                             * paragraph 3 of section 5.1.7.9 FSR_current_exception
1430                             * (cexc) for setting fsr.cexc bits on underflow and
1431                             * overflow traps when the fsr.tem.inexact bit is set,
1432                             * instead of following Table 28. Bugid 1263234.
1433                             */
1434                            {
1435                                    extern int spitfire_bb_fsr_bug;

1437                                    if (spitfire_bb_fsr_bug &&
1438                                        (fp->fpu_fsr & FSR_TEM_NX)) {
1439                                            if (((fp->fpu_fsr & FSR_TEM_OF) == 0) &&
1440                                                (fp->fpu_fsr & FSR_CEXC_OF)) {
1441                                                    fp->fpu_fsr &= ~FSR_CEXC_OF;
1442                                                    fp->fpu_fsr |= FSR_CEXC_NX;
1443                                                    _fp_write_pfsr(&fp->fpu_fsr);
1444                                                    siginfo.si_code = FPE_FLTRES;
1445                                            }
1446                                            if (((fp->fpu_fsr & FSR_TEM_UF) == 0) &&
1447                                                (fp->fpu_fsr & FSR_CEXC_UF)) {
1448                                                    fp->fpu_fsr &= ~FSR_CEXC_UF;
1449                                                    fp->fpu_fsr |= FSR_CEXC_NX;
1450                                                    _fp_write_pfsr(&fp->fpu_fsr);
1451                                                    siginfo.si_code = FPE_FLTRES;
1452                                            }
1453                                    }
1454                            }
1455 #endif /* SF_V9_TABLE_28 */
1456                            rp->r_pc = rp->r_npc;
1457                            rp->r_npc += 4;
1458                    } else if (utrapp && utrapp[UT_FP_EXCEPTION_IEEE_754]) {
1459                            /*
1460                             * The user had a trap handler installed.  Jump to
```



```
1461                             * the trap handler instead of signalling the process.
1462                             */
1463                            rp->r_pc = (long)utrapp[UT_FP_EXCEPTION_IEEE_754];
1464                            rp->r_npc = rp->r_pc + 4;
1465                            break;
1466                    }
1467                    siginfo.si_signo = SIGFPE;
1468                    fault = FLTFPE;
1469                    break;

1471            case T_DATA_EXCEPTION + T_USER:         /* user data access exception */
1472                    siginfo.si_signo = SIGSEGV;
1473                    fault = FLTBOUNDS;
1474                    break;

1476            case T_LDDF_ALIGN + T_USER: /* 64 bit user lddfa alignment error */
1477            case T_STDF_ALIGN + T_USER: /* 64 bit user stdfa alignment error */
1478                    alignfaults++;
1479                    lwp->lwp_state = LWP_SYS;
1480                    if (&vis1_partial_support != NULL) {
1481                            bzero(&siginfo, sizeof (siginfo));
1482                            if (vis1_partial_support(rp,
1483                                &siginfo, &fault) == 0)
1484                                    goto out;
1485                    }
1486                    if (do_unaligned(rp, &badaddr) == SIMU_SUCCESS) {
1487                            rp->r_pc = rp->r_npc;
1488                            rp->r_npc += 4;
1489                            goto out;
1490                    }
1491                    fp = lwptofpu(curthread->t_lwp);
1492                    fp->fpu_qcnt = 0;
1493                    siginfo.si_signo = SIGSEGV;
1494                    siginfo.si_code = SEGV_MAPERR;
1495                    siginfo.si_addr = badaddr;
1496                    fault = FLTBOUNDS;
1497                    break;

1499            case T_ALIGNMENT + T_USER:              /* user alignment error */
1500                    /*
1501                     * If the user has to do unaligned references
1502                     * the ugly stuff gets done here.
1503                     * Only handles vanilla loads and stores.
1504                     */
1505                    alignfaults++;
1506                    if (p->p_fixalignment) {
1507                            if (do_unaligned(rp, &badaddr) == SIMU_SUCCESS) {
1508                                    rp->r_pc = rp->r_npc;
1509                                    rp->r_npc += 4;
1510                                    goto out;
1511                            }
1512                            siginfo.si_signo = SIGSEGV;
1513                            siginfo.si_code = SEGV_MAPERR;
1514                            siginfo.si_addr = badaddr;
1515                            fault = FLTBOUNDS;
1516                    } else {
1517                            siginfo.si_signo = SIGBUS;
1518                            siginfo.si_code = BUS_ADRALN;
1519                            if (rp->r_pc & 3) {     /* offending address, if pc */
1520                                    siginfo.si_addr = (caddr_t)rp->r_pc;
1521                            } else {
1522                                    if (calc_memaddr(rp, &badaddr) == SIMU_UNALIGN)
1523                                            siginfo.si_addr = badaddr;
1524                                    else
1525                                            siginfo.si_addr = (caddr_t)rp->r_pc;
1526                            }
```

```
1527                        fault = FLTACCESS;
1528                }
1529                break;

1531        case T_UNIMP_INSTR + T_USER:            /* illegal instruction fault */
1532                siginfo.si_signo = SIGILL;
1533                inst = fetch_user_instr((caddr_t)rp->r_pc);
1534                op3 = (inst >> 19) & 0x3F;
1535                if ((op3 == IOP_V8_STQFA) || (op3 == IOP_V8_STDFA))
1536                        siginfo.si_code = ILL_ILLADR;
1537                else
1538                        siginfo.si_code = ILL_ILLTRP;
1539                fault = FLTILL;
1540                break;

1542        default:
1543                (void) die(type, rp, addr, 0);
1544                /*NOTREACHED*/
1545        }

1547        /*
1548         * We can't get here from a system trap
1549         * Never restart any instruction which got here from an fp trap.
1550         */
1551        ASSERT(type & T_USER);

1553        trap_cleanup(rp, fault, &siginfo, 0);
1554 out:
1555        trap_rtt();
1556        (void) new_mstate(curthread, mstate);
1557 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    6586 Fri May  8 18:10:43 2015
new/usr/src/uts/sun4u/Makefile.files
remove xhat
The xhat infrastructure was added to support hardware such as the zulu
graphics card - hardware which had on-board MMUs.  The VM used the xhat code
to keep the CPU's and Zulu's page tables in-sync.  Since the only xhat user
was zulu (which is gone), we can safely remove it simplifying the whole VM
subsystem.
Assorted notes:
- AS_BUSY flag was used solely by xhat
remove zulu (XVR-4000)
XVR-4000 was a very expensive, very rare graphics card.
**********************************************************
#
# 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.
#
#       This Makefile defines all file modules for the directory uts/sun4u
#       and it's children. These are the source files which are sun4u
#       "implementation architecture" dependent.
#

#
#       object lists
#
CORE_OBJS +=    atomic.o
CORE_OBJS +=    bootops.o
CORE_OBJS +=    cmp.o
CORE_OBJS +=    cpc_hwreg.o
CORE_OBJS +=    cpc_subr.o
CORE_OBJS +=    cpupm.o
CORE_OBJS +=    mach_cpu_states.o
CORE_OBJS +=    mach_ddi_impl.o
CORE_OBJS +=    ecc.o
CORE_OBJS +=    fillsysinfo.o
CORE_OBJS +=    forthdebug.o
CORE_OBJS +=    hardclk.o
CORE_OBJS +=    hat_sfmmu.o
CORE_OBJS +=    hat_kdi.o
CORE_OBJS +=    iscsi_boot.o
CORE_OBJS +=    mach_copy.o
CORE_OBJS +=    mach_kpm.o
CORE_OBJS +=    mach_mp_startup.o
CORE_OBJS +=    mach_mp_states.o
```



```
CORE_OBJS +=    mach_sfmmu.o
CORE_OBJS +=    mach_startup.o
CORE_OBJS +=    mach_subr_asm.o
CORE_OBJS +=    mach_trap.o
CORE_OBJS +=    mach_vm_dep.o
CORE_OBJS +=    mach_xc.o
CORE_OBJS +=    mem_cage.o
CORE_OBJS +=    mem_config.o
CORE_OBJS +=    memlist_new.o
CORE_OBJS +=    memscrub.o
CORE_OBJS +=    memscrub_asm.o
CORE_OBJS +=    ppage.o
CORE_OBJS +=    sfmmu_kdi.o
CORE_OBJS +=    swtch.o
CORE_OBJS +=    xhat_sfmmu.o

#
#       Some objects must be linked at the front of the image (or
#       near other objects at the front of the image).
#
SPECIAL_OBJS += trap_table.o
SPECIAL_OBJS += locore.o
SPECIAL_OBJS += mach_locore.o
SPECIAL_OBJS += sfmmu_asm.o
SPECIAL_OBJS += mach_sfmmu_asm.o
SPECIAL_OBJS += interrupt.o
SPECIAL_OBJS += mach_interrupt.o
SPECIAL_OBJS += wbuf.o

#
#                       driver modules
#
ROOTNEX_OBJS    += mach_rootnex.o
UPA64S_OBJS     += upa64s.o
SYSIO_SBUS_OBJS += iommu.o sysioerr.o sysiosbus.o iocache.o
PX_OBJS         += px_asm_4u.o px_err.o px_hlib.o px_lib4u.o px_tools_4u.o
PCI_COMMON_OBJS += pci.o pci_util.o pci_dma.o pci_devctl.o \
                        pci_fdvma.o pci_iommu.o pci_sc.o pci_debug.o \
                        pci_cb.o pci_ib.o pci_ecc.o pci_pbm.o pci_intr.o \
                        pci_space.o pci_counters.o pci_axq.o \
                        pci_fm.o pci_reloc.o pci_tools.o pci_asm.o
RMCLOMV_OBJS    += rmclomv.o

PSYCHO_PCI_OBJS += $(PCI_COMMON_OBJS) pcipsy.o
SCHIZO_PCI_OBJS += $(PCI_COMMON_OBJS) pcisch_asm.o pcisch.o pcix.o
SIMBA_PCI_OBJS  += simba.o
DB21554_OBJS    += db21554.o
US_OBJS         += cpudrv.o cpudrv_mach.o
POWER_OBJS      += power.o
EPIC_OBJS       += epic.o
GRBEEP_OBJS     += grbeep.o
ADM1031_OBJS    += adm1031.o
ICS951601_OBJS  += ics951601.o
PPM_OBJS        += ppm_subr.o ppm.o ppm_plat.o
OPLCFG_OBJS     += opl_cfg.o
PCF8584_OBJS    += pcf8584.o
PCA9556_OBJS    += pca9556.o
ADM1026_OBJS    += adm1026.o
BBC_OBJS        += bbc_beep.o
TDA8444_OBJS    += tda8444.o
MAX1617_OBJS    += max1617.o
SEEPROM_OBJS    += seeprom.o
I2C_SVC_OBJS    += i2c_svc.o
SMBUS_OBJS      += smbus.o
SCHPPM_OBJS     += schppm.o
MC_OBJS         += mc-us3.o mc-us3_asm.o
```

```
118 MC_US3I_OBJS    += mc-us3i.o
119 GPIO_87317_OBJS += gpio_87317.o
120 ISADMA_OBJS     += isadma.o
121 SBBC_OBJS       += sbbc.o
122 LM75_OBJS       += lm75.o
123 LTC1427_OBJS    += ltc1427.o
124 PIC16F747_OBJS  += pic16f747.o
125 PIC16F819_OBJS  += pic16f819.o
126 PCF8574_OBJS    += pcf8574.o
127 PCF8591_OBJS    += pcf8591.o
128 SSC050_OBJS     += ssc050.o
129 SSC100_OBJS     += ssc100.o
130 PMUBUS_OBJS     += pmubus.o
131 PMUGPIO_OBJS    += pmugpio.o
132 PMC_OBJS        += pmc.o
133 TRAPSTAT_OBJS   += trapstat.o
134 I2BSC_OBJS      += i2bsc.o
135 GPTWOCFG_OBJS   += gptwocfg.o
136 GPTWO_CPU_OBJS  += gptwo_cpu.o
138 ZULUVM_OBJS     += zuluvm.o zulu_asm.o zulu_hat.o zulu_hat_asm.o

138 JBUSPPM_OBJS    += jbusppm.o
139 RMC_COMM_OBJS   += rmc_comm.o rmc_comm_crctab.o rmc_comm_dp.o rmc_comm_drvintf.o
140 RMCADM_OBJS     += rmcadm.o
141 MEM_CACHE_OBJS  += mem_cache.o panther_asm.o

143 #
144 #                       kernel cryptographic framework
145 #

147 BIGNUM_PSR_OBJS += mont_mulf_kernel_v9.o

149 AES_OBJS        += aes.o aes_impl.o aes_modes.o aes_crypt_asm.o

151 DES_OBJS        += des_crypt_asm.o

153 ARCFOUR_OBJS    += arcfour.o arcfour_crypt.o arcfour_crypt_asm.o

155 SHA1_OBJS       += sha1_asm.o

157 #
158 #                       tod modules
159 #
160 TODMOSTEK_OBJS  += todmostek.o
161 TODDS1287_OBJS  += todds1287.o
162 TODDS1337_OBJS  += todds1337.o
163 TODSTARFIRE_OBJS += todstarfire.o
164 TODSTARCAT_OBJS += todstarcat.o
165 TODBLADE_OBJS   += todblade.o
166 TODM5819_OBJS   += todm5819.o
167 TODM5819P_RMC_OBJS      += todm5819p_rmc.o
168 TODBQ4802_OBJS  += todbq4802.o
169 TODSG_OBJS      += todsg.o
170 TODOPL_OBJS     = todopl.o

172 #
173 #                       Misc modules
174 #
175 OBPSYM_OBJS     += obpsym.o obpsym_1275.o
176 BOOTDEV_OBJS    += bootdev.o

178 CPR_FIRST_OBJS  = cpr_resume_setup.o
179 CPR_IMPL_OBJS   = cpr_impl.o

181 SBD_OBJS        += sbd.o sbd_cpu.o sbd_mem.o sbd_io.o
```



```
183 PCIE_MISC_OBJS += pci_cfgacc_4u.o pci_cfgacc.o

185 #
186 #                       Brand modules
187 #
188 SN1_BRAND_OBJS  = sn1_brand.o sn1_brand_asm.o
189 S10_BRAND_OBJS  = s10_brand.o s10_brand_asm.o

191 #
192 #                       Performance Counter BackEnd (PCBE) Modules
193 #
194 US_PCBE_OBJS    = us234_pcbe.o
195 OPL_PCBE_OBJS   = opl_pcbe.o

197 #
198 #                       cpu modules
199 #
200 CPU_OBJ         += $(OBJS_DIR)/mach_cpu_module.o
201 SPITFIRE_OBJS   = spitfire.o spitfire_asm.o spitfire_copy.o spitfire_kdi.o commo
202 HUMMINGBIRD_OBJS= $(SPITFIRE_OBJS)
203 US3_CMN_OBJS    = us3_common.o us3_common_mmu.o us3_common_asm.o us3_kdi.o cheet
204 CHEETAH_OBJS    = $(US3_CMN_OBJS) us3_cheetah.o us3_cheetah_asm.o
205 CHEETAHPLUS_OBJS= $(US3_CMN_OBJS) us3_cheetahplus.o us3_cheetahplus_asm.o
206 JALAPENO_OBJS   = $(US3_CMN_OBJS) us3_jalapeno.o us3_jalapeno_asm.o
207 OLYMPUS_OBJS    = opl_olympus.o opl_olympus_asm.o opl_olympus_copy.o \
208                   opl_kdi.o common_asm.o

210 #
211 #                       platform module
212 #
213 PLATMOD_OBJS    = platmod.o

215 #       Section 3:      Misc.
216 #
217 ALL_DEFS        += -Dsun4u
218 INC_PATH        += -I$(UTSBASE)/sun4u

220 #
221 # Since assym.h is a derived file, the dependency must be explicit for
222 # all files including this file. (This is only actually required in the
223 # instance when the .make.state file does not exist.) It may seem that
224 # the lint targets should also have a similar dependency, but they don't
225 # since only C headers are included when #defined(lint) is true.
226 #
227 ASSYM_DEPS      += mach_locore.o
228 ASSYM_DEPS      += module_sfmmu_asm.o
229 ASSYM_DEPS      += spitfire_asm.o spitfire_copy.o
230 ASSYM_DEPS      += cheetah_asm.o cheetah_copy.o
231 ASSYM_DEPS      += mach_subr_asm.o swtch.o
232 ASSYM_DEPS      += mach_interrupt.o mach_xc.o
233 ASSYM_DEPS      += trap_table.o wbuf.o
234 ASSYM_DEPS      += mach_sfmmu_asm.o sfmmu_asm.o memscrub_asm.o
235 ASSYM_DEPS      += mach_copy.o
```

```
**********************************************************
   13383 Fri May  8 18:10:43 2015
new/usr/src/uts/sun4u/Makefile.sun4u
remove zulu (XVR-4000)
XVR-4000 was a very expensive, very rare graphics card.
**********************************************************
#
# 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.
# Copyright (c) 2013 Andrew Stormont.  All rights reserved.
#
#       This makefile contains the common definitions for the sun4u unix
#       and all sun4u implementation architecture dependent modules.
#

#
#       Machine type (implementation architecture):
#
PLATFORM         = sun4u
PROMIF           = ieee1275
PSMBASE          = $(UTSBASE)/../psm

#
#       uname -m value
#
UNAME_M         = $(PLATFORM)

#
# Definitions for the platform-specific /platform directories.
#
# PLATFORMS designates those sun4u machines which have no platform
# specific code.
#
# IMPLEMENTATIONS is used to designate sun4u machines which do have
# platform specific modules (perhaps including their own unix).  All
# code specific to a given implementation resides in the appropriately
# named subdirectory.   This requires these platforms to have their
# own Makefiles to define ROOT_PLAT_DIRS, USR_PLAT_DIRS, etc.
#
# So if we had an implementation named 'foo', we would need the following
# Makefiles in the foo subdirectory:
#
#       sun4u/foo/Makefile
#       sun4u/foo/Makefile.foo
#       sun4u/foo/Makefile.targ
```



```
#

#
# /usr/platform/$(IMPLEMENTED_PLATFORM) is created as a directory that
# all the $(LINKED_PLATFORMS) link to.
#
IMPLEMENTED_PLATFORM    = SUNW,Ultra-2

LINKED_PLATFORMS        += SUNW,Ultra-30
LINKED_PLATFORMS        += SUNW,Ultra-60

#
# all PLATFORMS that do not belong in the $(IMPLEMENTATIONS) list
# ie. all desktop platforms
#
PLATFORMS                = $(IMPLEMENTED_PLATFORM)
PLATFORMS               += $(LINKED_PLATFORMS)

ROOT_PLAT_DIRS          = $(PLATFORMS:%=$(ROOT_PLAT_DIR)/%)
USR_PLAT_DIRS           = $(PLATFORMS:%=$(USR_PLAT_DIR)/%)

USR_DESKTOP_DIR         = $(USR_PLAT_DIR)/$(IMPLEMENTED_PLATFORM)
USR_DESKTOP_INC_DIR     = $(USR_DESKTOP_DIR)/include
USR_DESKTOP_SBIN_DIR    = $(USR_DESKTOP_DIR)/sbin
USR_DESKTOP_LIB_DIR     = $(USR_DESKTOP_DIR)/lib

#
# Welcome to SPARC V9.
#

#
#       Define supported builds
#
DEF_BUILDS      = $(DEF_BUILDS64)
ALL_BUILDS      = $(ALL_BUILDS64)

#
#       Everybody needs to know how to build modstubs.o and to locate unix.o
#
UNIX_DIR         = $(UTSBASE)/$(PLATFORM)/unix
GENLIB_DIR       = $(UTSBASE)/$(PLATFORM)/genunix
MODSTUBS_DIR     = $(UNIX_DIR)
DSF_DIR          = $(UTSBASE)/$(PLATFORM)/genassym
LINTS_DIR        = $(OBJS_DIR)
LINT_LIB_DIR     = $(UTSBASE)/$(PLATFORM)/lint-libs/$(OBJS_DIR)

DTRACESTUBS_O    = $(OBJS_DIR)/dtracestubs.o
DTRACESTUBS      = $(OBJS_DIR)/libdtracestubs.so

UNIX_O           = $(UNIX_DIR)/$(OBJS_DIR)/unix.o
MODSTUBS_O       = $(MODSTUBS_DIR)/$(OBJS_DIR)/modstubs.o
GENLIB           = $(GENLIB_DIR)/$(OBJS_DIR)/libgenunix.so

LINT_LIB         = $(LINT_LIB_DIR)/llib-lunix.ln
GEN_LINT_LIB     = $(LINT_LIB_DIR)/llib-lgenunix.ln

LINT64_DIRS      = $(LINT64_BUILDS:%=$(UTSBASE)/$(PLATFORM)/lint-libs/%)
LINT64_FILES     = $(LINT64_DIRS:%=%/llib-l$(MODULE).ln)

#
#       cpu and platform modules need to know how to build their own symcheck mo
#
PLATMOD          = platmod
PLATLIB          = $(PLAT_DIR)/$(OBJS_DIR)/libplatmod.so

CPUNAME          = cpu
```

```
127 CPULIB           = $(CPU_DIR)/$(OBJS_DIR)/libcpu.so

129 SYM_MOD          = $(OBJS_DIR)/unix.sym

131 #
132 #       Include the makefiles which define build rule templates, the
133 #       collection of files per module, and a few specific flags. Note
134 #       that order is significant, just as with an include path. The
135 #       first build rule template which matches the files name will be
136 #       used. By including these in order from most machine dependent
137 #       to most machine independent, we allow a machine dependent file
138 #       to be used in preference over a machine independent version
139 #       (Such as a machine specific optimization, which preserves the
140 #       interfaces.)
141 #
142 include $(UTSBASE)/sun4/Makefile.files
143 include $(UTSBASE)/$(PLATFORM)/Makefile.files
144 include $(UTSBASE)/sfmmu/Makefile.files
145 include $(UTSBASE)/sparc/v9/Makefile.files
146 include $(UTSBASE)/sparc/Makefile.files
147 include $(UTSBASE)/sun/Makefile.files
148 include $(SRC)/psm/promif/$(PROMIF)/common/Makefile.files
149 include $(SRC)/psm/promif/$(PROMIF)/$(PLATFORM)/Makefile.files
150 include $(UTSBASE)/common/Makefile.files

152 #
153 #       Include machine independent rules. Note that this does not imply
154 #       that the resulting module from rules in Makefile.uts is machine
155 #       independent. Only that the build rules are machine independent.
156 #
157 include $(UTSBASE)/Makefile.uts

159 # These come after Makefile.uts.
160 IMPLEMENTATIONS          = tazmo
161 IMPLEMENTATIONS         += starfire
162 IMPLEMENTATIONS         += javelin
163 IMPLEMENTATIONS         += darwin
164 IMPLEMENTATIONS         += quasar
165 IMPLEMENTATIONS         += grover
166 IMPLEMENTATIONS         += enchilada
167 IMPLEMENTATIONS         += taco
168 IMPLEMENTATIONS         += mpxu
169 IMPLEMENTATIONS         += excalibur
170 IMPLEMENTATIONS         += montecarlo
171 IMPLEMENTATIONS         += serengeti
172 IMPLEMENTATIONS         += littleneck
173 IMPLEMENTATIONS         += starcat
174 IMPLEMENTATIONS         += daktari
175 IMPLEMENTATIONS         += cherrystone
176 IMPLEMENTATIONS         += fjlite
177 IMPLEMENTATIONS         += snowbird
178 IMPLEMENTATIONS         += schumacher
179 IMPLEMENTATIONS         += blade
180 IMPLEMENTATIONS         += boston
181 IMPLEMENTATIONS         += seattle
182 IMPLEMENTATIONS         += chicago
183 IMPLEMENTATIONS         += sunfire
184 IMPLEMENTATIONS         += lw8
185 IMPLEMENTATIONS         += makaha
186 IMPLEMENTATIONS         += opl
187 IMPLEMENTATIONS         += lw2plus

189 #
190 #       machine specific optimization, override default in Makefile.master
191 #
192 CC_XARCH        = -m64 -xarch=sparcvis
```



```
193 AS_XARCH        = -xarch=v9a
194 COPTIMIZE       = -xO3
195 CCMODE          = -Xa

197 CFLAGS          = -xchip=ultra $(CCABS32) $(CCREGSYM)
198 CFLAGS          += $(CC_XARCH)
199 CFLAGS          += $(COPTIMIZE)
200 CFLAGS          += $(EXTRA_CFLAGS)
201 CFLAGS          += $(XAOPT)
202 CFLAGS          += $(INLINES) -D_ASM_INLINES
203 CFLAGS          += $(CCMODE)
204 CFLAGS          += $(SPACEFLAG)
205 CFLAGS          += $(CERRWARN)
206 CFLAGS          += $(CTF_FLAGS_$(CLASS))
207 CFLAGS          += $(C99MODE)
208 CFLAGS          += $(CCUNBOUND)
209 CFLAGS          += $(CCNOAUTOINLINE)
210 CFLAGS          += $(CCSTATICSYM)
211 CFLAGS          += $(CC32BITCALLERS)
212 CFLAGS          += $(IROPTFLAG)
213 CFLAGS          += $(CGLOBALSTATIC)
214 CFLAGS          += -xregs=no%float
215 CFLAGS          += -xstrconst
216 CFLAGS          += $(CSOURCEDEBUGFLAGS)
217 CFLAGS          += $(CUSERFLAGS)

219 ASFLAGS         += $(AS_XARCH)

221 AS_INC_PATH     += -I$(DSF_DIR)/$(OBJS_DIR)

223 LINT_KMODS      += $(GENUNIX_KMODS)

225 LINT_DEFS = -m64

227 #
228 #       The following must be defined for all implementations:
229 #
230 #       MAPFILE:                ld mapfile for the build of kernel/unix.
231 #       MODSTUBS:               Module stubs source file.
232 #       GENCONST_SRC:           genconst.c
233 #       OFFSETS:                offsets.in
234 #       PLATFORM_OFFSETS:       Platform specific mach_offsets.in
235 #       FDOFFSETS:              fd_offsets.in
236 #
237 MAPFILE          = $(UTSBASE)/sun4/conf/Mapfile
238 MODSTUBS         = $(UTSBASE)/sparc/ml/modstubs.s
239 GENCONST_SRC     = $(UTSBASE)/sun4/ml/genconst.c
240 OFFSETS          = $(UTSBASE)/sun4/ml/offsets.in
241 PLATFORM_OFFSETS = $(UTSBASE)/sun4u/ml/mach_offsets.in
242 FDOFFSETS        = $(UTSBASE)/sun/io/fd_offsets.in

244 #
245 #       Define the actual specific platforms
246 #

248 MACHINE_DEFS    = -D$(PLATFORM) -D_MACHDEP -DSFMMU

250 #
251 #       Software workarounds for hardware "features"
252 #

254 include $(UTSBASE)/$(PLATFORM)/Makefile.workarounds

256 #
257 #       Debugging level
258 #
```

```
#       Special knowledge of which special debugging options effect which
#       file is used to optimize the build if these flags are changed.
#
#       XXX: The above could possibly be done for more flags and files, but
#            is left as an experiment to the interested reader. Be forewarned,
#            that excessive use could lead to maintenance difficulties.
#
#       Note: kslice can be enabled for the sun4u, but is disabled by default
#             in all cases.
#

DEBUG_DEFS_OBJ64        =
DEBUG_DEFS_DBG64        = -DDEBUG
DEBUG_DEFS              = $(DEBUG_DEFS_$(BUILD_TYPE))

DEBUG_COND_OBJ64        = $(POUND_SIGN)
DEBUG_COND_DBG64        =
IF_DEBUG_OBJ            = $(DEBUG_COND_$(BUILD_TYPE))$(OBJS_DIR)/

$(IF_DEBUG_OBJ)trap.o           :=      DEBUG_DEFS      += -DTRAPDEBUG
$(IF_DEBUG_OBJ)mach_trap.o      :=      DEBUG_DEFS      += -DTRAPDEBUG
$(IF_DEBUG_OBJ)syscall_trap.o   :=      DEBUG_DEFS      += -DSYSCALLTRACE
$(IF_DEBUG_OBJ)clock.o          :=      DEBUG_DEFS      += -DKSLICE=0

IF_TRAPTRACE_OBJ = $(IF_DEBUG_OBJ)
# comment this out for a non-debug kernel with TRAPTRACE
#IF_TRAPTRACE_OBJ = $(OBJS_DIR)/

$(IF_TRAPTRACE_OBJ)mach_locore.o        :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)mlsetup.o            :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)syscall_trap.o       :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)startup.o            :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)mach_startup.o       :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)mp_startup.o         :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)cpu_states.o         :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)mach_cpu_states.o    :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)interrupt.o          :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)mach_interrupt.o     :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)sfmmu_asm.o          :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)trap_table.o         :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)xc.o                 :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)mach_xc.o            :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)wbuf.o               :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)trap.o               :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)mach_trap.o          :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)x_call.o             :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)spitfire_asm.o       :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)us3_common_asm.o     :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)us3_cheetah_asm.o    :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)us3_cheetahplus_asm.o :=     DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)us3_jalapeno_asm.o   :=      DEBUG_DEFS      += -DTRAPTRACE
$(IF_TRAPTRACE_OBJ)opl_olympus_asm.o    :=      DEBUG_DEFS      += -DTRAPTRACE

# Comment these out if you don’t want dispatcher lock statistics.

#$(IF_DEBUG_OBJ)lock_prim.o     := DEBUG_DEFS   += -DDISP_LOCK_STATS
#$(IF_DEBUG_OBJ)disp.o          := DEBUG_DEFS   += -DDISP_LOCK_STATS

# Comment these out if you don’t want dispatcher debugging

#$(IF_DEBUG_OBJ)lock_prim.o     := DEBUG_DEFS   += -DDISP_DEBUG

#
#       Collect the preprocessor definitions to be associated with *all*
#       files.
#
```



```
ALL_DEFS         = $(MACHINE_DEFS) $(WORKAROUND_DEFS) $(DEBUG_DEFS) \
                   $(OPTION_DEFS)
GENCONST_DEFS    = $(MACHINE_DEFS) $(OPTION_DEFS)

#
# ----- TRANSITIONAL SECTION --------------------------------------------------
#

#
#       Not everything which *should* be a module is a module yet. The
#       following is a list of such objects which are currently part of
#       the base kernel but should soon become kmods.
#
MACH_NOT_YET_KMODS       = $(AUTOCONF_OBJS)

#
# ----- END OF TRANSITIONAL SECTION -------------------------------------------
#

#
#       The kernels modules which are "implementation architecture"
#       specific for this machine are enumerated below. Note that most
#       of these modules must exist (in one form or another) for each
#       architecture.
#
#       Common Drivers (usually pseudo drivers) (/kernel/drv):
#

#
#       Machine Specific Driver Modules (/kernel/drv):
#
#       XXX: How many of these are really machine specific?
#
DRV_KMODS       += bbc_beep
DRV_KMODS       += cpc
DRV_KMODS       += fd
DRV_KMODS       += rootnex sbusmem upa64s zs zsh
DRV_KMODS       += sbus
DRV_KMODS       += pcisch pcipsy simba
DRV_KMODS       += px
DRV_KMODS       += ebus
DRV_KMODS       += su
DRV_KMODS       += tod
DRV_KMODS       += power
DRV_KMODS       += epic
DRV_KMODS       += grbeep
DRV_KMODS       += pcf8584 max1617 seeprom tda8444 pca9556
DRV_KMODS       += ics951601 adm1031
DRV_KMODS       += lm75 ltc1427 pcf8591 pcf8574 ssc050 ssc100
DRV_KMODS       += pic16f819
DRV_KMODS       += pic16f747
DRV_KMODS       += adm1026
DRV_KMODS       += us
DRV_KMODS       += ppm schppm jbusppm
DRV_KMODS       += mc-us3
DRV_KMODS       += mc-us3i
DRV_KMODS       += smbus
DRV_KMODS       += db21554
DRV_KMODS       += gpio_87317
DRV_KMODS       += isadma
DRV_KMODS       += sbbc
DRV_KMODS       += pmubus
DRV_KMODS       += pmugpio
DRV_KMODS       += pmc
DRV_KMODS       += trapstat
DRV_KMODS       += rmc_comm
```

```
391 DRV_KMODS       += rmcadm
392 DRV_KMODS       += rmclomv
393 DRV_KMODS       += sf
394 DRV_KMODS       += nxge
395 DRV_KMODS       += i2bsc
396 DRV_KMODS       += mem_cache

398 #
399 #       Exec Class Modules (/kernel/exec):
400 #
401 EXEC_KMODS      +=

403 #
404 #       Scheduling Class Modules (/kernel/sched):
405 #
406 SCHED_KMODS     +=

408 #
409 #       File System Modules (/kernel/fs):
410 #
411 FS_KMODS        +=

413 #
414 #       Streams Modules (/kernel/strmod):
415 #
416 STRMOD_KMODS    += kb

418 #
419 #       'System' Modules (/kernel/sys):
420 #
421 SYS_KMODS       +=

423 #
424 #       'User' Modules (/kernel/misc):
425 #
426 MISC_KMODS      += bignum
427 MISC_KMODS      += obpsym bootdev vis cpr platmod md5 sha1 i2c_svc
428 MISC_KMODS      += sbd

430 MISC_KMODS      += opl_cfg
431 MISC_KMODS      += zuluvm
431 MISC_KMODS      += gptwo_cpu gptwocfg
432 MISC_KMODS      += pcie

434 #
435 #       Brand modules
436 #
437 BRAND_KMODS     += sn1_brand s10_brand

439 #
440 #       Software Cryptographic Providers (/kernel/crypto):
441 #
442 CRYPTO_KMODS    += aes
443 CRYPTO_KMODS    += arcfour
444 CRYPTO_KMODS    += des

446 #
447 #       generic-unix module (/kernel/genunix):
448 #
449 GENUNIX_KMODS   += genunix

451 #       'User' "Modules" excluded from the Full Kernel lint target:
452 #

454 #
455 #       Modules eXcluded from the product:
```



```
456 #
457 XMODS           +=

459 #
460 #       cpu modules
461 #
462 CPU_KMODS       += cheetah cheetahplus jalapeno serrano spitfire hummingbird

464 #
465 #       sun4u 'TOD' Modules (/platform/.../kernel/tod):
466 #
467 TOD_KMODS       += todds1287 todds1337 todmostek todstarfire
468 TOD_KMODS       += todm5819 todblade todbq4802 todsg todopl
469 TOD_KMODS       += todm5819p_rmc todstarcat

471 #
472 #       Performance Counter BackEnd Modules (/usr/kernel/pcbe):
473 #
474 PCBE_KMODS      += us234_pcbe
475 PCBE_KMODS      += opl_pcbe
```

```
**********************************************************
   12643 Fri May  8 18:10:44 2015
new/usr/src/uts/sun4u/serengeti/sys/sgfrutypes.h
remove zulu (XVR-4000)
XVR-4000 was a very expensive, very rare graphics card.
**********************************************************
/*
 * 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.
 */

#ifndef _SYS_SGFRUTYPES_H
#define _SYS_SGFRUTYPES_H

#ifdef __cplusplus
extern "C" {
#endif

/*
 * sgfrutypes.h - Serengeti/WildCat/Lightweight8 common FRU definitions
 *
 * This header file contains the common FRU-ID definitions and macros for the
 * Serengeti, WildCat and Lightweight8 platforms.
 *
 *      - definitions of the various FRU types.
 *      - macros to generate FRU names.
 *
 * (Not to be confused with the header files for the SGFRU driver)
 */

/*
 * Known HPU/FRU types
 *
 * These FRU definitions are common to both the Serengeti and LightWeight8
 * platforms. They are used by various macros used by both platforms as well
 * as the LW8 specific SGENV (environmentals) driver.
 */
#define SG_HPU_TYPE_SYSTEM_CONTROLLER_BOARD                     (0x101)
#define SG_HPU_TYPE_SYSTEM_CONTROLLER_BOARD_STR  \
                        "System Controller Board"
#define SG_HPU_TYPE_SYSTEM_CONTROLLER_BOARD_ID                  "SSC"
#define SG_HPU_TYPE_SYSTEM_CONTROLLER_BOARD_SHORTNAME           "SSC"

#define SG_HPU_TYPE_SYSTEM_CONTROLLER_BOARD_F3800               (0x102)
#define SG_HPU_TYPE_SYSTEM_CONTROLLER_BOARD_F3800_STR  \
```



```
                        "System Controller Board (F3800)"
#define SG_HPU_TYPE_SYSTEM_CONTROLLER_BOARD_F3800_ID            "SSC"
#define SG_HPU_TYPE_SYSTEM_CONTROLLER_BOARD_F3800_SHORTNAME     "SSC"


#define SG_HPU_TYPE_CPU_BOARD                   (0x201)
#define SG_HPU_TYPE_CPU_BOARD_STR               "CPU Board"
#define SG_HPU_TYPE_CPU_BOARD_ID                "SB"
#define SG_HPU_TYPE_CPU_BOARD_SHORTNAME         "CPU"

#define SG_HPU_TYPE_WIB_BOARD                   (0x202)
#define SG_HPU_TYPE_WIB_BOARD_STR               "WIB Board"
#define SG_HPU_TYPE_WIB_BOARD_ID                "SB"
#define SG_HPU_TYPE_WIB_BOARD_SHORTNAME         "WIB"

#define SG_HPU_TYPE_ZULU_BOARD                  (0x203)
#define SG_HPU_TYPE_ZULU_BOARD_STR              "Zulu Board"
#define SG_HPU_TYPE_ZULU_BOARD_ID               "SB"
#define SG_HPU_TYPE_ZULU_BOARD_SHORTNAME        "GPX"


#define SG_HPU_TYPE_REPEATER_BOARD                      (0x301)
#define SG_HPU_TYPE_REPEATER_BOARD_STR                  "Repeater Board"
#define SG_HPU_TYPE_REPEATER_BOARD_ID                   "RP"

#define SG_HPU_TYPE_LOGIC_ANALYZER_BOARD                (0x302)
#define SG_HPU_TYPE_LOGIC_ANALYZER_BOARD_STR            "Logic Analyzer Board"
#define SG_HPU_TYPE_LOGIC_ANALYZER_BOARD_ID             "RP"

#define SG_HPU_TYPE_REPEATER_BOARD_F3800                (0x303)
#define SG_HPU_TYPE_REPEATER_BOARD_F3800_STR            "Repeater Board (F3800)"
#define SG_HPU_TYPE_REPEATER_BOARD_F3800_ID             "RP"
#define SG_HPU_TYPE_REPEATER_BOARD_F3800_SHORTNAME      "RP"


#define SG_HPU_TYPE_FAN_TRAY_F6800_IO                   (0x401)
#define SG_HPU_TYPE_FAN_TRAY_F6800_IO_STR               "Fan Tray (F6800, I/O)"
#define SG_HPU_TYPE_FAN_TRAY_F6800_IO_ID                "FT"
#define SG_HPU_TYPE_FAN_TRAY_F6800_IO_SHORTNAME         "FAN"

#define SG_HPU_TYPE_FAN_TRAY_F6800_CPU                  (0x402)
#define SG_HPU_TYPE_FAN_TRAY_F6800_CPU_STR              "Fan Tray (F6800, CPU)"
#define SG_HPU_TYPE_FAN_TRAY_F6800_CPU_ID               "FT"
#define SG_HPU_TYPE_FAN_TRAY_F6800_CPU_SHORTNAME        "FAN"

#define SG_HPU_TYPE_FAN_TRAY_RACK                       (0x403)
#define SG_HPU_TYPE_FAN_TRAY_RACK_STR                   "Fan Tray (Rack)"
#define SG_HPU_TYPE_FAN_TRAY_RACK_ID                    "FT"
#define SG_HPU_TYPE_FAN_TRAY_RACK_SHORTNAME             "RACKFAN"

#define SG_HPU_TYPE_FAN_TRAY_F4810                      (0x404)
#define SG_HPU_TYPE_FAN_TRAY_F4810_STR                  "Fan Tray (F4810)"
#define SG_HPU_TYPE_FAN_TRAY_F4810_ID                   "FT"
#define SG_HPU_TYPE_FAN_TRAY_F4810_SHORTNAME            "FAN"

#define SG_HPU_TYPE_FAN_TRAY_F4800_IO                   (0x405)
#define SG_HPU_TYPE_FAN_TRAY_F4800_IO_STR               "Fan Tray (F4800, I/O)"
#define SG_HPU_TYPE_FAN_TRAY_F4800_IO_ID                "FT"
#define SG_HPU_TYPE_FAN_TRAY_F4800_IO_SHORTNAME         "FAN"

#define SG_HPU_TYPE_FAN_TRAY_F4800_CPU                  (0x406)
#define SG_HPU_TYPE_FAN_TRAY_F4800_CPU_STR              "Fan Tray (F4800, CPU)"
#define SG_HPU_TYPE_FAN_TRAY_F4800_CPU_ID               "FT"
#define SG_HPU_TYPE_FAN_TRAY_F4800_CPU_SHORTNAME        "FAN"

#define SG_HPU_TYPE_FAN_TRAY_F4800_TOP_IO               (0x407)
```

```
121 #define SG_HPU_TYPE_FAN_TRAY_F4800_TOP_IO_STR  \
122                         "Fan Tray (F4800, Top I/O)"
123 #define SG_HPU_TYPE_FAN_TRAY_F4800_TOP_IO_ID            "FT"
124 #define SG_HPU_TYPE_FAN_TRAY_F4800_TOP_IO_SHORTNAME     "FAN"

126 #define SG_HPU_TYPE_FAN_TRAY_F3800                      (0x408)
127 #define SG_HPU_TYPE_FAN_TRAY_F3800_STR                  "Fan Tray (F3800)"
128 #define SG_HPU_TYPE_FAN_TRAY_F3800_ID                   "FT"
129 #define SG_HPU_TYPE_FAN_TRAY_F3800_SHORTNAME            "FAN"

131 #define SG_HPU_TYPE_FAN_TRAY_F4800_BOTTOM_IO            (0x409)
132 #define SG_HPU_TYPE_FAN_TRAY_F4800_BOTTOM_IO_STR \
133                         "Fan Tray (F4800, Bottom I/O)"
134 #define SG_HPU_TYPE_FAN_TRAY_F4800_BOTTOM_IO_ID         "FT"
135 #define SG_HPU_TYPE_FAN_TRAY_F4800_BOTTOM_IO_SHORTNAME  "FAN"


138 #define SG_HPU_TYPE_PCI_IO_BOARD                (0x501)
139 #define SG_HPU_TYPE_PCI_IO_BOARD_STR            "PCI I/O Board"
140 #define SG_HPU_TYPE_PCI_IO_BOARD_ID             "IB"
141 #define SG_HPU_TYPE_PCI_IO_BOARD_SHORTNAME      "PCIB"

143 #define SG_HPU_TYPE_CPCI_IO_BOARD               (0x502)
144 #define SG_HPU_TYPE_CPCI_IO_BOARD_STR           "CPCI I/O board"
145 #define SG_HPU_TYPE_CPCI_IO_BOARD_ID            "IB"
146 #define SG_HPU_TYPE_CPCI_IO_BOARD_SHORTNAME     "CPCB"

148 #define SG_HPU_TYPE_CPCI_IO_BOARD_F3800         (0x503)
149 #define SG_HPU_TYPE_CPCI_IO_BOARD_F3800_STR     "CPCI I/O board (F3800)"
150 #define SG_HPU_TYPE_CPCI_IO_BOARD_F3800_ID      "IB"

152 #define SG_HPU_TYPE_WCI_CPCI_IO_BOARD           (0x504)
153 #define SG_HPU_TYPE_WCI_CPCI_IO_BOARD_STR       "WCI cPCI I/O Board"
154 #define SG_HPU_TYPE_WCI_CPCI_IO_BOARD_ID        "IB"

156 #define SG_HPU_TYPE_WCI_CPCI_IO_BOARD_F3800     (0x505)
157 #define SG_HPU_TYPE_WCI_CPCI_IO_BOARD_F3800_STR "WCI cPCI I/O Board (F3800)"
158 #define SG_HPU_TYPE_WCI_CPCI_IO_BOARD_F3800_ID  "IB"


161 #define SG_HPU_TYPE_A123_POWER_SUPPLY           (0x601)
162 #define SG_HPU_TYPE_A123_POWER_SUPPLY_STR       "A123 Power Supply"
163 #define SG_HPU_TYPE_A123_POWER_SUPPLY_ID        "PS"
164 #define SG_HPU_TYPE_A123_POWER_SUPPLY_SHORTNAME "PS"

166 #define SG_HPU_TYPE_A138_POWER_SUPPLY           (0x602)
167 #define SG_HPU_TYPE_A138_POWER_SUPPLY_STR       "A138 Power Supply"
168 #define SG_HPU_TYPE_A138_POWER_SUPPLY_ID        "PS"
169 #define SG_HPU_TYPE_A138_POWER_SUPPLY_SHORTNAME "PS"

171 #define SG_HPU_TYPE_A145_POWER_SUPPLY           (0x603)
172 #define SG_HPU_TYPE_A145_POWER_SUPPLY_STR       "A145 Power Supply"
173 #define SG_HPU_TYPE_A145_POWER_SUPPLY_ID        "PS"
174 #define SG_HPU_TYPE_A145_POWER_SUPPLY_SHORTNAME "PS"

176 #define SG_HPU_TYPE_A152_POWER_SUPPLY           (0x604)
177 #define SG_HPU_TYPE_A152_POWER_SUPPLY_STR       "A152 Power Supply"
178 #define SG_HPU_TYPE_A152_POWER_SUPPLY_ID        "PS"
179 #define SG_HPU_TYPE_A152_POWER_SUPPLY_SHORTNAME "PS"

181 #define SG_HPU_TYPE_A153_POWER_SUPPLY           (0x605)
182 #define SG_HPU_TYPE_A153_POWER_SUPPLY_STR       "A153 Power Supply"
183 #define SG_HPU_TYPE_A153_POWER_SUPPLY_ID        "PS"
184 #define SG_HPU_TYPE_A153_POWER_SUPPLY_SHORTNAME "PS"
```



```
187 #define SG_HPU_TYPE_SUN_FIRE_3800_CENTERPLANE   (0x701) /* 0x701 */
188 #define SG_HPU_TYPE_SUN_FIRE_3800_CENTERPLANE_STR  \
189                         "Sun Fire 3800 Centerplane"
190 #define SG_HPU_TYPE_SUN_FIRE_3800_CENTERPLANE_ID        "ID"
191 #define SG_HPU_TYPE_SUN_FIRE_3800_CENTERPLANE_SHORTNAME "ID"

193 #define SG_HPU_TYPE_SUN_FIRE_6800_CENTERPLANE   (0x702) /* 0x702 */
194 #define SG_HPU_TYPE_SUN_FIRE_6800_CENTERPLANE_STR  \
195                         "Sun Fire 6800 Centerplane"
196 #define SG_HPU_TYPE_SUN_FIRE_6800_CENTERPLANE_ID        "ID"
197 #define SG_HPU_TYPE_SUN_FIRE_6800_CENTERPLANE_SHORTNAME "ID"

199 #define SG_HPU_TYPE_SUN_FIRE_4810_CENTERPLANE   (0x703) /* 0x703 */
200 #define SG_HPU_TYPE_SUN_FIRE_4810_CENTERPLANE_STR  \
201                         "Sun Fire 4810 Centerplane"
202 #define SG_HPU_TYPE_SUN_FIRE_4810_CENTERPLANE_ID        "ID"
203 #define SG_HPU_TYPE_SUN_FIRE_4810_CENTERPLANE_SHORTNAME "ID"

205 #define SG_HPU_TYPE_SUN_FIRE_4800_CENTERPLANE   (0x704) /* 0x704 */
206 #define SG_HPU_TYPE_SUN_FIRE_4800_CENTERPLANE_STR  \
207                         "Sun Fire 4800 Centerplane"
208 #define SG_HPU_TYPE_SUN_FIRE_4800_CENTERPLANE_ID        "ID"
209 #define SG_HPU_TYPE_SUN_FIRE_4800_CENTERPLANE_SHORTNAME "ID"

211 #define SG_HPU_TYPE_SUN_FIRE_3800_REPLACEMENT_CENTERPLANE       (0x705)
212 #define SG_HPU_TYPE_SUN_FIRE_3800_REPLACEMENT_CENTERPLANE_STR  \
213                         "Sun Fire 3800 Replacement Centerplane"
214 #define SG_HPU_TYPE_SUN_FIRE_3800_REPLACEMENT_CENTERPLANE_ID    "ID"
215 #define SG_HPU_TYPE_SUN_FIRE_3800_REPLACEMENT_CENTERPLANE_SHORTNAME     "ID"

217 #define SG_HPU_TYPE_SUN_FIRE_6800_REPLACEMENT_CENTERPLANE       (0x706)
218 #define SG_HPU_TYPE_SUN_FIRE_6800_REPLACEMENT_CENTERPLANE_STR  \
219                         "Sun Fire 6800 Replacement Centerplane"
220 #define SG_HPU_TYPE_SUN_FIRE_6800_REPLACEMENT_CENTERPLANE_ID    "ID"
221 #define SG_HPU_TYPE_SUN_FIRE_6800_REPLACEMENT_CENTERPLANE_SHORTNAME     "ID"

223 #define SG_HPU_TYPE_SUN_FIRE_4810_REPLACEMENT_CENTERPLANE       (0x707)
224 #define SG_HPU_TYPE_SUN_FIRE_4810_REPLACEMENT_CENTERPLANE_STR  \
225                         "Sun Fire 4810 Replacement Centerplane"
226 #define SG_HPU_TYPE_SUN_FIRE_4810_REPLACEMENT_CENTERPLANE_ID            "ID"
227 #define SG_HPU_TYPE_SUN_FIRE_4810_REPLACEMENT_CENTERPLANE_SHORTNAME     "ID"

229 #define SG_HPU_TYPE_SUN_FIRE_4800_REPLACEMENT_CENTERPLANE       (0x708)
230 #define SG_HPU_TYPE_SUN_FIRE_4800_REPLACEMENT_CENTERPLANE_STR  \
231                         "Sun Fire 4800 Replacement Centerplane"
232 #define SG_HPU_TYPE_SUN_FIRE_4800_REPLACEMENT_CENTERPLANE_ID            "ID"
233 #define SG_HPU_TYPE_SUN_FIRE_4800_REPLACEMENT_CENTERPLANE_SHORTNAME     "ID"

235 #define SG_HPU_TYPE_SUN_FIRE_REPLACEMENT_ID_BOARD       (0x709) /* 0x709 */
236 #define SG_HPU_TYPE_SUN_FIRE_REPLACEMENT_ID_BOARD_STR  \
237                         "Sun Fire Replacement ID Board"
238 #define SG_HPU_TYPE_SUN_FIRE_REPLACEMENT_ID_BOARD_ID    "ID"
239 #define SG_HPU_TYPE_SUN_FIRE_REPLACEMENT_ID_BOARD_SHORTNAME     "ID"


242 #define SG_HPU_TYPE_AC_SEQUENCER                (0x900)
243 #define SG_HPU_TYPE_AC_SEQUENCER_STR            "AC Sequencer"
244 #define SG_HPU_TYPE_AC_SEQUENCER_ID             "AC"
245 #define SG_HPU_TYPE_AC_SEQUENCER_SHORTNAME      "AC"


248 #define SG_HPU_TYPE_2MB_ECACHE_MODULE   ((10<<8)|1)     /* 0xA01 */
249 #define SG_HPU_TYPE_2MB_ECACHE_MODULE_STR  \
250                         "2MB Ecache module"

252 #define SG_HPU_TYPE_2MB_ECACHE_MODULE_SHORTNAME "ECACHE"
```

```
254 #define SG_HPU_TYPE_4MB_ECACHE_MODULE   ((10<<8)|2)     /* 0xA02 */
255 #define SG_HPU_TYPE_4MB_ECACHE_MODULE_STR  \
256                         "4MB Ecache module"

258 #define SG_HPU_TYPE_4MB_ECACHE_MODULE_SHORTNAME "ECACHE"

260 #define SG_HPU_TYPE_DRAM_SLOT   ((11<<8)|0)     /* 0xB00 */
261 #define SG_HPU_TYPE_DRAM_SLOT_STR  \
262                         "DRAM slot"

264 #define SG_HPU_TYPE_DRAM_SLOT_SHORTNAME "DIMM"

266 #define SG_HPU_TYPE_128MB_DRAM_MODULE   ((11<<8)|1)     /* 0xB01 */
267 #define SG_HPU_TYPE_128MB_DRAM_MODULE_STR  \
268                         "128MB DRAM module"

270 #define SG_HPU_TYPE_128MB_DRAM_MODULE_SHORTNAME "DIMM"

272 #define SG_HPU_TYPE_256MB_DRAM_MODULE   ((11<<8)|2)     /* 0xB02 */
273 #define SG_HPU_TYPE_256MB_DRAM_MODULE_STR  \
274                         "256MB DRAM module"

276 #define SG_HPU_TYPE_256MB_DRAM_MODULE_SHORTNAME "DIMM"

278 #define SG_HPU_TYPE_512MB_DRAM_MODULE   ((11<<8)|3)     /* 0xB03 */
279 #define SG_HPU_TYPE_512MB_DRAM_MODULE_STR  \
280                         "512MB DRAM module"

282 #define SG_HPU_TYPE_512MB_DRAM_MODULE_SHORTNAME "DIMM"

284 #define SG_HPU_TYPE_1GB_DRAM_MODULE     ((11<<8)|4)     /* 0xB04 */
285 #define SG_HPU_TYPE_1GB_DRAM_MODULE_STR  \
286                         "1GB DRAM module"

288 #define SG_HPU_TYPE_1GB_DRAM_MODULE_SHORTNAME   "DIMM"

290 /*
291  * These macros are used to generate the FRU Names of the various boards etc.
292  * A string is passed in to each macro and by calling a number of the
293  * macros a FRU name in the HLLN format can be built up.
294  *
295  * Note: The string needs to be initialized to an empty string before the
296  *       first of these macros is called to generate a FRU Name.
297  */
298 #define MAX_FRU_NAME_LEN                20

300 #define SG_SET_FRU_NAME_NODE(str, num) \
301 { \
302         char tmp_str[MAX_FRU_NAME_LEN]; \
303         (void) sprintf(tmp_str, "/N%d", num); \
304         (void) strcat(str, tmp_str); \
305 }
```

*unchanged_portion_omitted_*

****
60364 Fri May 8 18:10:44 2015
new/usr/src/uts/sun4u/vm/mach_kpm.c
remove xhat
The xhat infrastructure was added to support hardware such as the zulu graphics card - hardware which had on-board MMUs. The VM used the xhat code to keep the CPU's and Zulu's page tables in-sync. Since the only xhat user was zulu (which is gone), we can safely remove it simplifying the whole VM subsystem.
Assorted notes:
- AS_BUSY flag was used solely by xhat
****

_____unchanged_portion_omitted_

```
1628 /*
1629  * Check/handle potential hme/kpm mapping conflicts
1630  */
1631 static void
1632 sfmmu_kpm_vac_conflict(page_t *pp, caddr_t vaddr)
1633 {
1634         int             vcolor;
1635         struct sf_hment *sfhmep;
1636         struct hat      *tmphat;
1637         struct sf_hment *tmphme = NULL;
1638         struct hme_blk  *hmeblkp;
1639         tte_t           tte;

1641         ASSERT(sfmmu_mlist_held(pp));

1643         if (PP_ISNC(pp))
1644                 return;

1646         vcolor = addr_to_vcolor(vaddr);
1647         if (PP_GET_VCOLOR(pp) == vcolor)
1648                 return;

1650         /*
1651          * There could be no vcolor conflict between a large cached
1652          * hme page and a non alias range kpm page (neither large nor
1653          * small mapped). So if a hme conflict already exists between
1654          * a constituent page of a large hme mapping and a shared small
1655          * conflicting hme mapping, both mappings must be already
1656          * uncached at this point.
1657          */
1658         ASSERT(!PP_ISMAPPED_LARGE(pp));

1660         if (!PP_ISMAPPED(pp)) {
1661                 /*
1662                  * Previous hme user of page had a different color
1663                  * but since there are no current users
1664                  * we just flush the cache and change the color.
1665                  */
1666                 SFMMU_STAT(sf_pgcolor_conflict);
1667                 sfmmu_cache_flush(pp->p_pagenum, PP_GET_VCOLOR(pp));
1668                 PP_SET_VCOLOR(pp, vcolor);
1669                 return;
1670         }

1672         /*
1673          * If we get here we have a vac conflict with a current hme
1674          * mapping. This must have been established by forcing a wrong
1675          * colored mapping, e.g. by using mmap(2) with MAP_FIXED.
1676          */

1678         /*
1679          * Check if any mapping is in same as or if it is locked
```



```
1680          * since in that case we need to uncache.
1681          */
1682         for (sfhmep = pp->p_mapping; sfhmep; sfhmep = tmphme) {
1683                 tmphme = sfhmep->hme_next;
1684                 if (IS_PAHME(sfhmep))
1685                         continue;
1686                 hmeblkp = sfmmu_hmetohblk(sfhmep);
1687                 if (hmeblkp->hblk_xhat_bit)
1688                         continue;
1687                 tmphat = hblktosfmmu(hmeblkp);
1688                 sfmmu_copytte(&sfhmep->hme_tte, &tte);
1689                 ASSERT(TTE_IS_VALID(&tte));
1690                 if ((tmphat == ksfmmup) || hmeblkp->hblk_lckcnt) {
1691                         /*
1692                          * We have an uncache conflict
1693                          */
1694                         SFMMU_STAT(sf_uncache_conflict);
1695                         sfmmu_page_cache_array(pp, HAT_TMPNC, CACHE_FLUSH, 1);
1696                         return;
1697                 }
1698         }

1700         /*
1701          * We have an unload conflict
1702          */
1703         SFMMU_STAT(sf_unload_conflict);

1705         for (sfhmep = pp->p_mapping; sfhmep; sfhmep = tmphme) {
1706                 tmphme = sfhmep->hme_next;
1707                 if (IS_PAHME(sfhmep))
1708                         continue;
1709                 hmeblkp = sfmmu_hmetohblk(sfhmep);
1712                 if (hmeblkp->hblk_xhat_bit)
1713                         continue;
1710                 (void) sfmmu_pageunload(pp, sfhmep, TTE8K);
1711         }

1713         /*
1714          * Unloads only does tlb flushes so we need to flush the
1715          * dcache vcolor here.
1716          */
1717         sfmmu_cache_flush(pp->p_pagenum, PP_GET_VCOLOR(pp));
1718         PP_SET_VCOLOR(pp, vcolor);
1719 }
```

_____unchanged_portion_omitted_

```
**********************************************************
    6371 Fri May  8 18:10:44 2015
new/usr/src/uts/sun4v/Makefile.files
remove xhat
The xhat infrastructure was added to support hardware such as the zulu
graphics card - hardware which had on-board MMUs.  The VM used the xhat code
to keep the CPU's and Zulu's page tables in-sync.  Since the only xhat user
was zulu (which is gone), we can safely remove it simplifying the whole VM
subsystem.
Assorted notes:
- AS_BUSY flag was used solely by xhat
**********************************************************
#
# 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) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
#
#       This Makefile defines all file modules for the directory uts/sun4v
#       and it's children. These are the source files which are sun4v
#       "implementation architecture" dependent.
#


#
#       object lists
#
CORE_OBJS +=    bootops.o
CORE_OBJS +=    cmp.o
CORE_OBJS +=    cpc_hwreg.o
CORE_OBJS +=    cpc_subr.o
CORE_OBJS +=    error.o
CORE_OBJS +=    fillsysinfo.o
CORE_OBJS +=    forthdebug.o
CORE_OBJS +=    hardclk.o
CORE_OBJS +=    hat_sfmmu.o
CORE_OBJS +=    hat_kdi.o
CORE_OBJS +=    hsvc.o
CORE_OBJS +=    iscsi_boot.o
CORE_OBJS +=    kldc.o
CORE_OBJS +=    lpad.o
CORE_OBJS +=    mach_cpu_states.o
CORE_OBJS +=    mach_ddi_impl.o
CORE_OBJS +=    mach_descrip.o
CORE_OBJS +=    mach_kpm.o
CORE_OBJS +=    mach_mp_startup.o
CORE_OBJS +=    mach_mp_states.o
CORE_OBJS +=    mach_proc_init.o
CORE_OBJS +=    mach_sfmmu.o
```



```
CORE_OBJS +=    mach_startup.o
CORE_OBJS +=    mach_subr_asm.o
CORE_OBJS +=    mach_trap.o
CORE_OBJS +=    mach_vm_dep.o
CORE_OBJS +=    mach_xc.o
CORE_OBJS +=    mem_cage.o
CORE_OBJS +=    mem_config.o
CORE_OBJS +=    memlist_new.o
CORE_OBJS +=    memseg.o
CORE_OBJS +=    mpo.o
CORE_OBJS +=    ppage.o
CORE_OBJS +=    promif_asr.o
CORE_OBJS +=    promif_cpu.o
CORE_OBJS +=    promif_emul.o
CORE_OBJS +=    promif_mon.o
CORE_OBJS +=    promif_io.o
CORE_OBJS +=    promif_interp.o
CORE_OBJS +=    promif_key.o
CORE_OBJS +=    promif_power_off.o
CORE_OBJS +=    promif_prop.o
CORE_OBJS +=    promif_node.o
CORE_OBJS +=    promif_reboot.o
CORE_OBJS +=    promif_stree.o
CORE_OBJS +=    promif_test.o
CORE_OBJS +=    promif_version.o
CORE_OBJS +=    sfmmu_kdi.o
CORE_OBJS +=    suspend.o
CORE_OBJS +=    swtch.o
CORE_OBJS +=    wdt.o
CORE_OBJS +=    xhat_sfmmu.o

CORE_OBJS +=    mdesc_diff.o
CORE_OBJS +=    mdesc_findname.o
CORE_OBJS +=    mdesc_findnodeprop.o
CORE_OBJS +=    mdesc_fini.o
CORE_OBJS +=    mdesc_getbinsize.o
CORE_OBJS +=    mdesc_getgen.o
CORE_OBJS +=    mdesc_getpropdata.o
CORE_OBJS +=    mdesc_getpropstr.o
CORE_OBJS +=    mdesc_getpropval.o
CORE_OBJS +=    mdesc_init_intern.o
CORE_OBJS +=    mdesc_nodecount.o
CORE_OBJS +=    mdesc_rootnode.o
CORE_OBJS +=    mdesc_scandag.o

#
#       Some objects must be linked at the front of the image (or
#       near other objects at the front of the image).
#
SPECIAL_OBJS += trap_table.o
SPECIAL_OBJS += locore.o
SPECIAL_OBJS += mach_locore.o
SPECIAL_OBJS += sfmmu_asm.o
SPECIAL_OBJS += mach_sfmmu_asm.o
SPECIAL_OBJS += interrupt.o
SPECIAL_OBJS += mach_interrupt.o
SPECIAL_OBJS += wbuf.o
SPECIAL_OBJS += hcall.o
SPECIAL_OBJS += intrq.o

#
#                       driver modules
#
ROOTNEX_OBJS    += mach_rootnex.o
PX_OBJS         += px_lib4v.o px_err.o px_tools_4v.o px_hcall.o px_libhv.o
FPC_OBJS        += fpc-impl-4v.o fpc-asm-4v.o
```

```
120 N2PIUPC_OBJS    += n2piupc.o n2piupc_tables.o n2piupc_kstats.o \
121                        n2piupc_biterr.o n2piupc_asm.o
122 IOSPC_OBJS      += iospc.o rfios_iospc.o rfios_tables.o rfios_asm.o
123 TRAPSTAT_OBJS   += trapstat.o
124 NIUMX_OBJS      += niumx.o niumx_tools.o
125 N2RNG_OBJS      += n2rng.o n2rng_debug.o n2rng_hcall.o n2rng_kcf.o \
126                    n2rng_entp_algs.o n2rng_entp_setup.o n2rng_kstat.o \
127                    n2rng_provider.o

129 #
130 #                       CPU/Memory Error Injector (memtest) sun4v driver
131 #
132 MEMTEST_OBJS    += memtest.o memtest_asm.o \
133                        memtest_v.o memtest_v_asm.o \
134                        memtest_kt.o memtest_kt_asm.o \
135                        memtest_ni.o memtest_ni_asm.o \
136                        memtest_n2.o memtest_n2_asm.o \
137                        memtest_vf.o

139 #
140 #                       sun4v virtual devices
141 #
142 QCN_OBJS        = qcn.o
143 VNEX_OBJS       = vnex.o
144 CNEX_OBJS       = cnex.o
145 GLVC_OBJS       = glvc.o glvc_hcall.o
146 MDESC_OBJS      = mdesc.o
147 LDC_OBJS        = ldc.o ldc_shm.o vio_util.o vdsk_common.o vgen_stats.o \
148                        vnet_common.o
149 NTWDT_OBJS      = ntwdt.o
150 VLDC_OBJS       = vldc.o
151 VCC_OBJS        = vcc.o
152 VNET_OBJS       = vnet.o vnet_gen.o vnet_dds.o vnet_dds_hcall.o         \
153                        vnet_txdring.o  vnet_rxdring.o
154 VSW_OBJS        = vsw.o vsw_ldc.o vsw_phys.o vsw_switching.o vsw_hio.o  \
155                        vsw_txdring.o   vsw_rxdring.o
156 VDC_OBJS        = vdc.o
157 VDS_OBJS        = vds.o
158 DS_PRI_OBJS     = ds_pri.o ds_pri_hcall.o
159 DS_SNMP_OBJS    = ds_snmp.o
160 VLDS_OBJS       = vlds.o

162 #
163 #                       Misc modules
164 #
165 BOOTDEV_OBJS    += bootdev.o
166 DR_CPU_OBJS     += dr_cpu.o
167 DR_IO_OBJS      += dr_io.o
168 DR_MEM_OBJS     += dr_mem.o
169 DRCTL_OBJS      = drctl.o drctl_impl.o dr_util.o
170 DS_OBJS         = ds_common.o ds_drv.o
171 FAULT_ISO_OBJS  = fault_iso.o
172 OBPSYM_OBJS     += obpsym.o obpsym_1275.o
173 PLATSVC_OBJS    = platsvc.o mdeg.o
174 PCIE_MISC_OBJS += pci_cfgacc_4v.o pci_cfgacc_asm.o pci_cfgacc.o

176 #
177 #                       Brand modules
178 #
179 SN1_BRAND_OBJS          = sn1_brand.o sn1_brand_asm.o
180 S10_BRAND_OBJS          = s10_brand.o s10_brand_asm.o

182 #
183 #                       Performance Counter BackEnd (PCBE) Modules
184 #
185 NI_PCBE_OBJS    = niagara_pcbe.o
```



```
186 N2_PCBE_OBJS    = niagara2_pcbe.o

188 #
189 #                       cpu modules
190 #
191 CPU_OBJ         += $(OBJS_DIR)/mach_cpu_module.o
192 GENERIC_OBJS = generic.o generic_copy.o common_asm.o atomic.o
193 NIAGARACPU_OBJS = niagara.o niagara_copy.o common_asm.o niagara_perfctr.o
194 NIAGARACPU_OBJS += niagara_asm.o atomic.o
195 NIAGARA2CPU_OBJS = niagara2.o niagara_copy.o common_asm.o niagara_perfctr.o
196 NIAGARA2CPU_OBJS += niagara2_asm.o atomic.o

198 #
199 #                       platform module
200 #
201 PLATMOD_OBJS    = platmod.o

203 #       Section 3:      Misc.
204 #
205 ALL_DEFS        += -Dsun4u -Dsun4v
206 INC_PATH        += -I$(UTSBASE)/sun4v
207 #
208 # Since assym.h is a derived file, the dependency must be explicit for
209 # all files including this file. (This is only actually required in the
210 # instance when the .make.state file does not exist.) It may seem that
211 # the lint targets should also have a similar dependency, but they don't
212 # since only C headers are included when #defined(lint) is true.
213 #
214 ASSYM_DEPS      += mach_locore.o
215 ASSYM_DEPS      += module_sfmmu_asm.o
216 ASSYM_DEPS      += generic_asm.o generic_copy.o
217 ASSYM_DEPS      += niagara_copy.o niagara_asm.o niagara2_asm.o
218 ASSYM_DEPS      += mach_subr_asm.o swtch.o
219 ASSYM_DEPS      += mach_interrupt.o mach_xc.o
220 ASSYM_DEPS      += trap_table.o wbuf.o
221 ASSYM_DEPS      += mach_sfmmu_asm.o sfmmu_asm.o

223 #
224 #                       kernel cryptographic framework
225 #

227 ARCFOUR_OBJS    += arcfour.o arcfour_crypt.o
```