

new/usr/src/uts/common/Makefile.files

1

\*\*\*\*\*

44750 Fri May 8 18:02:53 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

\*\*\*\*\*

```
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 #
23 # Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
24 # Copyright (c) 2013 by Delphix. All rights reserved.
25 # Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
26 # Copyright 2014 Nexenta Systems, Inc. All rights reserved.
27 #
28 #
29 #
30 # This Makefile defines all file modules for the directory uts/common
31 # and its children. These are the source files which may be considered
32 # common to all SunOS systems.
33 #
34 i386_CORE_OBJS += \
35     atomic.o      \
36     avintr.o      \
37     pic.o
38 #
39 sparc_CORE_OBJS +=
40 #
41 COMMON_CORE_OBJS += \
42     beep.o        \
43     bitset.o      \
44     bp_map.o      \
45     brand.o       \
46     cpucaps.o     \
47     cmt.o         \
48     cmt_policy.o  \
49     cpu.o         \
50     cpu_event.o   \
51     cpu_intr.o    \
52     cpu_pm.o      \
53     cpupart.o     \
54     cap_util.o    \
```

new/usr/src/uts/common/Makefile.files

2

```
55     disp.o        \
56     group.o       \
57     kstat_fr.o    \
58     iscsiboot_prop.o \
59     lgrp.o        \
60     lgrp_topo.o   \
61     mmapobj.o     \
62     mutex.o       \
63     page_lock.o   \
64     page_retire.o \
65     panic.o       \
66     param.o       \
67     pg.o          \
68     pghw.o        \
69     putnext.o     \
70     rctl_proc.o   \
71     rwlock.o      \
72     seg_kmem.o    \
73     softint.o     \
74     string.o      \
75     strtol.o      \
76     strtoul.o     \
77     strtoll.o     \
78     strtoull.o   \
79     thread_intr.o \
80     vm_page.o     \
81     vm_pagelist.o \
82     zlib_obj.o    \
83     clock_tick.o
84 #
85 CORE_OBJS += $(COMMON_CORE_OBJS) $(MACH)_CORE_OBJS
86 #
87 ZLIB_OBJS = zutil.o zmod.o zmod_subr.o \
88     adler32.o crc32.o deflate.o inffast.o \
89     inflate.o inftrees.o trees.o
90 #
91 GENUNIX_OBJS += \
92     access.o      \
93     acl.o         \
94     acl_common.o \
95     adjtime.o     \
96     alarm.o       \
97     aio_subr.o    \
98     auditsys.o    \
99     audit_core.o  \
100    audit_zone.o   \
101    audit_memory.o \
102    autoconf.o     \
103    avl.o          \
104    bdev_dsort.o   \
105    bio.o          \
106    bitmap.o       \
107    blabel.o       \
108    brandsys.o     \
109    bz2blocksort.o \
110    bz2compress.o  \
111    bz2decompress.o \
112    bz2randtable.o \
113    bz2bzlib.o     \
114    bz2crctable.o \
115    bz2huffman.o   \
116    callb.o        \
117    callout.o      \
118    chdir.o        \
119    chmod.o        \
120    chown.o        \
```

## new/usr/src/uts/common/Makefile.files

```

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          exact.o          \
168          exact_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         \

```

3

## new/usr/src/uts/common/Makefile.files

```

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          ipppconf.o       \
205          kpcp.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       \

```

4

## new/usr/src/uts/common/Makefile.files

```

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 \

```

5

## new/usr/src/uts/common/Makefile.files

```

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 \

```

6

## new/usr/src/uts/common/Makefile.files

7

```

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     audiofltdata.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

```

## new/usr/src/uts/common/Makefile.files

8

```

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_mrout.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_queue.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 IPSECEP_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 sockscctp.o sockscctpsubr.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 +=      gentyty.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

```

## new/usr/src/uts/common/Makefile.files

11

```

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_srqp.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_srqp.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_srqp.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

```

## new/usr/src/uts/common/Makefile.files

12

```

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_v412.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 USBS49_PRL_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 +=      hcidi.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_uugen.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_async.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 +=      hcil1394.o hcil1394_async.o hcil1394_attach.o hcil1394_buf.o \
836                  hcil1394_csr.o hcil1394_detach.o hcil1394_extern.o \
837                  hcil1394_ioctl.o hcil1394_isoch.o hcil1394_isr.o \
838                  hcil1394_ixl_comp.o hcil1394_ixl_isr.o hcil1394_ixl_misc.o \
839                  hcil1394_ixl_update.o hcil1394_misc.o hcil1394_ohci.o \
840                  hcil1394_q.o hcil1394_s1394if.o hcil1394_tlabel.o \
841                  hcil1394_tlist.o hcil1394_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

```

## new/usr/src/uts/common/Makefile.files

15

```

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_eis.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 SYMSG_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 \

```

## new/usr/src/uts/common/Makefile.files

16

```

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 HPCSVCS_OBJS +=   hpcsvc.o

1013 PCIE_MISC_OBJS += pcie.o pcie_fault.o pcie_hp.o pciehp.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 \

```



## new/usr/src/uts/common/Makefile.files

17

```

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 AUTOFNS_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 \

```

## new/usr/src/uts/common/Makefile.files

18

```

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 \

```

## new/usr/src/uts/common/Makefile.files

```

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_marshallng.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

```

19

## new/usr/src/uts/common/Makefile.files

```

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_vnops.o

1252 PROC_OBJS += prcontrol.o prioctl.o prsubr.o prusr.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_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

```

20

## new/usr/src/uts/common/Makefile.files

21

```

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     dmuf.o \
1332     dmuf_diff.o \
1333     dmuf_send.o \
1334     dmuf_object.o \
1335     dmuf_objset.o \
1336     dmuf_traverse.o \
1337     dmuf_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     dmuf_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 \

```

## new/usr/src/uts/common/Makefile.files

22

```

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 PKCT_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     xdrdma_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 += rlmmod.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 mprime.o oid.o \
1570             secitem.o ec2_test.o ecp_test.o

```

## new/usr/src/uts/common/Makefile.files

25

```

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 ksslioct1.o

1581 KSSL_SOCKETFIL_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

```

## new/usr/src/uts/common/Makefile.files

26

```

1638 # crypto/hash_provider
1639 CRYPTO_HASH= hash_kef_generic.o hash_kmd5.o hash_crc32.o hash_kshal.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 += exaccts.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 uhcigt.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_cmnm.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 DLPSTUB_OBJS += dlpstub.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_eeeprom.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 += iw2.o
1809 IWP_OBJS += iwp.o
1811 MWL_OBJS += mw1.o
1813 MWLFW_OBJS += mw1fw_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_lle.o atge_mii.o atge_ll.o atge_llc.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 assembler
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 csapi.o espi.o ixfl1010.o mc3.o mc4.o mc5.o \
1881     mv88elxxx.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_nvmm.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 \

```

## new/usr/src/uts/common/Makefile.files

31

```

1968          napi_txdma.o napi_rxdma.o napi_txc.o          \
1969          napi_zcp.o napi_espc.o napi_fflp.o            \
1970          napi_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 #          sdcards 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_talk2ctrl.o \
2033          cpqary3_isr.o cpqary3_transport.o cpqary3_mem.o \

```

## new/usr/src/uts/common/Makefile.files

32

```

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 = myril0ge.o myril0ge_lro.o

2061 #
2062 #          nulldriver module
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          \

```



new/usr/src/uts/common/Makefile.files

33

```
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          \/
```

```

*****
38530 Fri May 8 18:02:54 2015
new/usr/src/uts/common/os/watchpoint.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_____

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.
186      * space lock leads to deadlocks with the clock thread. Note that if an
187      * as_fault() is servicing a fault to a watched page on behalf of an
188      * XHAT provider, watchpoint will be temporarily cleared (and wp_prot
189      * will be set to wp_oprot). Since this is done while holding as writer
190      * lock, we need to grab as lock (reader lock is good enough).
191      *
192      * p_maplock prevents simultaneous execution of this function. Under
193      * normal circumstances, holdwatch() will stop all other threads, so the
194      * lock isn't really needed. But there may be multiple threads within
195      * stop() when SWATCHOK is set, so we need to handle multiple threads
196      * at once. See holdwatch() for the details of this dance.
197      */

197     mutex_enter(&p->p_maplock);
198     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)) {
203
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         /*
214          * If the requested access does not exist in the page's
215          * original protections, we need not remap this page.
216          * If the page does not exist yet, we can't test it.
217          */
218         if ((prot = pwp->wp_oprot) != 0) {
219             if (!(kernel || (prot & PROT_USER)))
220                 continue;
221             if (!(prot & prot_rw))
222                 continue;
223         }
224
225         if (mapin) {
226             /*
227              * Before mapping the page in, ensure that
228              * all other lwps are held in the kernel.
229              */
230             if (p->p_mapcnt == 0) {
231                 /*
232                  * Release as lock while in holdwatch()
233                  * in case other threads need to grab it.
234                  */
235                 AS_LOCK_EXIT(as, &as->a_lock);
236                 mutex_exit(&p->p_maplock);
237                 if (holdwatch() != 0) {
238                     /*
239                      * We stopped in holdwatch().
240                      * Start all over again because the
241                      * watched page list may have changed.
242                      */
243                     goto startover;
244                 }
245                 mutex_enter(&p->p_maplock);
246                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
247             }
248             p->p_mapcnt++;
249         }
250
251         addr = pwp->wp_vaddr;
252         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;
265             if (pwp->wp_kmap[W] + pwp->wp_umap[W])

```

```

266             /* cannot have write-only protection */
267             prot |= PROT_READ|PROT_WRITE;
268 #if 0      /* damned broken mmu feature! */
269             if (sum(pwp->wp_umap) == 0)
270                 prot &= ~PROT_USER;
271 #endif
272         } else {
273             ASSERT(pwp->wp_flags & WP_NOWATCH);
274             if (kernel) {
275                 ASSERT(pwp->wp_kmap[xrw] != 0);
276                 --pwp->wp_kmap[xrw];
277             } else {
278                 ASSERT(pwp->wp_umap[xrw] != 0);
279                 --pwp->wp_umap[xrw];
280             }
281             if (sum(pwp->wp_kmap) + sum(pwp->wp_umap) == 0)
282                 pwp->wp_flags &= ~WP_NOWATCH;
283             else {
284                 if (pwp->wp_kmap[X] + pwp->wp_umap[X])
285                     /* cannot have exec-only protection */
286                     prot |= PROT_READ|PROT_EXEC;
287                 if (pwp->wp_kmap[R] + pwp->wp_umap[R])
288                     prot |= PROT_READ;
289                 if (pwp->wp_kmap[W] + pwp->wp_umap[W])
290                     /* cannot have write-only protection */
291                     prot |= PROT_READ|PROT_WRITE;
292 #if 0      /* damned broken mmu feature! */
293                 if (sum(pwp->wp_umap) == 0)
294                     prot &= ~PROT_USER;
295 #endif
296             }
297         }
298     }
299
300     if (pwp->wp_oprot != 0) {          /* if page exists */
301         struct seg *seg;
302         uint_t oprot;
303         int err, retrycnt = 0;
304
305         AS_LOCK_EXIT(as, &as->a_lock);
306         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
307     retry:
308         seg = as_segat(as, addr);
309         ASSERT(seg != NULL);
310         SEGOP_GETPROT(seg, addr, 0, &oprot);
311         if (prot != oprot) {
312             err = SEGOP_SETPROT(seg, addr, PAGE_SIZE, prot);
313             if (err == IE_RETRY) {
314                 ASSERT(retrycnt == 0);
315                 retrycnt++;
316                 goto retry;
317             }
318         }
319         AS_LOCK_EXIT(as, &as->a_lock);
320     } else
321         AS_LOCK_EXIT(as, &as->a_lock);
322
323     /*
324     * When all pages are mapped back to their normal state,
325     * continue the other lwps.
326     */
327     if (!mapin) {
328         ASSERT(p->p_mapcnt > 0);
329         p->p_mapcnt--;
330         if (p->p_mapcnt == 0) {
331             mutex_exit(&p->p_maplock);

```

```

332             mutex_enter(&p->p_lock);
333             continuelwps(p);
334             mutex_exit(&p->p_lock);
335             mutex_enter(&p->p_maplock);
336         }
337     }
338
339     AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
340 }
341
342 AS_LOCK_EXIT(as, &as->a_lock);
343 mutex_exit(&p->p_maplock);
344
345 return (rv);
346 }
347
_____unchanged_portion_omitted_____
348
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;
499
500     if ((as == &kas) || avl_numnodes(&as->a_wpage) == 0)
501         return (0);
502
503     /* Grab the lock because of XHAT (see comment in pr_mappage()) */
504     AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
505     rv = pr_is_watchpage_as(addr, rw, as);
506     AS_LOCK_EXIT(as, &as->a_lock);
507
508     return (rv);
509 }
510
_____unchanged_portion_omitted_____

```

new/usr/src/uts/common/vm/as.h

1

\*\*\*\*\*

11510 Fri May 8 18:02:54 2015

new/usr/src/uts/common/vm/as.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

\*\*\*\*\*

```
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 */
25
26 /*
27 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
28 */
29
30 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
31 /*      All Rights Reserved      */
32
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 */
42
43 #ifndef _VM_AS_H
44 #define _VM_AS_H
45
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>
52
53 #ifdef __cplusplus
54 extern "C" {
```

new/usr/src/uts/common/vm/as.h

2

55 #endif

```
57 /*
58  * VM - Address spaces.
59 */
60
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 */
106
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 (procf) */
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 (procf) */
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_CLRPGGLCK(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 asynchronous 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

\*\*\*\*\*

83524 Fri May 8 18:02:54 2015

new/usr/src/uts/common/vm/seg\_spt.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

```

1819 faultcode_t
1820 segspt_disefault(struct hat *hat, struct seg *seg, caddr_t addr,
1821                 size_t len, enum fault_type type, enum seg_rw rw)
1822 {
1823     struct shm_data *shmd = (struct shm_data *)seg->s_data;
1824     struct seg *sptseg = shmd->shm_sptseg;
1825     struct as *curspt = shmd->shm_sptas;
1826     struct spt_data *sptd = sptseg->s_data;
1827     pgcnt_t npages;
1828     size_t size;
1829     caddr_t segspt_addr, shm_addr;
1830     page_t **ppa;
1831     int i;
1832     ulong_t an_idx = 0;
1833     int err = 0;
1834     int dyn_ism_unmap = hat_supported(HAT_DYNAMIC_ISM_UNMAP, (void *)0);
1835     size_t pgsz;
1836     pgcnt_t pgcnt;
1837     caddr_t a;
1838     pgcnt_t pidx;

1840 #ifdef lint
1841     hat = hat;
1842 #endif
1843     ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));

1845     /*
1846     * Because of the way spt is implemented
1847     * the realsize of the segment does not have to be
1848     * equal to the segment size itself. The segment size is
1849     * often in multiples of a page size larger than PAGESIZE.
1850     * The realsize is rounded up to the nearest PAGESIZE
1851     * based on what the user requested. This is a bit of
1852     * ugliness that is historical but not easily fixed
1853     * without re-designing the higher levels of ISM.
1854     */
1855     ASSERT(addr >= seg->s_base);
1856     if (((addr + len) - seg->s_base) > sptd->spt_realsize)
1857         return (FC_NOMAP);
1858     /*
1859     * For all of the following cases except F_PROT, we need to
1860     * make any necessary adjustments to addr and len
1861     * and get all of the necessary page_t's into an array called ppa[].
1862     *
1863     * The code in shmat() forces base addr and len of ISM segment
1864     * to be aligned to largest page size supported. Therefore,
1865     * we are able to handle F_SOFTLOCK and F_INVALID calls in "large
1866     * pagesize" chunks. We want to make sure that we HAT_LOAD_LOCK
1867     * in large pagesize chunks, or else we will screw up the HAT
1868     * layer by calling hat_memload_array() with differing page sizes
1869     * over a given virtual range.

```

```

1870     /*
1871     pgsz = page_get_pagesize(sptseg->s_szc);
1872     pgcnt = page_get_pagecnt(sptseg->s_szc);
1873     shm_addr = (caddr_t)P2ALIGN((uintptr_t)(addr), pgsz);
1874     size = P2ROUNDUP((uintptr_t)(((addr + len) - shm_addr)), pgsz);
1875     npages = btopr(size);

1877     /*
1878     * Now we need to convert from addr in segshm to addr in segspt.
1879     */
1880     an_idx = seg_page(seg, shm_addr);
1881     segspt_addr = sptseg->s_base + ptob(an_idx);

1883     ASSERT((segspt_addr + ptob(npages)) <=
1884            (sptseg->s_base + sptd->spt_realsize));
1885     ASSERT(segspt_addr < (sptseg->s_base + sptseg->s_size));

1887     switch (type) {

1889     case F_SOFTLOCK:

1891         atomic_add_long((ulong_t *)&(shmd->shm_softlockcnt), npages);
1892         /*
1893         * Fall through to the F_INVALID case to load up the hat layer
1894         * entries with the HAT_LOAD_LOCK flag.
1895         */
1896         /* FALLTHRU */
1897     case F_INVALID:

1899         if ((rw == S_EXEC) && !(sptd->spt_prot & PROT_EXEC))
1900             return (FC_NOMAP);

1902         ppa = kmem_zalloc(npages * sizeof(page_t *), KM_SLEEP);

1904         err = spt_anon_getpages(sptseg, segspt_addr, size, ppa);
1905         if (err != 0) {
1906             if (type == F_SOFTLOCK) {
1907                 atomic_add_long((ulong_t *)&(shmd->shm_softlockcnt), -npages);
1908             }
1909             goto dism_err;
1910         }
1911         AS_LOCK_ENTER(sptseg->s_as, &sptseg->s_as->a_lock, RW_READER);
1912         a = segspt_addr;
1913         pidx = 0;
1914         if (type == F_SOFTLOCK) {

1917             /*
1918             * Load up the translation keeping it
1919             * locked and don't unlock the page.
1920             */
1921             for (; pidx < npages; a += pgsz, pidx += pgcnt) {
1922                 hat_memload_array(sptseg->s_as->a_hat,
1923                                 a, pgsz, &ppa[pidx], sptd->spt_prot,
1924                                 HAT_LOAD_LOCK | HAT_LOAD_SHARE);
1925             }
1926         } else {
1927             if (hat == seg->s_as->a_hat) {

1927                 /*
1928                 * Migrate pages marked for migration
1929                 */
1930                 if (lgrp_optimizations())
1931                     page_migrate(seg, shm_addr, ppa, npages);
1933                 page_migrate(seg, shm_addr, ppa,
1934                             npages);

```

```

1933     for (; pidx < npages; a += pgsz, pidx += pgcnt) {
1934         /* CPU HAT */
1935         for (; pidx < npages;
1936             a += pgsz, pidx += pgcnt) {
1937             hat_memload_array(sptseg->s_as->a_hat,
1938                 a, pgsz, &ppa[pidx],
1939                 sptd->spt_prot,
1940                 HAT_LOAD_SHARE);
1941         }
1942     } else {
1943         /* XHAT. Pass real address */
1944         hat_memload_array(hat, shm_addr,
1945             size, ppa, sptd->spt_prot, HAT_LOAD_SHARE);
1946     }
1947 }
1948
1949
1950     /*
1951     * And now drop the SE_SHARED lock(s).
1952     */
1953     if (dyn_ism_unmap) {
1954         for (i = 0; i < npages; i++) {
1955             page_unlock(ppa[i]);
1956         }
1957     }
1958
1959     if (!dyn_ism_unmap) {
1960         if (hat_share(seg->s_as->a_hat, shm_addr,
1961             curspt->a_hat, segspt_addr, ptob(npages),
1962             seg->s_szc) != 0) {
1963             panic("hat_share err in DISM fault");
1964             /* NOTREACHED */
1965         }
1966         if (type == F_INVALID) {
1967             for (i = 0; i < npages; i++) {
1968                 page_unlock(ppa[i]);
1969             }
1970         }
1971     }
1972
1973     AS_LOCK_EXIT(sptseg->s_as, &sptseg->s_as->a_lock);
1974
1975     if (dism_err)
1976         kmem_free(ppa, npages * sizeof(page_t *));
1977     return (err);
1978
1979     case F_SOFTUNLOCK:
1980
1981         /*
1982         * This is a bit ugly, we pass in the real seg pointer,
1983         * but the segspt_addr is the virtual address within the
1984         * dummy seg.
1985         */
1986         segspt_softunlock(seg, segspt_addr, size, rw);
1987         return (0);
1988
1989     case F_PROT:
1990
1991         /*
1992         * This takes care of the unusual case where a user
1993         * allocates a stack in shared memory and a register
1994         * window overflow is written to that stack page before
1995         * it is otherwise modified.
1996         *
1997         * We can get away with this because ISM segments are
1998         * always rw. Other than this unusual case, there
1999         * should be no instances of protection violations.
2000         */

```

```

1990         return (0);
1991
1992     default:
1993     #ifdef DEBUG
1994         panic("segspt_dismfault default type?");
1995     #else
1996         return (FC_NOMAP);
1997     #endif
1998     }
1999 }
2000
2001
2002 faultcode_t
2003 segspt_shmfault(struct hat *hat, struct seg *seg, caddr_t addr,
2004     size_t len, enum fault_type type, enum seg_rw rw)
2005 {
2006     struct shm_data *shmd = (struct shm_data *)seg->s_data;
2007     struct seg *sptseg = shmd->shm_sptseg;
2008     struct as *curspt = shmd->shm_sptas;
2009     struct spt_data *sptd = sptseg->s_data;
2010     pgcnt_t npages;
2011     size_t size;
2012     caddr_t sptseg_addr, shm_addr;
2013     page_t *pp, **ppa;
2014     int i;
2015     u_offset_t offset;
2016     ulong_t anon_index = 0;
2017     struct vnode *vp;
2018     struct anon_map *amp; /* XXX - for locknest */
2019     struct anon *ap = NULL;
2020     size_t pgsz;
2021     pgcnt_t pgcnt;
2022     caddr_t a;
2023     pgcnt_t pidx;
2024     size_t sz;
2025
2026     #ifdef lint
2027     hat = hat;
2028     #endif
2029
2030     ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as, &seg->s_as->a_lock));
2031
2032     if (sptd->spt_flags & SHM_PAGEABLE) {
2033         return (segspt_dismfault(hat, seg, addr, len, type, rw));
2034     }
2035
2036     /*
2037     * Because of the way spt is implemented
2038     * the real size of the segment does not have to be
2039     * equal to the segment size itself. The segment size is
2040     * often in multiples of a page size larger than PAGE_SIZE.
2041     * The real size is rounded up to the nearest PAGE_SIZE
2042     * based on what the user requested. This is a bit of
2043     * ugliness that is historical but not easily fixed
2044     * without re-designing the higher levels of ISM.
2045     */
2046     ASSERT(addr >= seg->s_base);
2047     if (((addr + len) - seg->s_base) > sptd->spt_realsize)
2048         return (FC_NOMAP);
2049
2050     /*
2051     * For all of the following cases except F_PROT, we need to
2052     * make any necessary adjustments to addr and len
2053     * and get all of the necessary page_t's into an array called ppa[].
2054     *
2055     * The code in shmat() forces base addr and len of ISM segment
2056     * to be aligned to largest page size supported. Therefore,

```

```

2056     * we are able to handle F_SOFTLOCK and F_INVALID calls in "large
2057     * pagesize" chunks. We want to make sure that we HAT_LOAD_LOCK
2058     * in large pagesize chunks, or else we will screw up the HAT
2059     * layer by calling hat_memload_array() with differing page sizes
2060     * over a given virtual range.
2061     */
2062     pgsz = page_get_pagesize(sptseg->s_szc);
2063     pgcnt = page_get_pagecnt(sptseg->s_szc);
2064     shm_addr = (caddr_t)P2ALIGN((uintptr_t)(addr), pgsz);
2065     size = P2ROUNDUP((uintptr_t)((addr + len) - shm_addr), pgsz);
2066     npages = btopr(size);

2068     /*
2069     * Now we need to convert from addr in segshm to addr in segspt.
2070     */
2071     anon_index = seg_page(seg, shm_addr);
2072     sptseg_addr = sptseg->s_base + ptob(anon_index);

2074     /*
2075     * And now we may have to adjust npages downward if we have
2076     * exceeded the realsize of the segment or initial anon
2077     * allocations.
2078     */
2079     if ((sptseg_addr + ptob(npages)) >
2080         (sptseg->s_base + sptd->spt_realsize))
2081         size = (sptseg->s_base + sptd->spt_realsize) - sptseg_addr;

2083     npages = btopr(size);

2085     ASSERT(sptseg_addr < (sptseg->s_base + sptseg->s_size));
2086     ASSERT((sptd->spt_flags & SHM_PAGEABLE) == 0);

2088     switch (type) {

2090     case F_SOFTLOCK:

2092         /*
2093         * availrmem is decremented once during anon_swap_adjust()
2094         * and is incremented during the anon_unresv(), which is
2095         * called from shm_rm_amp() when the segment is destroyed.
2096         */
2097         atomic_add_long((ulong_t *)&(shmd->shm_softlockcnt), npages);
2098         /*
2099         * Some platforms assume that ISM pages are SE_SHARED
2100         * locked for the entire life of the segment.
2101         */
2102         if (!hat_supported(HAT_DYNAMIC_ISM_UNMAP, (void *)0))
2103             return (0);
2104         /*
2105         * Fall through to the F_INVALID case to load up the hat layer
2106         * entries with the HAT_LOAD_LOCK flag.
2107         */

2109         /* FALLTHRU */
2110     case F_INVALID:

2112         if ((rw == S_EXEC) && !(sptd->spt_prot & PROT_EXEC))
2113             return (FC_NOMAP);

2115         /*
2116         * Some platforms that do NOT support DYNAMIC_ISM_UNMAP
2117         * may still rely on this call to hat_share(). That
2118         * would imply that those hat's can fault on a
2119         * HAT_LOAD_LOCK translation, which would seem
2120         * contradictory.
2121         */

```

```

2122         if (!hat_supported(HAT_DYNAMIC_ISM_UNMAP, (void *)0)) {
2123             if (hat_share(seg->s_as->a_hat, seg->s_base,
2124                 curspt->a_hat, sptseg->s_base,
2125                 sptseg->s_size, sptseg->s_szc) != 0) {
2126                 panic("hat_share error in ISM fault");
2127                 /*NOTREACHED*/
2128             }
2129             return (0);
2130         }
2131         ppa = kmem_zalloc(sizeof (page_t *) * npages, KM_SLEEP);

2133         /*
2134         * I see no need to lock the real seg,
2135         * here, because all of our work will be on the underlying
2136         * dummy seg.
2137         *
2138         * sptseg_addr and npages now account for large pages.
2139         */
2140         amp = sptd->spt_amp;
2141         ASSERT(amp != NULL);
2142         anon_index = seg_page(sptseg, sptseg_addr);

2144         ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
2145         for (i = 0; i < npages; i++) {
2146             ap = anon_get_ptr(amp->ahp, anon_index++);
2147             ASSERT(ap != NULL);
2148             swap_xlate(ap, &vp, &offset);
2149             pp = page_lookup(vp, offset, SE_SHARED);
2150             ASSERT(pp != NULL);
2151             ppa[i] = pp;
2152         }
2153         ANON_LOCK_EXIT(&amp->a_rwlock);
2154         ASSERT(i == npages);

2156         /*
2157         * We are already holding the as->a_lock on the user's
2158         * real segment, but we need to hold the a_lock on the
2159         * underlying dummy as. This is mostly to satisfy the
2160         * underlying HAT layer.
2161         */
2162         AS_LOCK_ENTER(sptseg->s_as, &sptseg->s_as->a_lock, RW_READER);
2163         a = sptseg_addr;
2164         pidx = 0;
2165         if (type == F_SOFTLOCK) {
2166             /*
2167             * Load up the translation keeping it
2168             * locked and don't unlock the page.
2169             */
2170             for (; pidx < npages; a += pgsz, pidx += pgcnt) {
2171                 sz = MIN(pgsz, ptob(npages - pidx));
2172                 hat_memload_array(sptseg->s_as->a_hat, a,
2173                     sz, &ppa[pidx], sptd->spt_prot,
2174                     HAT_LOAD_LOCK | HAT_LOAD_SHARE);
2175             }
2176         } else {
2177             if (hat == seg->s_as->a_hat) {
2178                 /*
2179                 * Migrate pages marked for migration.
2180                 */
2181                 if (lgrp_optimizations())
2182                     page_migrate(seg, shm_addr, ppa, npages);
2183                 page_migrate(seg, shm_addr, ppa,
2184                     npages);
2185             }
2186         }
2187         for (; pidx < npages; a += pgsz, pidx += pgcnt) {

```



```
2196     /* CPU HAT */
2197     for (; pidx < npages;
2198         a += pgsz, pidx += pgcnt) {
2184         sz = MIN(pgsz, ptob(npages - pidx));
2185         hat_memload_array(sptseg->s_as->a_hat,
2186             a, sz, &ppa[pidx],
2187             sptd->spt_prot, HAT_LOAD_SHARE);
2203     }
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);
2188     }
2190     /*
2191     * And now drop the SE_SHARED lock(s).
2192     */
2193     for (i = 0; i < npages; i++)
2194         page_unlock(ppa[i]);
2195     }
2196     AS_LOCK_EXIT(sptseg->s_as, &sptseg->s_as->a_lock);
2198     kmem_free(ppa, sizeof (page_t *) * npages);
2199     return (0);
2200 case F_SOFTUNLOCK:
2202     /*
2203     * This is a bit ugly, we pass in the real seg pointer,
2204     * but the sptseg_addr is the virtual address within the
2205     * dummy seg.
2206     */
2207     segspt_softunlock(seg, sptseg_addr, ptob(npages), rw);
2208     return (0);
2210 case F_PROT:
2212     /*
2213     * This takes care of the unusual case where a user
2214     * allocates a stack in shared memory and a register
2215     * window overflow is written to that stack page before
2216     * it is otherwise modified.
2217     *
2218     * We can get away with this because ISM segments are
2219     * always rw. Other than this unusual case, there
2220     * should be no instances of protection violations.
2221     */
2222     return (0);
2224     default:
2225     #ifdef DEBUG
2226         cmn_err(CE_WARN, "segspt_shmfault default type?");
2227     #endif
2228     return (FC_NOMAP);
2229     }
2230 }
_____unchanged_portion_omitted_
```

```

*****
285269 Fri May 8 18:02:55 2015
new/usr/src/uts/common/vm/seg_vn.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.
Asserted notes:
- AS_BUSY flag was used solely by xhat
*****
_____unchanged_portion_omitted_____

3783 int segvn_anypgsz = 0;

3785 #define SEGVN_RESTORE_SOFTLOCK_VP(type, pages) \
3786     if ((type) == F_SOFTLOCK) { \
3787         atomic_add_long((ulong_t *)&(svd->softlockcnt, \
3788             -(pages))); \
3789     }

3791 #define SEGVN_UPDATE_MODBITS(ppa, pages, rw, prot, vpprot) \
3792     if (IS_VMODSORT((ppa)[0]->p_vnode)) { \
3793         if ((rw) == S_WRITE) { \
3794             for (i = 0; i < (pages); i++) { \
3795                 ASSERT((ppa)[i]->p_vnode == \
3796                     (ppa)[0]->p_vnode); \
3797                 hat_setmod((ppa)[i]); \
3798             } \
3799         } else if ((rw) != S_OTHER && \
3800             ((prot) & (vpprot) & PROT_WRITE)) { \
3801             for (i = 0; i < (pages); i++) { \
3802                 ASSERT((ppa)[i]->p_vnode == \
3803                     (ppa)[0]->p_vnode); \
3804                 if (!hat_ismod((ppa)[i])) { \
3805                     prot &= ~PROT_WRITE; \
3806                     break; \
3807                 } \
3808             } \
3809         } \
3810     }

3812 #ifdef VM_STATS

3814 #define SEGVN_VMSTAT_FLTVNPAGES(idx) \
3815     VM_STAT_ADD(segvmstats.fltnvnpages[(idx)]);

3817 #else /* VM_STATS */

3819 #define SEGVN_VMSTAT_FLTVNPAGES(idx)

3821 #endif

3823 static faultcode_t
3824 segvn_fault_vnodepages(struct hat *hat, struct seg *seg, caddr_t lpgaddr,
3825     caddr_t lpgeaddr, enum fault_type type, enum seg_rw rw, caddr_t addr,
3826     caddr_t eaddr, int brkcw)
3827 {
3828     struct segvn_data *svd = (struct segvn_data *)seg->s_data;
3829     struct anon_map *amp = svd->amp;
3830     uchar_t segtype = svd->type;
3831     uint_t szc = seg->s_szc;
3832     size_t pgsz = page_get_pagesize(szc);
3833     size_t maxpgsz = pgsz;
3834     pgcnt_t pages = btop(pgsz);

```

```

3835     pgcnt_t maxpages = pages;
3836     size_t ppsize = (pages + 1) * sizeof (page_t *);
3837     caddr_t a = lpgaddr;
3838     caddr_t maxlpgeaddr = lpgeaddr;
3839     u_offset_t off = svd->offset + (uintptr_t)(a - seg->s_base);
3840     ulong_t aindx = svd->anon_index + seg_page(seg, a);
3841     struct vpage *vpage = (svd->vpage != NULL) ?
3842         &svd->vpage[seg_page(seg, a)] : NULL;
3843     vnode_t *vp = svd->vp;
3844     page_t **ppa;
3845     uint_t pszc;
3846     size_t ppgsz;
3847     pgcnt_t ppages;
3848     faultcode_t err = 0;
3849     int ierr;
3850     int vop_size_err = 0;
3851     uint_t protchk, prot, vpprot;
3852     ulong_t i;
3853     int hat_flag = (type == F_SOFTLOCK) ? HAT_LOAD_LOCK : HAT_LOAD;
3854     anon_sync_obj_t an_cookie;
3855     enum seg_rw arw;
3856     int alloc_failed = 0;
3857     int adjszc_chk;
3858     struct vattr va;
3859     int xhat = 0;
3860     page_t *pplist;
3861     pfn_t pfn;
3862     int physcontig;
3863     int upgrdfail;
3864     int segvn_anypgsz_vnode = 0; /* for now map vnode with 2 page sizes */
3865     int tron = (svd->tr_state == SEGVN_TR_ON);

3866     ASSERT(szc != 0);
3867     ASSERT(vp != NULL);
3868     ASSERT(brkcw == 0 || amp != NULL);
3869     ASSERT(tron == 0 || amp != NULL);
3870     ASSERT(enable_mbit_wa == 0); /* no mbit simulations with large pages */
3871     ASSERT(!(svd->flags & MAP_NORESERVE));
3872     ASSERT(type != F_SOFTUNLOCK);
3873     ASSERT(IS_P2ALIGNED(a, maxpgsz));
3874     ASSERT(amp == NULL || IS_P2ALIGNED(aindx, maxpages));
3875     ASSERT(SEGVN_LOCK_HELD(seg->s_as, &svd->lock));
3876     ASSERT(seg->s_szc < NBBY * sizeof (int));
3877     ASSERT(type != F_SOFTLOCK || lpgeaddr - a == maxpgsz);
3878     ASSERT(svd->tr_state != SEGVN_TR_INIT);

3880     VM_STAT_COND_ADD(type == F_SOFTLOCK, segvmstats.fltnvnpages[0]);
3881     VM_STAT_COND_ADD(type != F_SOFTLOCK, segvmstats.fltnvnpages[1]);

3883     if (svd->flags & MAP_TEXT) {
3884         hat_flag |= HAT_LOAD_TEXT;
3885     }

3887     if (svd->pageprot) {
3888         switch (rw) {
3889             case S_READ:
3890                 protchk = PROT_READ;
3891                 break;
3892             case S_WRITE:
3893                 protchk = PROT_WRITE;
3894                 break;
3895             case S_EXEC:
3896                 protchk = PROT_EXEC;
3897                 break;
3898             case S_OTHER:
3899                 default:

```

```

3900         protchk = PROT_READ | PROT_WRITE | PROT_EXEC;
3901         break;
3902     }
3903     } else {
3904         prot = svd->prot;
3905         /* caller has already done segment level protection check. */
3906     }
3909     if (seg->s_as->a_hat != hat) {
3910         xhat = 1;
3911     }
3908     if (rw == S_WRITE && segtype == MAP_PRIVATE) {
3909         SEGVN_VMSTAT_FLTVNPAGES(2);
3910         arw = S_READ;
3911     } else {
3912         arw = rw;
3913     }
3915     ppa = kmem_alloc(ppasize, KM_SLEEP);
3917     VM_STAT_COND_ADD(amp != NULL, segvnmstats.fltnvpages[3]);
3919     for (;;) {
3920         adjszc_chk = 0;
3921         for (; a < lpgeaddr; a += pgsz, off += pgsz, aindx += pages) {
3922             if (adjszc_chk) {
3923                 while (szc < seg->s_szc) {
3924                     uintptr_t e;
3925                     uint_t tszc;
3926                     tszc = segvn_anypgsz_vnode ? szc + 1 :
3927                         seg->s_szc;
3928                     ppgsz = page_get_pagesize(tszc);
3929                     if (!IS_P2ALIGNED(a, ppgsz) ||
3930                         ((alloc_failed >> tszc) & 0x1)) {
3931                         break;
3932                     }
3933                     SEGVN_VMSTAT_FLTVNPAGES(4);
3934                     szc = tszc;
3935                     pgsz = ppgsz;
3936                     pages = btop(pgsz);
3937                     e = P2ROUNDUP((uintptr_t)eaddr, pgsz);
3938                     lpgeaddr = (caddr_t)e;
3939                 }
3940             }
3942             again:
3943             if (IS_P2ALIGNED(a, maxpgsz) && amp != NULL) {
3944                 ASSERT(IS_P2ALIGNED(aindx, maxpages));
3945                 ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
3946                 anon_array_enter(amp, aindx, &an_cookie);
3947                 if (anon_get_ptr(amp->ahp, aindx) != NULL) {
3948                     SEGVN_VMSTAT_FLTVNPAGES(5);
3949                     ASSERT(anon_pages(amp->ahp, aindx,
3950                         maxpages) == maxpages);
3951                     anon_array_exit(&an_cookie);
3952                     ANON_LOCK_EXIT(&amp->a_rwlock);
3953                     err = segvn_fault_anonpages(hat, seg,
3954                         a, a + maxpgsz, type, rw,
3955                         MAX(a, addr),
3956                         MIN(a + maxpgsz, eaddr), brkcow);
3957                     if (err != 0) {
3958                         SEGVN_VMSTAT_FLTVNPAGES(6);
3959                         goto out;
3960                     }
3961                     if (szc < seg->s_szc) {

```

```

3962             szc = seg->s_szc;
3963             pgsz = maxpgsz;
3964             pages = maxpages;
3965             lpgeaddr = maxlpgeaddr;
3966         }
3967         goto next;
3968     } else {
3969         ASSERT(anon_pages(amp->ahp, aindx,
3970             maxpages) == 0);
3971         SEGVN_VMSTAT_FLTVNPAGES(7);
3972         anon_array_exit(&an_cookie);
3973         ANON_LOCK_EXIT(&amp->a_rwlock);
3974     }
3975 }
3976 ASSERT(!brkcow || IS_P2ALIGNED(a, maxpgsz));
3977 ASSERT(!tron || IS_P2ALIGNED(a, maxpgsz));
3979     if (svd->pageprot != 0 && IS_P2ALIGNED(a, maxpgsz)) {
3980         ASSERT(vp != NULL);
3981         prot = VPP_PROT(vp);
3982         ASSERT(sameprot(seg, a, maxpgsz));
3983         if ((prot & protchk) == 0) {
3984             SEGVN_VMSTAT_FLTVNPAGES(8);
3985             err = FC_PROT;
3986             goto out;
3987         }
3988     }
3989     if (type == F_SOFTLOCK) {
3990         atomic_add_long((ulong_t *)&svd->softlockcnt,
3991             pages);
3992     }
3994     pplist = NULL;
3995     physcontig = 0;
3996     ppa[0] = NULL;
3997     if (!brkcow && !tron && szc &&
3998         !page_exists_physcontig(vp, off, szc,
3999             segtype == MAP_PRIVATE ? ppa : NULL)) {
4000         SEGVN_VMSTAT_FLTVNPAGES(9);
4001         if (page_alloc_pages(vp, seg, a, &pplist, NULL,
4002             szc, 0, 0) && type != F_SOFTLOCK) {
4003             SEGVN_VMSTAT_FLTVNPAGES(10);
4004             psz = 0;
4005             ierr = -1;
4006             alloc_failed |= (1 << szc);
4007             break;
4008         }
4009     }
4010     if (pplist != NULL &&
4011         vp->v_mpssdata == SEGVN_PAGEIO) {
4012         int downsize;
4013         SEGVN_VMSTAT_FLTVNPAGES(11);
4014         physcontig = segvn_fill_vp_pages(svd,
4015             vp, off, szc, ppa, &pplist,
4016             &psz, &downsize);
4017         ASSERT(!physcontig || pplist == NULL);
4018         if (!physcontig && downsize &&
4019             type != F_SOFTLOCK) {
4020             ASSERT(pplist == NULL);
4021             SEGVN_VMSTAT_FLTVNPAGES(12);
4022             ierr = -1;
4023             break;
4024         }
4025     }
4026     ASSERT(!physcontig ||
4027         segtype == MAP_PRIVATE ||
4028         ppa[0] == NULL);
4029     if (physcontig && ppa[0] == NULL) {

```

```

4028         physcontig = 0;
4029     }
4030 } else if (!brkcow && !tron && szc && ppa[0] != NULL) {
4031     SEGVN_VMSTAT_FLTVNPAGES(13);
4032     ASSERT(segtype == MAP_PRIVATE);
4033     physcontig = 1;
4034 }
4035
4037 if (!physcontig) {
4038     SEGVN_VMSTAT_FLTVNPAGES(14);
4039     ppa[0] = NULL;
4040     ierr = VOP_GETPAGE(vp, (offset_t)off, pgsz,
4041         &vpprot, ppa, pgsz, seg, a, arw,
4042         svd->cred, NULL);
4043 #ifdef DEBUG
4044     if (ierr == 0) {
4045         for (i = 0; i < pages; i++) {
4046             ASSERT(PAGE_LOCKED(ppa[i]));
4047             ASSERT(!PP_ISFREE(ppa[i]));
4048             ASSERT(ppa[i]->p_vnode == vp);
4049             ASSERT(ppa[i]->p_offset ==
4050                 off + (i << PAGESHIFT));
4051         }
4052     }
4053 #endif /* DEBUG */
4054     if (segtype == MAP_PRIVATE) {
4055         SEGVN_VMSTAT_FLTVNPAGES(15);
4056         vpprot &= ~PROT_WRITE;
4057     }
4058 } else {
4059     ASSERT(segtype == MAP_PRIVATE);
4060     SEGVN_VMSTAT_FLTVNPAGES(16);
4061     vpprot = PROT_ALL & ~PROT_WRITE;
4062     ierr = 0;
4063 }
4064
4065 if (ierr != 0) {
4066     SEGVN_VMSTAT_FLTVNPAGES(17);
4067     if (pplist != NULL) {
4068         SEGVN_VMSTAT_FLTVNPAGES(18);
4069         page_free_replacement_page(pplist);
4070         page_create_putback(pages);
4071     }
4072     SEGVN_RESTORE_SOFTLOCK_VP(type, pages);
4073     if (a + pgsz <= eaddr) {
4074         SEGVN_VMSTAT_FLTVNPAGES(19);
4075         err = FC_MAKE_ERR(ierr);
4076         goto out;
4077     }
4078     va.va_mask = AT_SIZE;
4079     if (VOP_GETATTR(vp, &va, 0, svd->cred, NULL)) {
4080         SEGVN_VMSTAT_FLTVNPAGES(20);
4081         err = FC_MAKE_ERR(EIO);
4082         goto out;
4083     }
4084     if (btopr(va.va_size) >= btopr(off + pgsz)) {
4085         SEGVN_VMSTAT_FLTVNPAGES(21);
4086         err = FC_MAKE_ERR(ierr);
4087         goto out;
4088     }
4089     if (btopr(va.va_size) <
4090         btopr(off + (eaddr - a))) {
4091         SEGVN_VMSTAT_FLTVNPAGES(22);
4092         err = FC_MAKE_ERR(ierr);
4093         goto out;

```

```

4094     }
4095     if (brkcow || tron || type == F_SOFTLOCK) {
4096         /* can't reduce map area */
4097         SEGVN_VMSTAT_FLTVNPAGES(23);
4098         vop_size_err = 1;
4099         goto out;
4100     }
4101     SEGVN_VMSTAT_FLTVNPAGES(24);
4102     ASSERT(szc != 0);
4103     pszc = 0;
4104     ierr = -1;
4105     break;
4106 }
4107
4108 if (amp != NULL) {
4109     ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
4110     anon_array_enter(amp, aindx, &an_cookie);
4111 }
4112 if (amp != NULL &&
4113     anon_get_ptr(amp->ahp, aindx) != NULL) {
4114     ulong_t taindx = P2ALIGN(aindx, maxpages);
4115
4116     SEGVN_VMSTAT_FLTVNPAGES(25);
4117     ASSERT(anon_pages(amp->ahp, taindx,
4118         maxpages) == maxpages);
4119     for (i = 0; i < pages; i++) {
4120         page_unlock(ppa[i]);
4121     }
4122     anon_array_exit(&an_cookie);
4123     ANON_LOCK_EXIT(&amp->a_rwlock);
4124     if (pplist != NULL) {
4125         page_free_replacement_page(pplist);
4126         page_create_putback(pages);
4127     }
4128     SEGVN_RESTORE_SOFTLOCK_VP(type, pages);
4129     if (szc < seg->s_szc) {
4130         SEGVN_VMSTAT_FLTVNPAGES(26);
4131         /*
4132          * For private segments SOFTLOCK
4133          * either always breaks cow (any rw
4134          * type except S_READ_NOCOW) or
4135          * address space is locked as writer
4136          * (S_READ_NOCOW case) and anon slots
4137          * can't show up on second check.
4138          * Therefore if we are here for
4139          * SOFTLOCK case it must be a cow
4140          * break but cow break never reduces
4141          * szc. text replication (tron) in
4142          * this case works as cow break.
4143          * Thus the assert below.
4144          */
4145         ASSERT(!brkcow && !tron &&
4146             type != F_SOFTLOCK);
4147         pszc = seg->s_szc;
4148         ierr = -2;
4149         break;
4150     }
4151     ASSERT(IS_P2ALIGNED(a, maxpgsz));
4152     goto again;
4153 }
4154 #ifdef DEBUG
4155 if (amp != NULL) {
4156     ulong_t taindx = P2ALIGN(aindx, maxpages);
4157     ASSERT(!anon_pages(amp->ahp, taindx, maxpages));
4158 }
4159 #endif /* DEBUG */

```

```

4161     if (brkcow || tron) {
4162         ASSERT(amp != NULL);
4163         ASSERT(pplist == NULL);
4164         ASSERT(szc == seg->s_szc);
4165         ASSERT(IS_P2ALIGNED(a, maxpgsz));
4166         ASSERT(IS_P2ALIGNED(aindx, maxpages));
4167         SEGVN_VMSTAT_FLTVPAGES(27);
4168         ierr = anon_map_privatepages(amp, aindx, szc,
4169             seg, a, prot, ppa, vpage, segvn_anypgsz,
4170             tron ? PG_LOCAL : 0, svd->cred);
4171         if (ierr != 0) {
4172             SEGVN_VMSTAT_FLTVPAGES(28);
4173             anon_array_exit(&an_cookie);
4174             ANON_LOCK_EXIT(&amp->a_rwlock);
4175             SEGVN_RESTORE_SOFTLOCK_VP(type, pages);
4176             err = FC_MAKE_ERR(ierr);
4177             goto out;
4178         }
4180         ASSERT(!IS_VMODSORT(ppa[0]->p_vnode));
4181         /*
4182          * p_szc can't be changed for locked
4183          * swapfs pages.
4184          */
4185         ASSERT(svd->rcookie ==
4186             HAT_INVALID_REGION_COOKIE);
4187         hat_memload_array(hat, a, pgsz, ppa, prot,
4188             hat_flag);
4190         if (!(hat_flag & HAT_LOAD_LOCK)) {
4191             SEGVN_VMSTAT_FLTVPAGES(29);
4192             for (i = 0; i < pages; i++) {
4193                 page_unlock(ppa[i]);
4194             }
4195         }
4196         anon_array_exit(&an_cookie);
4197         ANON_LOCK_EXIT(&amp->a_rwlock);
4198         goto next;
4199     }
4201     ASSERT(svd->rcookie == HAT_INVALID_REGION_COOKIE ||
4202         (!svd->pageprot && svd->prot == (prot & vpprot)));
4204     pfn = page_pptonum(ppa[0]);
4205     /*
4206      * hat_page_demote() needs an SE_EXCL lock on one of
4207      * constituent page_t's and it decreases root's p_szc
4208      * last. This means if root's p_szc is equal szc and
4209      * all its constituent pages are locked
4210      * hat_page_demote() that could have changed p_szc to
4211      * szc is already done and no new have page_demote()
4212      * can start for this large page.
4213      */
4215     /*
4216      * we need to make sure same mapping size is used for
4217      * the same address range if there's a possibility the
4218      * address is already mapped because hat layer panics
4219      * when translation is loaded for the range already
4220      * mapped with a different page size. We achieve it
4221      * by always using largest page size possible subject
4222      * to the constraints of page size, segment page size
4223      * and page alignment. Since mappings are invalidated
4224      * when those constraints change and make it
4225      * impossible to use previously used mapping size no

```

```

4226         * mapping size conflicts should happen.
4227         */
4229     chkszc:
4230     if ((pszc = ppa[0]->p_szc) == szc &&
4231         IS_P2ALIGNED(pfn, pages)) {
4233         SEGVN_VMSTAT_FLTVPAGES(30);
4234         #ifdef DEBUG
4235         for (i = 0; i < pages; i++) {
4236             ASSERT(PAGE_LOCKED(ppa[i]));
4237             ASSERT(!PP_ISFREE(ppa[i]));
4238             ASSERT(page_pptonum(ppa[i]) ==
4239                 pfn + i);
4240             ASSERT(ppa[i]->p_szc == szc);
4241             ASSERT(ppa[i]->p_vnode == vp);
4242             ASSERT(ppa[i]->p_offset ==
4243                 off + (i << PAGESHIFT));
4244         }
4245         #endif /* DEBUG */
4246         /*
4247          * All pages are of szc we need and they are
4248          * all locked so they can't change szc. load
4249          * translations.
4250          *
4251          * if page got promoted since last check
4252          * we don't need pplist.
4253          */
4254         if (pplist != NULL) {
4255             page_free_replacement_page(pplist);
4256             page_create_putback(pages);
4257         }
4258         if (PP_ISMIGRATE(ppa[0])) {
4259             page_migrate(seg, a, ppa, pages);
4260         }
4261         SEGVN_UPDATE_MODBITS(ppa, pages, rw,
4262             prot, vpprot);
4263         if (!xhat) {
4264             hat_memload_array_region(hat, a, pgsz,
4265                 ppa, prot & vpprot, hat_flag,
4266                 svd->rcookie);
4267         } else {
4268             /*
4269              * avoid large xhat mappings to FS
4270              * pages so that hat_page_demote()
4271              * doesn't need to check for xhat
4272              * large mappings.
4273              * Don't use regions with xhats.
4274              */
4275             for (i = 0; i < pages; i++) {
4276                 hat_memload(hat,
4277                     a + (i << PAGESHIFT),
4278                     ppa[i], prot & vpprot,
4279                     hat_flag);
4280             }
4281         }
4282     }
4283     if (!(hat_flag & HAT_LOAD_LOCK)) {
4284         for (i = 0; i < pages; i++) {
4285             page_unlock(ppa[i]);
4286         }
4287     }
4288     if (amp != NULL) {
4289         anon_array_exit(&an_cookie);
4290         ANON_LOCK_EXIT(&amp->a_rwlock);
4291     }

```

```

4276         goto next;
4277     }
4279     /*
4280     * See if upsize is possible.
4281     */
4282     if (pszc > szc && szc < seg->s_szc &&
4283         (segvn_anypgsz_vnode || pszc >= seg->s_szc)) {
4284         pgcnt_t aphase;
4285         uint_t pszcl = MIN(pszc, seg->s_szc);
4286         ppgsz = page_get_pagesize(pszcl);
4287         ppages = btop(ppgsz);
4288         aphase = btop(P2PHASE((uintptr_t)a, ppgsz));
4290
4291         ASSERT(type != F_SOFTLOCK);
4292
4293         SEGVN_VMSTAT_FLTVNPAGES(31);
4294         if (aphase != P2PHASE(pfn, ppages)) {
4295             segvn_faultvmmpss_align_err4++;
4296         } else {
4297             SEGVN_VMSTAT_FLTVNPAGES(32);
4298             if (pplist != NULL) {
4299                 page_t *pl = pplist;
4300                 page_free_replacement_page(pl);
4301                 page_create_putback(pages);
4302             }
4303             for (i = 0; i < pages; i++) {
4304                 page_unlock(ppa[i]);
4305             }
4306             if (amp != NULL) {
4307                 anon_array_exit(&an_cookie);
4308                 ANON_LOCK_EXIT(&amp->a_rwlock);
4309             }
4310             pszc = pszcl;
4311             ierr = -2;
4312             break;
4313         }
4315     /*
4316     * check if we should use smallest mapping size.
4317     */
4318     upgrdfail = 0;
4319     if (szc == 0 ||
4320         if (szc == 0 || xhat ||
4321             (pszc >= szc &&
4322              !IS_P2ALIGNED(pfn, pages)) ||
4323             (pszc < szc &&
4324              !segvn_full_szcpages(ppa, szc, &upgrdfail,
4325                                   &pszcl))) {
4326
4327         if (upgrdfail && type != F_SOFTLOCK) {
4328             /*
4329             * segvn_full_szcpages failed to lock
4330             * all pages EXCL. Size down.
4331             */
4332             ASSERT(pszc < szc);
4333
4334             SEGVN_VMSTAT_FLTVNPAGES(33);
4335
4336             if (pplist != NULL) {
4337                 page_t *pl = pplist;
4338                 page_free_replacement_page(pl);
4339                 page_create_putback(pages);
4340             }

```

```

4341         for (i = 0; i < pages; i++) {
4342             page_unlock(ppa[i]);
4343         }
4344         if (amp != NULL) {
4345             anon_array_exit(&an_cookie);
4346             ANON_LOCK_EXIT(&amp->a_rwlock);
4347         }
4348         ierr = -1;
4349         break;
4350     }
4351     if (szc != 0 && !upgrdfail) {
4352         if (szc != 0 && !xhat && !upgrdfail) {
4353             segvn_faultvmmpss_align_err5++;
4354         }
4355         SEGVN_VMSTAT_FLTVNPAGES(34);
4356         if (pplist != NULL) {
4357             page_free_replacement_page(pplist);
4358             page_create_putback(pages);
4359         }
4360         SEGVN_UPDATE_MODBITS(ppa, pages, rw,
4361                               prot, vpprot);
4362         if (upgrdfail && segvn_anypgsz_vnode) {
4363             /* SOFTLOCK case */
4364             hat_memload_array_region(hat, a, pgsz,
4365                                     ppa, prot & vpprot, hat_flag,
4366                                     svd->rcookie);
4367         } else {
4368             for (i = 0; i < pages; i++) {
4369                 hat_memload_region(hat,
4370                                   a + (i << PAGESHIFT),
4371                                   ppa[i], prot & vpprot,
4372                                   hat_flag, svd->rcookie);
4373             }
4374         }
4375         if (!(hat_flag & HAT_LOAD_LOCK)) {
4376             for (i = 0; i < pages; i++) {
4377                 page_unlock(ppa[i]);
4378             }
4379         }
4380         if (amp != NULL) {
4381             anon_array_exit(&an_cookie);
4382             ANON_LOCK_EXIT(&amp->a_rwlock);
4383         }
4384         goto next;
4386     }
4387     if (pszc == szc) {
4388         /*
4389         * segvn_full_szcpages() upgraded pages szc.
4390         */
4391         ASSERT(pszc == ppa[0]->p_szc);
4392         ASSERT(IS_P2ALIGNED(pfn, pages));
4393         goto chkszc;
4395     }
4396     if (pszc > szc) {
4397         kmutex_t *szcmtx;
4398         SEGVN_VMSTAT_FLTVNPAGES(35);
4399         /*
4400         * p_szc of ppa[0] can change since we haven't
4401         * locked all constituent pages. Call
4402         * page_lock_szc() to prevent szc changes.
4403         * This should be a rare case that happens when
4404         * multiple segments use a different page size
4405         * to map the same file offsets.
4406         */

```

```

4406     szcmtx = page_szc_lock(ppa[0]);
4407     pszcz = ppa[0]->p_szc;
4408     ASSERT(szcmtx != NULL || pszcz == 0);
4409     ASSERT(ppa[0]->p_szc <= pszcz);
4410     if (pszcz <= szc) {
4411         SEGVN_VMSTAT_FLTVNPAGES(36);
4412         if (szcmtx != NULL) {
4413             mutex_exit(szcmtx);
4414         }
4415         goto chkszc;
4416     }
4417     if (pplist != NULL) {
4418         /*
4419          * page got promoted since last check.
4420          * we don't need preallocated large
4421          * page.
4422          */
4423         SEGVN_VMSTAT_FLTVNPAGES(37);
4424         page_free_replacement_page(pplist);
4425         page_create_putback(pages);
4426     }
4427     SEGVN_UPDATE_MODBITS(ppa, pages, rw,
4428                          prot, vpprot);
4429     hat_memload_array_region(hat, a, pgsz, ppa,
4430                             prot & vpprot, hat_flag, svd->rcookie);
4431     mutex_exit(szcmtx);
4432     if (!(hat_flag & HAT_LOAD_LOCK)) {
4433         for (i = 0; i < pages; i++) {
4434             page_unlock(ppa[i]);
4435         }
4436     }
4437     if (amp != NULL) {
4438         anon_array_exit(&an_cookie);
4439         ANON_LOCK_EXIT(&amp->a_rwlock);
4440     }
4441     goto next;
4442 }
4443
4444 /*
4445  * if page got demoted since last check
4446  * we could have not allocated larger page.
4447  * allocate now.
4448  */
4449 if (pplist == NULL &&
4450     page_alloc_pages(vp, seg, a, &pplist, NULL,
4451                     szc, 0, 0) && type != F_SOFTLOCK) {
4452     SEGVN_VMSTAT_FLTVNPAGES(38);
4453     for (i = 0; i < pages; i++) {
4454         page_unlock(ppa[i]);
4455     }
4456     if (amp != NULL) {
4457         anon_array_exit(&an_cookie);
4458         ANON_LOCK_EXIT(&amp->a_rwlock);
4459     }
4460     ierr = -1;
4461     alloc_failed |= (1 << szc);
4462     break;
4463 }
4464
4465 SEGVN_VMSTAT_FLTVNPAGES(39);
4466
4467 if (pplist != NULL) {
4468     segvn_relocate_pages(ppa, pplist);
4469 #ifdef DEBUG
4470 } else {
4471     ASSERT(type == F_SOFTLOCK);

```

```

4472         SEGVN_VMSTAT_FLTVNPAGES(40);
4473 #endif /* DEBUG */
4474     }
4475
4476     SEGVN_UPDATE_MODBITS(ppa, pages, rw, prot, vpprot);
4477
4478     if (pplist == NULL && segvn_anypgsz_vnode == 0) {
4479         ASSERT(type == F_SOFTLOCK);
4480         for (i = 0; i < pages; i++) {
4481             ASSERT(ppa[i]->p_szc < szc);
4482             hat_memload_region(hat,
4483                                a + (i << PAGESHIFT),
4484                                ppa[i], prot & vpprot, hat_flag,
4485                                svd->rcookie);
4486         }
4487     } else {
4488         ASSERT(pplist != NULL || type == F_SOFTLOCK);
4489         hat_memload_array_region(hat, a, pgsz, ppa,
4490                                  prot & vpprot, hat_flag, svd->rcookie);
4491     }
4492     if (!(hat_flag & HAT_LOAD_LOCK)) {
4493         for (i = 0; i < pages; i++) {
4494             ASSERT(PAGE_SHARED(ppa[i]));
4495             page_unlock(ppa[i]);
4496         }
4497     }
4498     if (amp != NULL) {
4499         anon_array_exit(&an_cookie);
4500         ANON_LOCK_EXIT(&amp->a_rwlock);
4501     }
4502
4503 next:
4504     if (vpage != NULL) {
4505         vpage += pages;
4506     }
4507     adjszc_chk = 1;
4508 }
4509 if (a == lpgeaddr)
4510     break;
4511 ASSERT(a < lpgeaddr);
4512
4513 ASSERT(!brkcow && !tron && type != F_SOFTLOCK);
4514
4515 /*
4516  * ierr == -1 means we failed to map with a large page.
4517  * (either due to allocation/relocation failures or
4518  * misalignment with other mappings to this file.
4519  *
4520  * ierr == -2 means some other thread allocated a large page
4521  * after we gave up tp map with a large page.  retry with
4522  * larger mapping.
4523  */
4524 ASSERT(ierr == -1 || ierr == -2);
4525 ASSERT(ierr == -2 || szc != 0);
4526 ASSERT(ierr == -1 || szc < seg->s_szc);
4527 if (ierr == -2) {
4528     SEGVN_VMSTAT_FLTVNPAGES(41);
4529     ASSERT(pszcz > szc && pszcz <= seg->s_szc);
4530     szc = pszcz;
4531 } else if (segvn_anypgsz_vnode) {
4532     SEGVN_VMSTAT_FLTVNPAGES(42);
4533     szc--;
4534 } else {
4535     SEGVN_VMSTAT_FLTVNPAGES(43);
4536     ASSERT(pszcz < szc);
4537     /*

```

```

4538         * other process created psize large page.
4539         * but we still have to drop to 0 szc.
4540         */
4541         szc = 0;
4542     }

4544     pgsz = page_get_pagesize(szc);
4545     pages = btop(pgsz);
4546     if (ierr == -2) {
4547         /*
4548          * Size up case. Note lpgaddr may only be needed for
4549          * softlock case so we don't adjust it here.
4550          */
4551         a = (caddr_t)P2ALIGN((uintptr_t)a, pgsz);
4552         ASSERT(a >= lpgaddr);
4553         lpgeaddr = (caddr_t)P2ROUNDUP((uintptr_t)eaddr, pgsz);
4554         off = svd->offset + (uintptr_t)(a - seg->s_base);
4555         aindx = svd->anon_index + seg_page(seg, a);
4556         vpage = (svd->vpage != NULL) ?
4557             &svd->vpage[seg_page(seg, a)] : NULL;
4558     } else {
4559         /*
4560          * Size down case. Note lpgaddr may only be needed for
4561          * softlock case so we don't adjust it here.
4562          */
4563         ASSERT(IS_P2ALIGNED(a, pgsz));
4564         ASSERT(IS_P2ALIGNED(lpgeaddr, pgsz));
4565         lpgeaddr = (caddr_t)P2ROUNDUP((uintptr_t)eaddr, pgsz);
4566         ASSERT(a < lpgeaddr);
4567         if (a < addr) {
4568             SEGVN_VMSTAT_FLTVNPAGES(44);
4569             /*
4570              * The beginning of the large page region can
4571              * be pulled to the right to make a smaller
4572              * region. We haven't yet faulted a single
4573              * page.
4574              */
4575             a = (caddr_t)P2ALIGN((uintptr_t)addr, pgsz);
4576             ASSERT(a >= lpgaddr);
4577             off = svd->offset +
4578                 (uintptr_t)(a - seg->s_base);
4579             aindx = svd->anon_index + seg_page(seg, a);
4580             vpage = (svd->vpage != NULL) ?
4581                 &svd->vpage[seg_page(seg, a)] : NULL;
4582         }
4583     }
4584 }
4585 out:
4586 kmem_free(ppa, ppsize);
4587 if (!ierr && !vop_size_err) {
4588     SEGVN_VMSTAT_FLTVNPAGES(45);
4589     return (0);
4590 }
4591 if (type == F_SOFTLOCK && a > lpgaddr) {
4592     SEGVN_VMSTAT_FLTVNPAGES(46);
4593     segvn_softunlock(seg, lpgaddr, a - lpgaddr, S_OTHER);
4594 }
4595 if (!vop_size_err) {
4596     SEGVN_VMSTAT_FLTVNPAGES(47);
4597     return (err);
4598 }
4599 ASSERT(brkcow || tron || type == F_SOFTLOCK);
4600 /*
4601  * Large page end is mapped beyond the end of file and it's a cow
4602  * fault (can be a text replication induced cow) or softlock so we can't
4603  * reduce the map area. For now just demote the segment. This should

```

```

4604         * really only happen if the end of the file changed after the mapping
4605         * was established since when large page segments are created we make
4606         * sure they don't extend beyond the end of the file.
4607         */
4608     SEGVN_VMSTAT_FLTVNPAGES(48);

4610     SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);
4611     SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_WRITER);
4612     err = 0;
4613     if (seg->s_szc != 0) {
4614         segvn_fltvnpages_clrsrc_cnt++;
4615         ASSERT(svd->softlockcnt == 0);
4616         err = segvn_clrsrc(seg);
4617         if (err != 0) {
4618             segvn_fltvnpages_clrsrc_err++;
4619         }
4620     }
4621     ASSERT(err || seg->s_szc == 0);
4622     SEGVN_LOCK_DOWNGRADE(seg->s_as, &svd->lock);
4623     /* segvn_fault will do its job as if szc had been zero to begin with */
4624     return (err == 0 ? IE_RETRY : FC_MAKE_ERR(err));
4625 }

```

unchanged portion omitted



new/usr/src/uts/common/vm/vm\_as.c

1

\*\*\*\*\*

92184 Fri May 8 18:02:55 2015

new/usr/src/uts/common/vm/vm\_as.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

\*\*\*\*\*

```
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/system.h>
49 #include <sys/mman.h>
50 #include <sys/sysmacros.h>
51 #include <sys/cpusvar.h>
52 #include <sys/sysinfo.h>
53 #include <sys/kmem.h>
54 #include <sys/vnode.h>
```

new/usr/src/uts/common/vm/vm\_as.c

2

```
55 #include <sys/vmsystem.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     }
133
134     /* Allocate and initialize a callback entry */
135     cb = kmem_zalloc(sizeof(struct as_callback), sleepflag);
136     if (cb == NULL)
137         return (EAGAIN);
138
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;
144
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;
150
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     }
161
162     mutex_exit(&as->a_contents);
163     return (0);
164 }

```

unchanged portion omitted

```

642 /*
643 * Allocate and initialize an address space data structure.
644 * We call hat_alloc to allow any machine dependent
645 * information in the hat structure to be initialized.
646 */
647 struct as *
648 as_alloc(void)
649 {
650     struct as *as;
651
652     as = kmem_cache_alloc(as_cache, KM_SLEEP);
653
654     as->a_flags = 0;
655     as->a_vbits = 0;
656     as->a_hrm = NULL;
657     as->a_seglast = NULL;
658     as->a_size = 0;
659     as->a_resvsize = 0;
660     as->a_updatedir = 0;

```

```

661     getthretime(&as->a_updatetime);
662     as->a_objectdir = NULL;
663     as->a_sizedir = 0;
664     as->a_userlimit = (caddr_t)USERLIMIT;
665     as->a_lastgap = NULL;
666     as->a_lastgaphl = NULL;
667     as->a_callbacks = NULL;
668
669     AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
670     as->a_hat = hat_alloc(as); /* create hat for default system mmu */
671     AS_LOCK_EXIT(as, &as->a_lock);
672
673     as->a_xhat = NULL;
674
675     return (as);
676 }
677
678 /*
679 * Free an address space data structure.
680 * Need to free the hat first and then
681 * all the segments on this as and finally
682 * the space for the as struct itself.
683 */
684 void
685 as_free(struct as *as)
686 {
687     struct hat *hat = as->a_hat;
688     struct seg *seg, *next;
689     boolean_t free_started = B_FALSE;
690     int called = 0;
691
692 top:
693     /*
694     * Invoke ALL callbacks. as_do_callbacks will do one callback
695     * per call, and not return (-1) until the callback has completed.
696     * When as_do_callbacks returns zero, all callbacks have completed.
697     */
698     mutex_enter(&as->a_contents);
699     while (as->a_callbacks && as_do_callbacks(as, AS_ALL_EVENT, 0, 0))
700         ;
701
702     /* This will prevent new XHATs from attaching to as */
703     if (!called)
704         AS_SETBUSY(as);
705     mutex_exit(&as->a_contents);
706     AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
707
708     if (!free_started) {
709         free_started = B_TRUE;
710         if (!called) {
711             called = 1;
712             hat_free_start(hat);
713             if (as->a_xhat != NULL)
714                 xhat_free_start_all(as);
715         }
716         for (seg = AS_SEGFIRST(as); seg != NULL; seg = next) {
717             int err;
718
719             next = AS_SEGNEXT(as, seg);
720
721             err = SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
722             if (err == EAGAIN) {
723                 mutex_enter(&as->a_contents);
724                 if (as->a_callbacks) {
725                     AS_LOCK_EXIT(as, &as->a_lock);
726                 } else if (!AS_ISNOUNMAPWAIT(as)) {

```

```

717     /*
718     * Memory is currently locked. Wait for a
719     * cv_signal that it has been unlocked, then
720     * try the operation again.
721     */
722     if (AS_ISUNMAPWAIT(as) == 0)
723         cv_broadcast(&as->a_cv);
724     AS_SETUNMAPWAIT(as);
725     AS_LOCK_EXIT(as, &as->a_lock);
726     while (AS_ISUNMAPWAIT(as))
727         cv_wait(&as->a_cv, &as->a_contents);
728 } else {
729     /*
730     * We may have raced with
731     * segvn_reclaim()/segspt_reclaim(). In this
732     * case clean nounmapwait flag and retry since
733     * softlockcnt in this segment may be already
734     * 0. We don't drop as writer lock so our
735     * number of retries without sleeping should
736     * be very small. See segvn_reclaim() for
737     * more comments.
738     */
739     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);
754 if (as->a_xhat != NULL)
755     xhat_free_end_all(as);
756 AS_LOCK_EXIT(as, &as->a_lock);
757
758 /* /proc stuff */
759 ASSERT(avl_numnodes(&as->a_wpage) == 0);
760 if (as->a_objectdir) {
761     kmem_free(as->a_objectdir, as->a_sizedir * sizeof (vnode_t *));
762     as->a_objectdir = NULL;
763     as->a_sizedir = 0;
764 }
765
766 /*
767 * Free the struct as back to kmem. Assert it has no segments.
768 */
769 ASSERT(avl_numnodes(&as->a_segtree) == 0);
770 kmem_cache_free(as_cache, as);
771 }
772
773 int
774 as_dup(struct as *as, struct proc *forkedproc)
775 {
776     struct as *newas;
777     struct seg *seg, *newseg;
778     size_t purgesize = 0;
779     int error;
780
781     AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
782     as_clearwatch(as);

```

```

781     newas = as_alloc();
782     newas->a_userlimit = as->a_userlimit;
783     newas->a_proc = forkedproc;
784
785     AS_LOCK_ENTER(newas, &newas->a_lock, RW_WRITER);
786
787     /* This will prevent new XHATs from attaching */
788     mutex_enter(&as->a_contents);
789     AS_SETBUSY(as);
790     mutex_exit(&as->a_contents);
791     mutex_enter(&newas->a_contents);
792     AS_SETBUSY(newas);
793     mutex_exit(&newas->a_contents);
794
795     (void) hat_dup(as->a_hat, newas->a_hat, NULL, 0, HAT_DUP_SRD);
796
797     for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {
798
799         if (seg->s_flags & S_PURGE) {
800             purgesize += seg->s_size;
801             continue;
802         }
803
804         newseg = seg_alloc(newas, seg->s_base, seg->s_size);
805         if (newseg == NULL) {
806             AS_LOCK_EXIT(newas, &newas->a_lock);
807             as_setwatch(as);
808             mutex_enter(&as->a_contents);
809             AS_CLRBUSY(as);
810             mutex_exit(&as->a_contents);
811             AS_LOCK_EXIT(as, &as->a_lock);
812             as_free(newas);
813             return (-1);
814         }
815         if ((error = SEGOP_DUP(seg, newseg)) != 0) {
816             /*
817             * We call seg_free() on the new seg
818             * because the segment is not set up
819             * completely; i.e. it has no ops.
820             */
821             as_setwatch(as);
822             mutex_enter(&as->a_contents);
823             AS_CLRBUSY(as);
824             mutex_exit(&as->a_contents);
825             AS_LOCK_EXIT(as, &as->a_lock);
826             seg_free(newseg);
827             AS_LOCK_EXIT(newas, &newas->a_lock);
828             as_free(newas);
829             return (error);
830         }
831         newas->a_size += seg->s_size;
832     }
833     newas->a_resvsize = as->a_resvsize - purgesize;
834
835     error = hat_dup(as->a_hat, newas->a_hat, NULL, 0, HAT_DUP_ALL);
836     if (as->a_xhat != NULL)
837         error |= xhat_dup_all(as, newas, NULL, 0, HAT_DUP_ALL);
838
839     mutex_enter(&newas->a_contents);
840     AS_CLRBUSY(newas);
841     mutex_exit(&newas->a_contents);
842     AS_LOCK_EXIT(newas, &newas->a_lock);
843
844     as_setwatch(as);
845     mutex_enter(&as->a_contents);
846     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     klpw_t *lwp = ttolwp(curthread);
851     int is_xhat = 0;
852     int holding_wpage = 0;
853     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;
934     }

936     /* Kernel probe */
937     TNF_PROBE_3(address_fault, "vm pagefault", /* CSTYLED */,
938               tnfn_opaque, address, addr,
939               tnfn_fault_type, fault_type, type,
940               tnfn_seg_access, access, rw);

942     raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
943     rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
944             (size_t)raddr;

946     /*
947      * XXX -- Don't grab the as lock for segkmap. We should grab it for
948      * correctness, but then we could be stuck holding this lock for
949      * a LONG time if the fault needs to be resolved on a slow
950      * filesystem, and then no-one will be able to exec new commands,
951      * as exec'ing requires the write lock on the as.
952      */
953     if (as == &kas && segkmap && segkmap->s_base <= raddr &&
954         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          */
959         seg = segkmap;
960         as_lock_held = 0;
961     } else {
962         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         }

966         seg = as_segat(as, raddr);
967         if (seg == NULL) {

```

```

918     AS_LOCK_EXIT(as, &as->a_lock);
919     if (lwp != NULL)
920     if ((lwp != NULL) && (!is_xhat))
921         lwp->lwp_nostop--;
922     return (FC_NOMAP);
923 }
924     as_lock_held = 1;
925 }
926
927     addrsav = raddr;
928     segsav = seg;
929
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;
942
943         if (!is_xhat || (seg->s_ops != &segdev_ops)) {
944             if (is_xhat && avl_numnodes(&as->a_wpage) != 0 &&
945                 pr_is_watchpage_as(raddr, rw, as)) {
946                 /*
947                  * Handle watch pages. If we're faulting on a
948                  * watched page from an X-hat, we have to
949                  * restore the original permissions while we
950                  * handle the fault.
951                  */
952                 as_clearwatch(as);
953                 holding_wpage = 1;
954             }
955
956             res = SEGOP_FAULT(hat, seg, raddr, ssize, type, rw);
957
958             /* Restore watchpoints */
959             if (holding_wpage) {
960                 as_setwatch(as);
961                 holding_wpage = 0;
962             }
963
964             if (res != 0)
965                 break;
966         } else {
967             /* XHAT does not support seg_dev */
968             res = FC_NOSUPPORT;
969             break;
970         }
971     }
972
973     /*
974     * If we were SOFTLOCKing and encountered a failure,
975     * we must SOFTUNLOCK the range we already did. (Maybe we
976     * should just panic if we are SOFTLOCKing or even SOFTUNLOCKing
977     * right here...)
978     */
979     if (res != 0 && type == F_SOFTLOCK) {
980         for (seg = segsav; addrsav < raddr; addrsav += ssize) {
981             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,
976             F_SOFTUNLOCK, S_OTHER);
977     }
978     if (as_lock_held)
979         AS_LOCK_EXIT(as, &as->a_lock);
980     if (lwp != NULL)
981     if ((lwp != NULL) && (!is_xhat))
982         lwp->lwp_nostop--;
983
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 }
1000     unchanged_portion_omitted
1001
1002     /*
1003     * Swap the pages associated with the address space as out to
1004     * secondary storage, returning the number of bytes actually
1005     * swapped.
1006     *
1007     * The value returned is intended to correlate well with the process's
1008     * memory requirements. Its usefulness for this purpose depends on
1009     * how well the segment-level routines do at returning accurate
1010     * information.
1011     */
1012     size_t
1013     as_swapout(struct as *as)
1014     {
1015         struct seg *seg;
1016         size_t swpcnt = 0;
1017
1018         /*
1019         * Kernel-only processes have given up their address
1020         * spaces. Of course, we shouldn't be attempting to
1021         * swap out such processes in the first place...
1022         */
1023         if (as == NULL)
1024             return (0);
1025
1026         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1027
1028         /* Prevent XHATs from attaching */

```

```
2170     mutex_enter(&as->a_contents);
2171     AS_SETBUSY(as);
2172     mutex_exit(&as->a_contents);

2086     /*
2087     * Free all mapping resources associated with the address
2088     * space. The segment-level swapout routines capitalize
2089     * on this unmapping by scavenging pages that have become
2090     * unmapped here.
2091     */
2092     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);

2094     /*
2095     * Call the swapout routines of all segments in the address
2096     * space to do the actual work, accumulating the amount of
2097     * space reclaimed.
2098     */
2099     for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {
2100         struct seg_ops *ov = seg->s_ops;

2102         /*
2103         * We have to check to see if the seg has
2104         * an ops vector because the seg may have
2105         * been in the middle of being set up when
2106         * the process was picked for swapout.
2107         */
2108         if ((ov != NULL) && (ov->swapout != NULL))
2109             swpcnt += SEGOP_SWAPOUT(seg);
2110     }
2111     AS_LOCK_EXIT(as, &as->a_lock);
2112     return (swpcnt);
2113 }
_____unchanged_portion_omitted_
```

new/usr/src/uts/common/vm/vm\_rm.c

1

\*\*\*\*\*

3093 Fri May 8 18:02:55 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

\*\*\*\*\*

```
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) 1984, 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 #include <sys/types.h>
40 #include <sys/t_lock.h>
41 #include <sys/param.h>
42 #include <sys/system.h>
43 #include <sys/mman.h>
44 #include <sys/sysmacros.h>
45 #include <sys/errno.h>
46 #include <sys/signal.h>
47 #include <sys/user.h>
48 #include <sys/proc.h>
49 #include <sys/cmn_err.h>
50 #include <sys/debug.h>

52 #include <vm/hat.h>
53 #include <vm/as.h>
54 #include <vm/seg_vn.h>
```

new/usr/src/uts/common/vm/vm\_rm.c

2

```
55 #include <vm/rm.h>
56 #include <vm/seg.h>
57 #include <vm/page.h>

59 /*
60 * Yield the memory claim requirement for an address space.
61 *
62 * This is currently implemented as the number of active hardware
63 * translations that have page structures. Therefore, it can
64 * underestimate the traditional resident set size, eg, if the
65 * physical page is present and the hardware translation is missing;
66 * and it can overestimate the rss, eg, if there are active
67 * translations to a frame buffer with page structs.
68 * Also, it does not take sharing into account.
69 * Also, it does not take sharing and XHATs into account.
70 */
71 size_t
72 rm_asrss(as)
73     register struct as *as;
74 {
75     if (as != (struct as *)NULL && as != &kas)
76         return ((size_t)btop(hat_get_mapped_size(as->a_hat)));
77     else
78         return (0);
79 }
80
81 unchanged_portion_omitted
```

```

*****
417773 Fri May 8 18:02:56 2015
new/usr/src/uts/sfmmu/vm/hat_sfmmu.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
*****
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 */
27
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 */
39
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/system.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/archsystem.h>
59 #include <sys/machsystem.h>
60 #include <sys/vmsystem.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/fpusystem.h>
85 #include <vm/mach_kpm.h>
86 #include <sys/callb.h>
87
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    }
111
112 #define SFMMU_VALIDATE_SHAREDHLK(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_pgsize);
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;
1062
1063     hat_lock_init();
1064     hat_kstat_init();
1065
1066     /*
1067      * Hardware-only bits in a TTE
1068      */
1069     MAKE_TTE_MASK(&hw_tte);
1070
1071     hat_init_pagesizes();
1072
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 */
1086
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;
1100
1101     size = max_mmu_ctxdoms * sizeof (mmu_ctx_t *);
1102     mmu_ctxs_tbl = kmem_zalloc(size, KM_SLEEP);
1103
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, 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);
1114
1115     /*
1116      * Intialize ism mapping list lock.
1117      */
1118
1119     mutex_init(&ism_mlist_lock, NULL, MUTEX_DEFAULT, NULL);
1120
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_tll.
1132      */
1133     size = sizeof (sfmmu_t) + sizeof (sfmmu_ctx_t) * (max_mmu_ctxdoms - 1);
1134
1135     sfmmuid_cache = kmem_cache_create("sfmmuid_cache", size,
1136         64, sfmmu_idcache_constructor, sfmmu_idcache_destructor,
1137         NULL, NULL, static_arena, 0);
1138
1139     sfmmu_tsbinfo_cache = kmem_cache_create("sfmmu_tsbinfo_cache",
1140         sizeof (struct tsb_info), 0, NULL, NULL, NULL, NULL, 0);
1141
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);
1151
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(phymem);

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(phymem);

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 = kphymem_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     VM_DUMP_SAFE | VM_SLEEP);

1300     sfmmul_cache = kmem_cache_create("sfmmul_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 #ifndef 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].srd_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_scdcach_constructor, sfmmu_scdcach_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

1448 #define SFMMU_KERNEL_MAXVA \
1449     (kmem64_base ? (uintptr_t)kmem64_end : (SYSLIMIT))

1451 /*
1452     * Allocate a hat structure.

```

```

1453 * Called when an address space first uses a hat.
1454 */
1455 struct hat *
1456 hat_alloc(struct as *as)
1457 {
1458     sfmmu_t *sfmmup;
1459     int i;
1460     uint64_t cnum;
1461     extern uint_t get_color_start(struct as *);
1462
1463     ASSERT(AS_WRITE_HELD(as, &as->a_lock));
1464     sfmmup = kmem_cache_alloc(sfmmuid_cache, KM_SLEEP);
1465     sfmmup->sfmmu_as = as;
1466     sfmmup->sfmmu_flags = 0;
1467     sfmmup->sfmmu_tteflags = 0;
1468     sfmmup->sfmmu_rtteflags = 0;
1469     LOCK_INIT_CLEAR(&sfmmup->sfmmu_ctx_lock);
1470
1471     if (as == &kas) {
1472         ksfmmup = sfmmup;
1473         sfmmup->sfmmu_cext = 0;
1474         cnum = KCONTEXT;
1475
1476         sfmmup->sfmmu_clrstart = 0;
1477         sfmmup->sfmmu_tsb = NULL;
1478         /*
1479          * hat_kern_setup() will call sfmmu_init_ktsbinfo()
1480          * to setup tsb_info for ksfmmup.
1481          */
1482     } else {
1483         /*
1484          * Just set to invalid ctx. When it faults, it will
1485          * get a valid ctx. This would avoid the situation
1486          * where we get a ctx, but it gets stolen and then
1487          * we fault when we try to run and so have to get
1488          * another ctx.
1489          */
1490         sfmmup->sfmmu_cext = 0;
1491         cnum = INVALID_CONTEXT;
1492
1493         /* initialize original physical page coloring bin */
1494         sfmmup->sfmmu_clrstart = get_color_start(as);
1495
1496 #ifdef DEBUG
1497         if (tsb_random_size) {
1498             uint32_t randval = (uint32_t)gettick() >> 4;
1499             int size = randval % (tsb_max_growsize + 1);
1500
1501             /* chose a random tsb size for stress testing */
1502             (void) sfmmu_tsbinfo_alloc(&sfmmup->sfmmu_tsb, size,
1503                                     TSB8K|TSB64K|TSB512K, 0, sfmmup);
1504         } else
1505 #endif /* DEBUG */
1506         (void) sfmmu_tsbinfo_alloc(&sfmmup->sfmmu_tsb,
1507                                   default_tsb_size,
1508                                   TSB8K|TSB64K|TSB512K, 0, sfmmup);
1509         sfmmup->sfmmu_flags = HAT_SWAPPED | HAT_ALLCTX_INVALID;
1510         ASSERT(sfmmup->sfmmu_tsb != NULL);
1511     }
1512
1513     ASSERT(max_mmu_ctxdoms > 0);
1514     for (i = 0; i < max_mmu_ctxdoms; i++) {
1515         sfmmup->sfmmu_ctxs[i].cnum = cnum;
1516         sfmmup->sfmmu_ctxs[i].gnum = 0;
1517     }

```

```

1519         for (i = 0; i < max_mmu_page_sizes; i++) {
1520             sfmmup->sfmmu_ttecnt[i] = 0;
1521             sfmmup->sfmmu_scdrttecnt[i] = 0;
1522             sfmmup->sfmmu_ismttecnt[i] = 0;
1523             sfmmup->sfmmu_scdismttecnt[i] = 0;
1524             sfmmup->sfmmu_pgsz[i] = TTE8K;
1525         }
1526         sfmmup->sfmmu_tsb0_4minflcnt = 0;
1527         sfmmup->sfmmu_iblk = NULL;
1528         sfmmup->sfmmu_ismhat = 0;
1529         sfmmup->sfmmu_scdhat = 0;
1530         sfmmup->sfmmu_ismblkpa = (uint64_t)-1;
1531         if (sfmmup == ksfmmup) {
1532             CPUSET_ALL(sfmmup->sfmmu_cpusran);
1533         } else {
1534             CPUSET_ZERO(sfmmup->sfmmu_cpusran);
1535         }
1536         sfmmup->sfmmu_free = 0;
1537         sfmmup->sfmmu_rmstat = 0;
1538         sfmmup->sfmmu_clrbin = sfmmup->sfmmu_clrstart;
1539         sfmmup->sfmmu_xhat_provider = NULL;
1540         cv_init(&sfmmup->sfmmu_tsb_cv, NULL, CV_DEFAULT, NULL);
1541         sfmmup->sfmmu_srdp = NULL;
1542         SF_RGNMAP_ZERO(sfmmup->sfmmu_region_map);
1543         bzero(sfmmup->sfmmu_hmeregion_links, SFMMU_L1_HMERLINKS_SIZE);
1544         sfmmup->sfmmu_scdp = NULL;
1545         sfmmup->sfmmu_scd_link.next = NULL;
1546         sfmmup->sfmmu_scd_link.prev = NULL;
1547         return (sfmmup);
1548     }
1549 }
1550
1551 unchanged_portion_omitted
1552
1553 1909 /*
1554 1910 * Free all the translation resources for the specified address space.
1555 1911 * Called from as_free when an address space is being destroyed.
1556 1912 */
1557 1913 void
1558 1914 hat_free_start(struct hat *sfmmup)
1559 1915 {
1560     1916     ASSERT(AS_WRITE_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));
1561     1917     ASSERT(sfmmup != ksfmmup);
1562     1922     ASSERT(sfmmup->sfmmu_xhat_provider == NULL);
1563
1564     1919     sfmmup->sfmmu_free = 1;
1565     1920     if (sfmmup->sfmmu_scdp != NULL) {
1566         1921         sfmmu_leave_scd(sfmmup, 0);
1567     }
1568
1569     1924     ASSERT(sfmmup->sfmmu_scdp == NULL);
1570     1925 }
1571
1572 1927 void
1573 1928 hat_free_end(struct hat *sfmmup)
1574 1929 {
1575     1930     int i;
1576
1577     1937     ASSERT(sfmmup->sfmmu_xhat_provider == NULL);
1578     1932     ASSERT(sfmmup->sfmmu_free == 1);
1579     1933     ASSERT(sfmmup->sfmmu_ttecnt[TTE8K] == 0);
1580     1934     ASSERT(sfmmup->sfmmu_ttecnt[TTE64K] == 0);
1581     1935     ASSERT(sfmmup->sfmmu_ttecnt[TTE512K] == 0);
1582     1936     ASSERT(sfmmup->sfmmu_ttecnt[TTE4M] == 0);
1583     1937     ASSERT(sfmmup->sfmmu_ttecnt[TTE32M] == 0);
1584     1938     ASSERT(sfmmup->sfmmu_ttecnt[TTE256M] == 0);
1585
1586     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_hmregion_links[i] != NULL) {
1955                 kmem_free(sfmmup->sfmmu_hmregion_links[i],
1956                     SFMMU_L2_HMERLINKS_SIZE);
1957                 sfmmup->sfmmu_hmregion_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_hmregion_links[i] == NULL);
1966     }
1967 #endif

1969     kmem_cache_free(sfmmuid_cache, sfmmup);
1970 }

1972 /*
1973  * Set up any translation structures, for the specified address space,
1974  * that are needed or preferred when the process is being swapped in.
1975  */
1976 /* ARGSUSED */
1977 void
1978 hat_swapin(struct hat *hat)
1979 {
1980     ASSERT(hat->sfmmu_xhat_provider == NULL);

1982 /*
1983  * Free all of the translation resources, for the specified address space,
1984  * that can be freed while the process is swapped out. Called from as_swapout.
1985  * Also, free up the ctx that this process was using.
1986  */
1987 void
1988 hat_swapout(struct hat *sfmmup)
1989 {
1990     struct hmehash_bucket *hmebp;
1991     struct hme_blk *hmeblkp;
1992     struct hme_blk *pr_hblk = NULL;
1993     struct hme_blk *nx_hblk;
1994     int i;
1995     struct hme_blk *list = NULL;
1996     hatlock_t *hatlockp;
1997     struct tsb_info *tsbinfop;
1998     struct free_tsb {
1999         struct free_tsb *next;
2000         struct tsb_info *tsbinfop;
2001     };
2002     /* free list of TSBs */
2003     struct free_tsb *freelist, *last, *next;

2011     ASSERT(sfmmup->sfmmu_xhat_provider == NULL);
2012     SFMMU_STAT(sf_swapout);

```

```

2006     /*
2007     * There is no way to go from an as to all its translations in sfmmu.
2008     * Here is one of the times when we take the big hit and traverse
2009     * the hash looking for hme_blks to free up. Not only do we free up
2010     * this as hme_blks but all those that are free. We are obviously
2011     * swapping because we need memory so let's free up as much
2012     * as we can.
2013     *
2014     * Note that we don't flush TLB/TSB here -- it's not necessary
2015     * because:
2016     * 1) we free the ctx we're using and throw away the TSB(s);
2017     * 2) processes aren't runnable while being swapped out.
2018     */
2019     ASSERT(sfmmup != KHATID);
2020     for (i = 0; i <= UHMEHASH_SZ; i++) {
2021         hmebp = &uhme_hash[i];
2022         SFMMU_HASH_LOCK(hmebp);
2023         hmeblkp = hmebp->hmeblkp;
2024         pr_hblk = NULL;
2025         while (hmeblkp) {
2035             ASSERT(!hmeblkp->hblk_xhat_bit);

2027             if ((hmeblkp->hblk_tag.htag_id == sfmmup) &&
2028                 !hmeblkp->hblk_shw_bit && !hmeblkp->hblk_lckcnt) {
2029                 ASSERT(!hmeblkp->hblk_shared);
2030                 (void) sfmmu_hblk_unload(sfmmup, hmeblkp,
2031                     (caddr_t) get_hblk_base(hmeblkp),
2032                     get_hblk_endaddr(hmeblkp),
2033                     NULL, HAT_UNLOAD);
2034             }
2035             nx_hblk = hmeblkp->hblk_next;
2036             if (!hmeblkp->hblk_vcvt && !hmeblkp->hblk_hmecnt) {
2037                 ASSERT(!hmeblkp->hblk_lckcnt);
2038                 sfmmu_hblk_hash_rm(hmebp, hmeblkp, pr_hblk,
2039                     &list, 0);
2040             } else {
2041                 pr_hblk = hmeblkp;
2042             }
2043             hmeblkp = nx_hblk;
2044         }
2045         SFMMU_HASH_UNLOCK(hmebp);
2046     }

2048     sfmmu_hblks_list_purge(&list, 0);

2050     /*
2051     * Now free up the ctx so that others can reuse it.
2052     */
2053     hatlockp = sfmmu_hat_enter(sfmmup);

2055     sfmmu_invalidate_ctx(sfmmup);

2057     /*
2058     * Free TSBs, but not tsbinfos, and set SWAPPED flag.
2059     * If TSBs were never swapped in, just return.
2060     * This implies that we don't support partial swapping
2061     * of TSBs -- either all are swapped out, or none are.
2062     *
2063     * We must hold the HAT lock here to prevent racing with another
2064     * thread trying to unmap TTEs from the TSB or running the post-
2065     * relocater after relocating the TSB's memory. Unfortunately, we
2066     * can't free memory while holding the HAT lock or we could
2067     * deadlock, so we build a list of TSBs to be freed after marking
2068     * the tsbinfos as swapped out and free them after dropping the

```

```

2069     * lock.
2070     */
2071     if (SFMMU_FLAGS_ISSET(sfmmup, HAT_SWAPPED)) {
2072         sfmmu_hat_exit(hatlockp);
2073         return;
2074     }
2075
2076     SFMMU_FLAGS_SET(sfmmup, HAT_SWAPPED);
2077     last = freelist = NULL;
2078     for (tsbinfop = sfmmup->sfmmu_tsb; tsbinfop != NULL;
2079         tsbinfop = tsbinfop->tsb_next) {
2080         ASSERT((tsbinfop->tsb_flags & TSB_SWAPPED) == 0);
2081
2082         /*
2083          * Cast the TSB into a struct free_tsb and put it on the free
2084          * list.
2085          */
2086         if (freelist == NULL) {
2087             last = freelist = (struct free_tsb *)tsbinfop->tsb_va;
2088         } else {
2089             last->next = (struct free_tsb *)tsbinfop->tsb_va;
2090             last = last->next;
2091         }
2092         last->next = NULL;
2093         last->tsbinfop = tsbinfop;
2094         tsbinfop->tsb_flags |= TSB_SWAPPED;
2095         /*
2096          * Zero out the TTE to clear the valid bit.
2097          * Note we can't use a value like 0xbad because we want to
2098          * ensure diagnostic bits are NEVER set on TTEs that might
2099          * be loaded. The intent is to catch any invalid access
2100          * to the swapped TSB, such as a thread running with a valid
2101          * context without first calling sfmmu_tsb_swapin() to
2102          * allocate TSB memory.
2103          */
2104         tsbinfop->tsb_tte.ll = 0;
2105     }
2106
2107     /* Now we can drop the lock and free the TSB memory. */
2108     sfmmu_hat_exit(hatlockp);
2109     for (; freelist != NULL; freelist = next) {
2110         next = freelist->next;
2111         sfmmu_tsb_free(freelist->tsbinfop);
2112     }
2113 }
2114
2115 /*
2116  * Duplicate the translations of an as into another newas
2117  */
2118 /* ARGSUSED */
2119 int
2120 hat_dup(struct hat *hat, struct hat *newhat, caddr_t addr, size_t len,
2121         uint_t flag)
2122 {
2123     sf_srd_t *srdp;
2124     sf_scd_t *scdp;
2125     int i;
2126     extern uint_t get_color_start(struct as *);
2127
2128     ASSERT(hat->sfmmu_xhat_provider == NULL);
2129     ASSERT((flag == 0) || (flag == HAT_DUP_ALL) || (flag == HAT_DUP_COW) ||
2130           (flag == HAT_DUP_SRD));
2131     ASSERT(hat != ksfmmup);
2132     ASSERT(newhat != ksfmmup);
2133     ASSERT(flag != HAT_DUP_ALL || hat->sfmmu_srdp == newhat->sfmmu_srdp);

```

```

2134     if (flag == HAT_DUP_COW) {
2135         panic("hat_dup: HAT_DUP_COW not supported");
2136     }
2137
2138     if (flag == HAT_DUP_SRD && ((srdp = hat->sfmmu_srdp) != NULL)) {
2139         ASSERT(srdp->srd_evpt != NULL);
2140         VN_HOLD(srdp->srd_evpt);
2141         ASSERT(srdp->srd_refcnt > 0);
2142         newhat->sfmmu_srdp = srdp;
2143         atomic_inc_32((volatile uint_t *)&srdp->srd_refcnt);
2144     }
2145
2146     /*
2147      * HAT_DUP_ALL flag is used after as duplication is done.
2148      */
2149     if (flag == HAT_DUP_ALL && ((srdp = newhat->sfmmu_srdp) != NULL)) {
2150         ASSERT(newhat->sfmmu_srdp->srd_refcnt >= 2);
2151         newhat->sfmmu_rtteflags = hat->sfmmu_rtteflags;
2152         if (hat->sfmmu_flags & HAT_4MTEXT_FLAG) {
2153             newhat->sfmmu_flags |= HAT_4MTEXT_FLAG;
2154         }
2155
2156         /* check if need to join scd */
2157         if ((scdp = hat->sfmmu_scdp) != NULL &&
2158             newhat->sfmmu_scdp != scdp) {
2159             int ret;
2160             SF_RGNMAP_IS_SUBSET(&newhat->sfmmu_region_map,
2161                               &scdp->scd_region_map, ret);
2162             ASSERT(ret);
2163             sfmmu_join_scd(scdp, newhat);
2164             ASSERT(newhat->sfmmu_scdp == scdp &&
2165                 scdp->scd_refcnt >= 2);
2166             for (i = 0; i < max_mmu_page_sizes; i++) {
2167                 newhat->sfmmu_ismttecnt[i] =
2168                     hat->sfmmu_ismttecnt[i];
2169                 newhat->sfmmu_scdismttecnt[i] =
2170                     hat->sfmmu_scdismttecnt[i];
2171             }
2172         }
2173
2174         sfmmu_check_page_sizes(newhat, 1);
2175     }
2176
2177     if (flag == HAT_DUP_ALL && consistent_coloring == 0 &&
2178         update_proc_pgcolorbase_after_fork != 0) {
2179         hat->sfmmu_clrbin = get_color_start(hat->sfmmu_as);
2180     }
2181     return (0);
2182 }

```

unchanged\_portion\_omitted

```

2192 void
2193 hat_memload_region(struct hat *hat, caddr_t addr, struct page *pp,
2194                   uint_t attr, uint_t flags, hat_region_cookie_t rcookie)
2195 {
2196     uint_t rid;
2197     if (rcookie == HAT_INVALID_REGION_COOKIE) {
2198         if (rcookie == HAT_INVALID_REGION_COOKIE ||
2199             hat->sfmmu_xhat_provider != NULL) {
2198             hat_do_memload(hat, addr, pp, attr, flags,
2199                 SFMMU_INVALID_SHMERID);
2200             return;
2201         }
2202         rid = (uint_t)((uint64_t)rcookie);
2203         ASSERT(rid < SFMMU_MAX_HME_REGIONS);
2204         hat_do_memload(hat, addr, pp, attr, flags, rid);

```

```

2205 }

2207 /*
2208  * Set up addr to map to page pp with protection prot.
2209  * As an optimization we also load the TSB with the
2210  * corresponding tte but it is no big deal if the tte gets kicked out.
2211  */
2212 static void
2213 hat_do_memload(struct hat *hat, caddr_t addr, struct page *pp,
2214               uint_t attr, uint_t flags, uint_t rid)
2215 {
2216     tte_t tte;

2219     ASSERT(hat != NULL);
2220     ASSERT(PAGE_LOCKED(pp));
2221     ASSERT(!((uintptr_t)addr & MMU_PAGEOFFSET));
2222     ASSERT(!(flags & ~SFMMU_LOAD_ALLFLAG));
2223     ASSERT(!(attr & ~SFMMU_LOAD_ALLATTR));
2224     SFMMU_VALIDATE_HMERID(hat, rid, addr, MMU_PAGESIZE);

2226     if (PP_ISFREE(pp)) {
2227         panic("hat_memload: loading a mapping to free page %p",
2228             (void *)pp);
2229     }

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     }

2231     ASSERT((hat == ksfmtup) ||
2232           AS_LOCK_HELD(hat->sfmmu_as, &hat->sfmmu_as->a_lock));

2234     if (flags & ~SFMMU_LOAD_ALLFLAG)
2235         cmn_err(CE_NOTE, "hat_memload: unsupported flags %d",
2236             flags & ~SFMMU_LOAD_ALLFLAG);

2238     if (hat->sfmmu_rmstat)
2239         hat_resvstat(MMU_PAGESIZE, hat->sfmmu_as, addr);

2241 #if defined(SF_ERRATA_57)
2242     if ((hat != ksfmtup) && AS_TYPE_64BIT(hat->sfmmu_as) &&
2243         (addr < errata57_limit) && (attr & PROT_EXEC) &&
2244         !(flags & HAT_LOAD_SHARE)) {
2245         cmn_err(CE_WARN, "hat_memload: illegal attempt to make user "
2246             "page executable");
2247         attr &= ~PROT_EXEC;
2248     }
2249 #endif

2251     sfmmu_memtte(&tte, pp->p_pagenum, attr, TTE8K);
2252     (void) sfmmu_tteload_array(hat, &tte, addr, &pp, flags, rid);

2254     /*
2255      * Check TSB and TLB page sizes.
2256      */
2257     if ((flags & HAT_LOAD_SHARE) == 0) {
2258         sfmmu_check_page_sizes(hat, 1);
2259     }
2260 }

2262 /*
2263  * hat_devload can be called to map real memory (e.g.

```

```

2264  * /dev/kmem) and even though hat_devload will determine pf is
2265  * for memory, it will be unable to get a shared lock on the
2266  * page (because someone else has it exclusively) and will
2267  * pass dp = NULL. If tteload doesn't get a non-NULL
2268  * page pointer it can't cache memory.
2269  */
2270 void
2271 hat_devload(struct hat *hat, caddr_t addr, size_t len, pfn_t pfn,
2272            uint_t attr, int flags)
2273 {
2274     tte_t tte;
2275     struct page *pp = NULL;
2276     int use_lgpg = 0;

2278     ASSERT(hat != NULL);

2299     if (hat->sfmmu_xhat_provider) {
2300         XHAT_DEVLOAD(hat, addr, len, pfn, attr, flags);
2301         return;
2302     }

2280     ASSERT(!(flags & ~SFMMU_LOAD_ALLFLAG));
2281     ASSERT(!(attr & ~SFMMU_LOAD_ALLATTR));
2282     ASSERT((hat == ksfmtup) ||
2283           AS_LOCK_HELD(hat->sfmmu_as, &hat->sfmmu_as->a_lock));
2284     if (len == 0)
2285         panic("hat_devload: zero len");
2286     if (flags & ~SFMMU_LOAD_ALLFLAG)
2287         cmn_err(CE_NOTE, "hat_devload: unsupported flags %d",
2288             flags & ~SFMMU_LOAD_ALLFLAG);

2290 #if defined(SF_ERRATA_57)
2291     if ((hat != ksfmtup) && AS_TYPE_64BIT(hat->sfmmu_as) &&
2292         (addr < errata57_limit) && (attr & PROT_EXEC) &&
2293         !(flags & HAT_LOAD_SHARE)) {
2294         cmn_err(CE_WARN, "hat_devload: illegal attempt to make user "
2295             "page executable");
2296         attr &= ~PROT_EXEC;
2297     }
2298 #endif

2300     /*
2301      * If it's a memory page find its pp
2302      */
2303     if (!(flags & HAT_LOAD_NOCONSIST) && pf_is_memory(pfn)) {
2304         pp = page_numtopp_nolock(pfn);
2305         if (pp == NULL) {
2306             flags |= HAT_LOAD_NOCONSIST;
2307         } else {
2308             if (PP_ISFREE(pp)) {
2309                 panic("hat_memload: loading "
2310                     "a mapping to free page %p",
2311                     (void *)pp);
2312             }
2313             if (!PAGE_LOCKED(pp) && !PP_ISNORELOC(pp)) {
2314                 panic("hat_memload: loading a mapping "
2315                     "to unlocked relocatable page %p",
2316                     (void *)pp);
2317             }
2318             ASSERT(len == MMU_PAGESIZE);
2319         }
2320     }

2322     if (hat->sfmmu_rmstat)
2323         hat_resvstat(len, hat->sfmmu_as, addr);

```

```

2325     if (flags & HAT_LOAD_NOCONSIST) {
2326         attr |= SFMMU_UNCACHEVTTE;
2327         use_lgpg = 1;
2328     }
2329     if (!pf_is_memory(pfn)) {
2330         attr |= SFMMU_UNCACHEPTTE | HAT_NOSYNC;
2331         use_lgpg = 1;
2332         switch (attr & HAT_ORDER_MASK) {
2333             case HAT_STRICTORDER:
2334             case HAT_UNORDERED_OK:
2335                 /*
2336                  * we set the side effect bit for all non
2337                  * memory mappings unless merging is ok
2338                  */
2339                 attr |= SFMMU_SIDEFFECT;
2340                 break;
2341             case HAT_MERGING_OK:
2342             case HAT_LOADCACHING_OK:
2343             case HAT_STORECACHING_OK:
2344                 break;
2345             default:
2346                 panic("hat_devload: bad attr");
2347                 break;
2348         }
2349     }
2350     while (len) {
2351         if (!use_lgpg) {
2352             sfmmu_memtte(&tte, pfn, attr, TTE8K);
2353             (void) sfmmu_tteload_array(hat, &tte, addr, &pp,
2354                                     flags, SFMMU_INVALID_SHMERID);
2355             len -= MMU_PAGESIZE;
2356             addr += MMU_PAGESIZE;
2357             pfn++;
2358             continue;
2359         }
2360         /*
2361          * try to use large pages, check va/pa alignments
2362          * Note that 32M/256M page sizes are not (yet) supported.
2363          */
2364         if ((len >= MMU_PAGESIZE4M) &&
2365             !((uintptr_t)addr & MMU_PAGEOFFSET4M) &&
2366             !(disable_large_pages & (1 << TTE4M)) &&
2367             !(mmu_ptob(pfn) & MMU_PAGEOFFSET4M)) {
2368             sfmmu_memtte(&tte, pfn, attr, TTE4M);
2369             (void) sfmmu_tteload_array(hat, &tte, addr, &pp,
2370                                     flags, SFMMU_INVALID_SHMERID);
2371             len -= MMU_PAGESIZE4M;
2372             addr += MMU_PAGESIZE4M;
2373             pfn += MMU_PAGESIZE4M / MMU_PAGESIZE;
2374         } else if ((len >= MMU_PAGESIZE512K) &&
2375                 !((uintptr_t)addr & MMU_PAGEOFFSET512K) &&
2376                 !(disable_large_pages & (1 << TTE512K)) &&
2377                 !(mmu_ptob(pfn) & MMU_PAGEOFFSET512K)) {
2378             sfmmu_memtte(&tte, pfn, attr, TTE512K);
2379             (void) sfmmu_tteload_array(hat, &tte, addr, &pp,
2380                                     flags, SFMMU_INVALID_SHMERID);
2381             len -= MMU_PAGESIZE512K;
2382             addr += MMU_PAGESIZE512K;
2383             pfn += MMU_PAGESIZE512K / MMU_PAGESIZE;
2384         } else if ((len >= MMU_PAGESIZE64K) &&
2385                 !((uintptr_t)addr & MMU_PAGEOFFSET64K) &&
2386                 !(disable_large_pages & (1 << TTE64K)) &&
2387                 !(mmu_ptob(pfn) & MMU_PAGEOFFSET64K)) {
2388             sfmmu_memtte(&tte, pfn, attr, TTE64K);
2389             (void) sfmmu_tteload_array(hat, &tte, addr, &pp,
2390                                     flags, SFMMU_INVALID_SHMERID);

```

```

2391         len -= MMU_PAGESIZE64K;
2392         addr += MMU_PAGESIZE64K;
2393         pfn += MMU_PAGESIZE64K / MMU_PAGESIZE;
2394     } else {
2395         sfmmu_memtte(&tte, pfn, attr, TTE8K);
2396         (void) sfmmu_tteload_array(hat, &tte, addr, &pp,
2397                                   flags, SFMMU_INVALID_SHMERID);
2398         len -= MMU_PAGESIZE;
2399         addr += MMU_PAGESIZE;
2400         pfn++;
2401     }
2402 }

2404 /*
2405  * Check TSB and TLB page sizes.
2406  */
2407 if ((flags & HAT_LOAD_SHARE) == 0) {
2408     sfmmu_check_page_sizes(hat, 1);
2409 }
2410 }

unchanged_portion_omitted

2420 void
2421 hat_memload_array_region(struct hat *hat, caddr_t addr, size_t len,
2422                         struct page **pps, uint_t attr, uint_t flags,
2423                         hat_region_cookie_t rcookie)
2424 {
2425     uint_t rid;
2426     if (rcookie == HAT_INVALID_REGION_COOKIE) {
2427         if (rcookie == HAT_INVALID_REGION_COOKIE ||
2428             hat->sfmmu_xhat_provider != NULL) {
2429             hat_do_memload_array(hat, addr, len, pps, attr, flags,
2430                                 SFMMU_INVALID_SHMERID);
2431             return;
2432         }
2433         rid = (uint_t)((uint64_t)rcookie);
2434         ASSERT(rid < SFMMU_MAX_HME_REGIONS);
2435         hat_do_memload_array(hat, addr, len, pps, attr, flags, rid);
2436     }
2437 }

2436 /*
2437  * Map the largest extend possible out of the page array. The array may NOT
2438  * be in order. The largest possible mapping a page can have
2439  * is specified in the p_szc field. The p_szc field
2440  * cannot change as long as there any mappings (large or small)
2441  * to any of the pages that make up the large page. (ie. any
2442  * promotion/demotion of page size is not up to the hat but up to
2443  * the page free list manager). The array
2444  * should consist of properly aligned contiguous pages that are
2445  * part of a big page for a large mapping to be created.
2446  */
2447 static void
2448 hat_do_memload_array(struct hat *hat, caddr_t addr, size_t len,
2449                     struct page **pps, uint_t attr, uint_t flags, uint_t rid)
2450 {
2451     int ttesz;
2452     size_t mapsz;
2453     pgcnt_t numpg, npgs;
2454     tte_t tte;
2455     page_t *pp;
2456     uint_t large_pages_disable;

2458     ASSERT(!((uintptr_t)addr & MMU_PAGEOFFSET));
2459     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 }

2461 if (hat->sfmmu_rmstat)
2462     hat_resvstat(len, hat->sfmmu_as, addr);

2464 #if defined(SF_ERRATA_57)
2465     if ((hat != ksfmtup) && AS_TYPE_64BIT(hat->sfmmu_as) &&
2466         (addr < errata57_limit) && (attr & PROT_EXEC) &&
2467         !(flags & HAT_LOAD_SHARE)) {
2468         cmn_err(CE_WARN, "hat_memload_array: illegal attempt to make "
2469             "user page executable");
2470         attr &= ~PROT_EXEC;
2471     }
2472 #endif

2474 /* Get number of pages */
2475 npgs = len >> MMU_PAGESHIFT;

2477 if (flags & HAT_LOAD_SHARE) {
2478     large_pages_disable = disable_ism_large_pages;
2479 } else {
2480     large_pages_disable = disable_large_pages;
2481 }

2483 if (npgs < NHMENTS || large_pages_disable == LARGE_PAGES_OFF) {
2484     sfmmu_memload_batchsmall(hat, addr, pps, attr, flags, npgs,
2485         rid);
2486     return;
2487 }

2489 while (npgs >= NHMENTS) {
2490     pp = *pps;
2491     for (ttesz = pp->p_szc; ttesz != TTE8K; ttesz--) {
2492         /*
2493          * Check if this page size is disabled.
2494          */
2495         if (large_pages_disable & (1 << ttesz))
2496             continue;

2498         numpg = TTEPAGES(ttesz);
2499         mapsz = numpg << MMU_PAGESHIFT;
2500         if ((npgs >= numpg) &&
2501             IS_P2ALIGNED(addr, mapsz) &&
2502             IS_P2ALIGNED(pp->p_pagenum, numpg)) {
2503             /*
2504              * At this point we have enough pages and
2505              * we know the virtual address and the pfn
2506              * are properly aligned. We still need
2507              * to check for physical contiguity but since
2508              * it is very likely that this is the case
2509              * we will assume they are so and undo
2510              * the request if necessary. It would
2511              * be great if we could get a hint flag
2512              * like HAT_CONTIG which would tell us
2513              * the pages are contiguous for sure.
2514              */
2515             sfmmu_memtte(&tte, (*pps)->p_pagenum,
2516                 attr, ttesz);
2517             if (!sfmmu_tteload_array(hat, &tte, addr,
2518                 pps, flags, rid)) {
2519                 break;
2520             }
2521         }

```

```

2522     }
2523     if (ttesz == TTE8K) {
2524         /*
2525          * We were not able to map array using a large page
2526          * batch a hmeblk or fraction at a time.
2527          */
2528         numpg = ((uintptr_t)addr >> MMU_PAGESHIFT)
2529             & (NHMENTS-1);
2530         numpg = NHMENTS - numpg;
2531         ASSERT(numpg <= npgs);
2532         mapsz = numpg * MMU_PAGESIZE;
2533         sfmmu_memload_batchsmall(hat, addr, pps, attr, flags,
2534             numpg, rid);
2535     }
2536     addr += mapsz;
2537     npgs -= numpg;
2538     pps += numpg;
2539 }

2541 if (npgs) {
2542     sfmmu_memload_batchsmall(hat, addr, pps, attr, flags, npgs,
2543         rid);
2544 }

2546 /*
2547 * Check TSB and TLB page sizes.
2548 */
2549 if ((flags & HAT_LOAD_SHARE) == 0) {
2550     sfmmu_check_page_sizes(hat, 1);
2551 }
2552 }

```

unchanged portion omitted

```

3930 /*
3931 * Release one hardware address translation lock on the given address range.
3932 */
3933 void
3934 hat_unlock(struct hat *sfmmup, caddr_t addr, size_t len)
3935 {
3936     struct hmehash_bucket *hmebp;
3937     hmeblk_tag hblktag;
3938     int hmeshift, hashno = 1;
3939     struct hme_blk *hmeblkp, *list = NULL;
3940     caddr_t endaddr;

3942     ASSERT(sfmmup != NULL);
3944     ASSERT((sfmmup == ksfmtup) ||
3945         AS_LOCK_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));
3946     ASSERT((len & MMU_PAGEOFFSET) == 0);
3947     endaddr = addr + len;
3948     hblktag.htag_id = sfmmup;
3949     hblktag.htag_rid = SFMMU_INVALID_SHMERID;

3951     /*
3952     * Spitfire supports 4 page sizes.
3953     * Most pages are expected to be of the smallest page size (8K) and
3954     * these will not need to be rehashed. 64K pages also don't need to be
3955     * rehashed because an hmeblk spans 64K of address space. 512K pages
3956     * might need 1 rehash and 4M pages might need 2 rehashes.
3957     */
3958     while (addr < endaddr) {
3959         hmeshift = HME_HASH_SHIFT(hashno);
3960         hblktag.htag_bspage = HME_HASH_BSPAGE(addr, hmeshift);
3961         hblktag.htag_rehash = hashno;

```

```

3962         hmebp = HME_HASH_FUNCTION(sfmmup, addr, hmeshift);
3964         SFMMU_HASH_LOCK(hmebp);
3966         HME_HASH_SEARCH(hmebp, hblktag, hmeblkp, &list);
3967         if (hmeblkp != NULL) {
3968             ASSERT(!hmeblkp->hblk_shared);
3969             /*
3970              * If we encounter a shadow hmeblk then
3971              * we know there are no valid hmeblks mapping
3972              * this address at this size or larger.
3973              * Just increment address by the smallest
3974              * page size.
3975              */
3976             if (hmeblkp->hblk_shw_bit) {
3977                 addr += MMU_PAGESIZE;
3978             } else {
3979                 addr = sfmmu_hblk_unlock(hmeblkp, addr,
3980                                         endaddr);
3981             }
3982             SFMMU_HASH_UNLOCK(hmebp);
3983             hashno = 1;
3984             continue;
3985         }
3986         SFMMU_HASH_UNLOCK(hmebp);
3988         if (!HME_REHASH(sfmmup) || (hashno >= mmu_hashcnt)) {
3989             /*
3990              * We have traversed the whole list and rehashed
3991              * if necessary without finding the address to unlock
3992              * which should never happen.
3993              */
3994             panic("sfmmu_unlock: addr not found. "
3995                 "addr %p hat %p", (void *)addr, (void *)sfmmup);
3996         } else {
3997             hashno++;
3998         }
3999     }
4001     sfmmu_hblks_list_purge(&list, 0);
4002 }
4004 void
4005 hat_unlock_region(struct hat *sfmmup, caddr_t addr, size_t len,
4006                  hat_region_cookie_t rcookie)
4007 {
4008     sf_srd_t *srdp;
4009     sf_region_t *rgnp;
4010     int ttesz;
4011     uint_t rid;
4012     caddr_t eaddr;
4013     caddr_t va;
4014     int hmeshift;
4015     hmeblk_tag hblktag;
4016     struct hmehash_bucket *hmebp;
4017     struct hme_blk *hmeblkp;
4018     struct hme_blk *pr_hblk;
4019     struct hme_blk *list;
4021     if (rcookie == HAT_INVALID_REGION_COOKIE) {
4022         hat_unlock(sfmmup, addr, len);
4023         return;
4024     }
4026     ASSERT(sfmmup != NULL);
4029     ASSERT(sfmmup->sfmmu_xhat_provider == NULL);

```

```

4027         ASSERT(sfmmup != ksffmmup);
4029         srdp = sfmmup->sfmmu_srdp;
4030         rid = (uint_t)((uint64_t)rcookie);
4031         VERIFY3U(rid, <, SFMMU_MAX_HME_REGIONS);
4032         eaddr = addr + len;
4033         va = addr;
4034         list = NULL;
4035         rgnp = srdp->srd_hmergnp[rid];
4036         SFMMU_VALIDATE_HMERID(sfmmup, rid, addr, len);
4038         ASSERT(IS_P2ALIGNED(addr, TTEBYTES(rgnp->rgn_pgszc)));
4039         ASSERT(IS_P2ALIGNED(len, TTEBYTES(rgnp->rgn_pgszc)));
4040         if (rgnp->rgn_pgszc < HBLK_MIN_TTESZ) {
4041             ttesz = HBLK_MIN_TTESZ;
4042         } else {
4043             ttesz = rgnp->rgn_pgszc;
4044         }
4045         while (va < eaddr) {
4046             while (ttesz < rgnp->rgn_pgszc &&
4047                 IS_P2ALIGNED(va, TTEBYTES(ttesz + 1))) {
4048                 ttesz++;
4049             }
4050             while (ttesz >= HBLK_MIN_TTESZ) {
4051                 if (!(rgnp->rgn_hmeflags & (1 << ttesz))) {
4052                     ttesz--;
4053                     continue;
4054                 }
4055                 hmeshift = HME_HASH_SHIFT(ttesz);
4056                 hblktag.htag_bspage = HME_HASH_BSPAGE(va, hmeshift);
4057                 hblktag.htag_rehash = ttesz;
4058                 hblktag.htag_rid = rid;
4059                 hmebp = HME_HASH_FUNCTION(srdp, va, hmeshift);
4060                 SFMMU_HASH_LOCK(hmebp);
4061                 HME_HASH_SEARCH_PREV(hmebp, hblktag, hmeblkp, pr_hblk,
4062                                     &list);
4063                 if (hmeblkp == NULL) {
4064                     SFMMU_HASH_UNLOCK(hmebp);
4065                     ttesz--;
4066                     continue;
4067                 }
4068                 ASSERT(hmeblkp->hblk_shared);
4069                 va = sfmmu_hblk_unlock(hmeblkp, va, eaddr);
4070                 ASSERT(va >= eaddr ||
4071                     IS_P2ALIGNED((uintptr_t)va, TTEBYTES(ttesz)));
4072                 SFMMU_HASH_UNLOCK(hmebp);
4073                 break;
4074             }
4075             if (ttesz < HBLK_MIN_TTESZ) {
4076                 panic("hat_unlock_region: addr not found "
4077                     "addr %p hat %p", (void *)va, (void *)sfmmup);
4078             }
4079         }
4080         sfmmu_hblks_list_purge(&list, 0);
4082     }

```

unchanged\_portion\_omitted

```

4721 /*
4722  * hat_probe returns 1 if the translation for the address 'addr' is
4723  * loaded, zero otherwise.
4724  *
4725  * hat_probe should be used only for advisory purposes because it may
4726  * occasionally return the wrong value. The implementation must guarantee that
4727  * returning the wrong value is a very rare event. hat_probe is used
4728  * to implement optimizations in the segment drivers.

```

```

4729 *
4730 */
4731 int
4732 hat_probe(struct hat *sfmmup, caddr_t addr)
4733 {
4734     pfn_t pfn;
4735     tte_t tte;

4737     ASSERT(sfmmup != NULL);
4771     ASSERT(sfmmup->sfmmu_xhat_provider == NULL);

4739     ASSERT((sfmmup == ksfcmmup) ||
4740            AS_LOCK_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));

4742     if (sfmmup == ksfcmmup) {
4743         while ((pfn = sfmmu_vatopfn(addr, sfmmup, &tte))
4744                == PFN_SUSPENDED) {
4745             sfmmu_vatopfn_suspended(addr, sfmmup, &tte);
4746         }
4747     } else {
4748         pfn = sfmmu_uvatopfn(addr, sfmmup, NULL);
4749     }

4751     if (pfn != PFN_INVALID)
4752         return (1);
4753     else
4754         return (0);
4755 }

4757 ssize_t
4758 hat_getpagesize(struct hat *sfmmup, caddr_t addr)
4759 {
4760     tte_t tte;

4796     ASSERT(sfmmup->sfmmu_xhat_provider == NULL);

4762     if (sfmmup == ksfcmmup) {
4763         if (sfmmu_vatopfn(addr, sfmmup, &tte) == PFN_INVALID) {
4764             return (-1);
4765         }
4766     } else {
4767         if (sfmmu_uvatopfn(addr, sfmmup, &tte) == PFN_INVALID) {
4768             return (-1);
4769         }
4770     }

4772     ASSERT(TTE_IS_VALID(&tte));
4773     return (TTEBYTES(TTE_CSZ(&tte)));
4774 }

4776 uint_t
4777 hat_getattr(struct hat *sfmmup, caddr_t addr, uint_t *attr)
4778 {
4779     tte_t tte;

4817     ASSERT(sfmmup->sfmmu_xhat_provider == NULL);

4781     if (sfmmup == ksfcmmup) {
4782         if (sfmmu_vatopfn(addr, sfmmup, &tte) == PFN_INVALID) {
4783             tte.ll = 0;
4784         }
4785     } else {
4786         if (sfmmu_uvatopfn(addr, sfmmup, &tte) == PFN_INVALID) {
4787             tte.ll = 0;
4788         }
4789     }

```

```

4790         if (TTE_IS_VALID(&tte)) {
4791             *attr = sfmmu_ptov_attr(&tte);
4792             return (0);
4793         }
4794         *attr = 0;
4795         return ((uint_t)0xffffffff);
4796     }

4798 /*
4799  * Enables more attributes on specified address range (ie. logical OR)
4800  */
4801 void
4802 hat_setattr(struct hat *hat, caddr_t addr, size_t len, uint_t attr)
4803 {
4804     if (hat->sfmmu_xhat_provider) {
4805         XHAT_SETATTR(hat, addr, len, attr);
4806         return;
4807     } else {
4808         /*
4809          * This must be a CPU HAT. If the address space has
4810          * XHATs attached, change attributes for all of them,
4811          * just in case
4812          */
4804         ASSERT(hat->sfmmu_as != NULL);
4805         if (hat->sfmmu_as->a_xhat != NULL)
4806             xhat_setattr_all(hat->sfmmu_as, addr, len, attr);
4807     }

4806     sfmmu_chgattr(hat, addr, len, attr, SFMMU_SETATTR);
4807 }

4809 /*
4810  * Assigns attributes to the specified address range. All the attributes
4811  * are specified.
4812  */
4813 void
4814 hat_chgattr(struct hat *hat, caddr_t addr, size_t len, uint_t attr)
4815 {
4816     if (hat->sfmmu_xhat_provider) {
4817         XHAT_CHGATTR(hat, addr, len, attr);
4818         return;
4819     } else {
4820         /*
4821          * This must be a CPU HAT. If the address space has
4822          * XHATs attached, change attributes for all of them,
4823          * just in case
4824          */
4816         ASSERT(hat->sfmmu_as != NULL);
4817         if (hat->sfmmu_as->a_xhat != NULL)
4818             xhat_chgattr_all(hat->sfmmu_as, addr, len, attr);
4819     }

4818     sfmmu_chgattr(hat, addr, len, attr, SFMMU_CHGATTR);
4819 }

4821 /*
4822  * Remove attributes on the specified address range (ie. logical NAND)
4823  */
4824 void
4825 hat_clrattr(struct hat *hat, caddr_t addr, size_t len, uint_t attr)
4826 {
4827     if (hat->sfmmu_xhat_provider) {
4828         XHAT_CLRATTR(hat, addr, len, attr);
4829         return;
4830     } else {
4831         /*

```

```

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      */
4827      ASSERT(hat->sfmmu_as != NULL);
4899      if (hat->sfmmu_as->a_xhat != NULL)
4900          xhat_clrattr_all(hat->sfmmu_as, addr, len, attr);
4901      }

4829      sfmmu_chgattr(hat, addr, len, attr, SFMMU_CLRATTR);
4830 }

unchanged_portion_omitted_

5156 /*
5157 * hat_chgprot is a deprecated hat call. New segment drivers
5158 * should store all attributes and use hat_*attr calls.
5159 *
5160 * Change the protections in the virtual address range
5161 * given to the specified virtual protection. If vprot is ~PROT_WRITE,
5162 * then remove write permission, leaving the other
5163 * permissions unchanged. If vprot is ~PROT_USER, remove user permissions.
5164 *
5165 */
5166 void
5167 hat_chgprot(struct hat *sfmmup, caddr_t addr, size_t len, uint_t vprot)
5168 {
5169     struct hmehash_bucket *hmebp;
5170     hmeblk_tag hblktag;
5171     int hmeshift, hashno = 1;
5172     struct hme_blk *hmeblkp, *list = NULL;
5173     caddr_t endaddr;
5174     cpuset_t cpuset;
5175     demap_range_t dmr;

5177     ASSERT((len & MMU_PAGEOFFSET) == 0);
5178     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          */
5180         ASSERT(sfmmup->sfmmu_as != NULL);
5264         if (sfmmup->sfmmu_as->a_xhat != NULL)
5265             xhat_chgprot_all(sfmmup->sfmmu_as, addr, len, vprot);
5266     }

5182     CPuset_ZERO(cpuset);

5184     if ((vprot != (uint_t)~PROT_WRITE) && (vprot & PROT_USER) &&
5185         ((addr + len) > (caddr_t)USERLIMIT)) {
5186         panic("user addr %p vprot %x in kernel space",
5187             (void *)addr, vprot);
5188     }
5189     endaddr = addr + len;
5190     hblktag.htag_id = sfmmup;
5191     hblktag.htag_rid = SFMMU_INVALID_SHMERID;
5192     DEMAP_RANGE_INIT(sfmmup, &dmr);

5194     while (addr < endaddr) {
5195         hmeshift = HME_HASH_SHIFT(hashno);
5196         hblktag.htag_bspage = HME_HASH_BSPAGE(addr, hmeshift);

```

```

5197     hblktag.htag_rehash = hashno;
5198     hmebp = HME_HASH_FUNCTION(sfmmup, addr, hmeshift);

5200     SFMMU_HASH_LOCK(hmebp);

5202     HME_HASH_SEARCH(hmebp, hblktag, hmeblkp, &list);
5203     if (hmeblkp != NULL) {
5204         ASSERT(!hmeblkp->hblk_shared);
5205         /*
5206          * We've encountered a shadow hmeblk so skip the range
5207          * of the next smaller mapping size.
5208          */
5209         if (hmeblkp->hblk_shw_bit) {
5210             ASSERT(sfmmup != ksfcmmup);
5211             ASSERT(hashno > 1);
5212             addr = (caddr_t)P2END((uintptr_t)addr,
5213                 TTEBYTES(hashno - 1));
5214         } else {
5215             addr = sfmmu_hblk_chgprot(sfmmup, hmeblkp,
5216                 addr, endaddr, &dmr, vprot);
5217         }
5218         SFMMU_HASH_UNLOCK(hmebp);
5219         hashno = 1;
5220         continue;
5221     }
5222     SFMMU_HASH_UNLOCK(hmebp);

5224     if (!HME_REHASH(sfmmup) || (hashno >= mmu_hashcnt)) {
5225         /*
5226          * We have traversed the whole list and rehashed
5227          * if necessary without finding the address to chgprot.
5228          * This is ok so we increment the address by the
5229          * smallest hmeblk range for kernel mappings and the
5230          * largest hmeblk range, to account for shadow hmeblks,
5231          * for user mappings and continue.
5232          */
5233         if (sfmmup == ksfcmmup)
5234             addr = (caddr_t)P2END((uintptr_t)addr,
5235                 TTEBYTES(1));
5236         else
5237             addr = (caddr_t)P2END((uintptr_t)addr,
5238                 TTEBYTES(hashno));
5239         hashno = 1;
5240     } else {
5241         hashno++;
5242     }
5243     }

5245     sfmmu_hblks_list_purge(&list, 0);
5246     DEMAP_RANGE_FLUSH(&dmr);
5247     cpuset = sfmmup->sfmmu_cpusran;
5248     xt_sync(cpuset);
5249 }

unchanged_portion_omitted_

5584 /*
5585 * Unload all the mappings in the range [addr..addr+len). addr and len must
5586 * be MMU_PAGESIZE aligned.
5587 */

5589 extern struct seg *segkmap;
5590 #define ISSEGKMAP(sfmmup, addr) (sfmmup == ksfcmmup && \
5591     segkmap->s_base <= (addr) && (addr) < (segkmap->s_base + segkmap->s_size))

5594 void

```

```

5595 hat_unload_callback(
5596     struct hat *sfmmup,
5597     caddr_t addr,
5598     size_t len,
5599     uint_t flags,
5600     hat_callback_t *callback)
5601 {
5602     struct hmehash_bucket *hmebp;
5603     hmeblk_tag hblktag;
5604     int hmeshift, hashno, iskernel;
5605     struct hme_blk *hmeblkp, *pr_hblk, *list = NULL;
5606     caddr_t endaddr;
5607     cpuset_t cpuset;
5608     int addr_count = 0;
5609     int a;
5610     caddr_t cb_start_addr[MAX_CB_ADDR];
5611     caddr_t cb_end_addr[MAX_CB_ADDR];
5612     int issegkmap = ISSEGKMAP(sfmmup, addr);
5613     demap_range_t dmr, *dmrpp;

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          */
5615     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     }

5617     ASSERT((sfmmup == ksfsmmup) || (flags & HAT_UNLOAD_OTHER) || \
5618           AS_LOCK_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));

5620     ASSERT(sfmmup != NULL);
5621     ASSERT((len & MMU_PAGEOFFSET) == 0);
5622     ASSERT(!((uintptr_t)addr & MMU_PAGEOFFSET));

5624     /*
5625      * Probing through a large VA range (say 63 bits) will be slow, even
5626      * at 4 Meg steps between the probes. So, when the virtual address range
5627      * is very large, search the HME entries for what to unload.
5628      *
5629      * len >> TTE_PAGE_SHIFT(TTE4M) is the # of 4Meg probes we'd need
5630      *
5631      * UHMEHASH_SZ is number of hash buckets to examine
5632      *
5633      */
5634     if (sfmmup != KHATID && (len >> TTE_PAGE_SHIFT(TTE4M)) > UHMEHASH_SZ) {
5635         hat_unload_large_virtual(sfmmup, addr, len, flags, callback);
5636         return;
5637     }

5639     CPuset_ZERO(cpuset);

5641     /*
5642      * If the process is exiting, we can save a lot of fuss since
5643      * we'll flush the TLB when we free the ctx anyway.
5644      */
5645     if (sfmmup->sfmmu_free) {
5646         dmrp = NULL;
5647     } else {

```

```

5648         dmrp = &dmr;
5649         DEMAP_RANGE_INIT(sfmmup, dmrp);
5650     }

5652     endaddr = addr + len;
5653     hblktag.htag_id = sfmmup;
5654     hblktag.htag_rid = SFMMU_INVALID_SHMERID;

5656     /*
5657      * It is likely for the vm to call unload over a wide range of
5658      * addresses that are actually very sparsely populated by
5659      * translations. In order to speed this up the sfmmu hat supports
5660      * the concept of shadow hmeblks. Dummy large page hmeblks that
5661      * correspond to actual small translations are allocated at tteload
5662      * time and are referred to as shadow hmeblks. Now, during unload
5663      * time, we first check if we have a shadow hmeblk for that
5664      * translation. The absence of one means the corresponding address
5665      * range is empty and can be skipped.
5666      *
5667      * The kernel is an exception to above statement and that is why
5668      * we don't use shadow hmeblks and hash starting from the smallest
5669      * page size.
5670      */
5671     if (sfmmup == KHATID) {
5672         iskernel = 1;
5673         hashno = TTE64K;
5674     } else {
5675         iskernel = 0;
5676         if (mmu_page_sizes == max_mmu_page_sizes) {
5677             hashno = TTE256M;
5678         } else {
5679             hashno = TTE4M;
5680         }
5681     }
5682     while (addr < endaddr) {
5683         hmeshift = HME_HASH_SHIFT(hashno);
5684         hblktag.htag_bspage = HME_HASH_BSPAGE(addr, hmeshift);
5685         hblktag.htag_rehash = hashno;
5686         hmebp = HME_HASH_FUNCTION(sfmmup, addr, hmeshift);

5688         SFMMU_HASH_LOCK(hmebp);

5690         HME_HASH_SEARCH_PREV(hmebp, hblktag, hmeblkp, pr_hblk, &list);
5691         if (hmeblkp == NULL) {
5692             /*
5693              * didn't find an hmeblk. skip the appropriate
5694              * address range.
5695              */
5696             SFMMU_HASH_UNLOCK(hmebp);
5697             if (iskernel) {
5698                 if (hashno < mmu_hashcnt) {
5699                     hashno++;
5700                     continue;
5701                 } else {
5702                     hashno = TTE64K;
5703                     addr = (caddr_t)roundup((uintptr_t)addr
5704                                             + 1, MMU_PAGESIZE64K);
5705                     continue;
5706                 }
5707             }
5708             addr = (caddr_t)roundup((uintptr_t)addr + 1,
5709                                   (1 << hmeshift));
5709         }
5710         if ((uintptr_t)addr & MMU_PAGEOFFSET512K) {
5711             ASSERT(hashno == TTE64K);
5712             continue;
5713         }

```

```

5714         if ((uintptr_t)addr & MMU_PAGEOFFSET4M) {
5715             hashno = TTE512K;
5716             continue;
5717         }
5718         if (mmu_page_sizes == max_mmu_page_sizes) {
5719             if ((uintptr_t)addr & MMU_PAGEOFFSET32M) {
5720                 hashno = TTE4M;
5721                 continue;
5722             }
5723             if ((uintptr_t)addr & MMU_PAGEOFFSET256M) {
5724                 hashno = TTE32M;
5725                 continue;
5726             }
5727             hashno = TTE256M;
5728             continue;
5729         } else {
5730             hashno = TTE4M;
5731             continue;
5732         }
5733     }
5734     ASSERT(hmeblkp);
5735     ASSERT(!hmeblkp->hblk_shared);
5736     if (!hmeblkp->hblk_vcnt && !hmeblkp->hblk_hmecnt) {
5737         /*
5738          * If the valid count is zero we can skip the range
5739          * mapped by this hmeblk.
5740          * We free hblks in the case of HAT_UNMAP. HAT_UNMAP
5741          * is used by segment drivers as a hint
5742          * that the mapping resource won't be used any longer.
5743          * The best example of this is during exit().
5744          */
5745         addr = (caddr_t)roundup((uintptr_t)addr + 1,
5746             get_hblk_span(hmeblkp));
5747         if ((flags & HAT_UNLOAD_UNMAP) ||
5748             (iskernel && !issegkmap)) {
5749             sfmmu_hblk_hash_rm(hmebp, hmeblkp, pr_hblk,
5750                 &list, 0);
5751         }
5752         SFMMU_HASH_UNLOCK(hmebp);
5753     }
5754     if (iskernel) {
5755         hashno = TTE64K;
5756         continue;
5757     }
5758     if ((uintptr_t)addr & MMU_PAGEOFFSET512K) {
5759         ASSERT(hashno == TTE64K);
5760         continue;
5761     }
5762     if ((uintptr_t)addr & MMU_PAGEOFFSET4M) {
5763         hashno = TTE512K;
5764         continue;
5765     }
5766     if (mmu_page_sizes == max_mmu_page_sizes) {
5767         if ((uintptr_t)addr & MMU_PAGEOFFSET32M) {
5768             hashno = TTE4M;
5769             continue;
5770         }
5771         if ((uintptr_t)addr & MMU_PAGEOFFSET256M) {
5772             hashno = TTE32M;
5773             continue;
5774         }
5775         hashno = TTE256M;
5776         continue;
5777     } else {
5778         hashno = TTE4M;
5779         continue;

```

```

5780     }
5781     }
5782     if (hmeblkp->hblk_shw_bit) {
5783         /*
5784          * If we encounter a shadow hmeblk we know there is
5785          * smaller sized hmeblks mapping the same address space.
5786          * Decrement the hash size and rehash.
5787          */
5788         ASSERT(sfmmup != KHATID);
5789         hashno--;
5790         SFMMU_HASH_UNLOCK(hmebp);
5791         continue;
5792     }
5793 }
5794 /*
5795  * track callback address ranges.
5796  * only start a new range when it's not contiguous
5797  */
5798 if (callback != NULL) {
5799     if (addr_count > 0 &&
5800         addr == cb_end_addr[addr_count - 1])
5801         --addr_count;
5802     else
5803         cb_start_addr[addr_count] = addr;
5804 }
5805
5806 addr = sfmmu_hblk_unload(sfmmup, hmeblkp, addr, endaddr,
5807     dmrp, flags);
5808
5809 if (callback != NULL)
5810     cb_end_addr[addr_count++] = addr;
5811
5812 if (((flags & HAT_UNLOAD_UNMAP) || (iskernel && !issegkmap)) &&
5813     !hmeblkp->hblk_vcnt && !hmeblkp->hblk_hmecnt) {
5814     sfmmu_hblk_hash_rm(hmebp, hmeblkp, pr_hblk, &list, 0);
5815 }
5816 SFMMU_HASH_UNLOCK(hmebp);
5817
5818 /*
5819  * Notify our caller as to exactly which pages
5820  * have been unloaded. We do these in clumps,
5821  * to minimize the number of xt_sync()s that need to occur.
5822  */
5823 if (callback != NULL && addr_count == MAX_CB_ADDR) {
5824     if (dmrp != NULL) {
5825         DEMAP_RANGE_FLUSH(dmrp);
5826         cpuset = sfmmup->sfmmu_cpusran;
5827         xt_sync(cpuset);
5828     }
5829 }
5830
5831 for (a = 0; a < MAX_CB_ADDR; ++a) {
5832     callback->hcb_start_addr = cb_start_addr[a];
5833     callback->hcb_end_addr = cb_end_addr[a];
5834     callback->hcb_function(callback);
5835 }
5836 addr_count = 0;
5837 }
5838 if (iskernel) {
5839     hashno = TTE64K;
5840     continue;
5841 }
5842 if ((uintptr_t)addr & MMU_PAGEOFFSET512K) {
5843     ASSERT(hashno == TTE64K);
5844     continue;
5845 }
5846 if ((uintptr_t)addr & MMU_PAGEOFFSET4M) {

```

```

5846             hashno = TTE512K;
5847             continue;
5848         }
5849         if (mmu_page_sizes == max_mmu_page_sizes) {
5850             if ((uintptr_t)addr & MMU_PAGEOFFSET32M) {
5851                 hashno = TTE4M;
5852                 continue;
5853             }
5854             if ((uintptr_t)addr & MMU_PAGEOFFSET256M) {
5855                 hashno = TTE32M;
5856                 continue;
5857             }
5858             hashno = TTE256M;
5859         } else {
5860             hashno = TTE4M;
5861         }
5862     }

5864     sfmmu_hblks_list_purge(&list, 0);
5865     if (dmrp != NULL) {
5866         DEMAP_RANGE_FLUSH(dmrp);
5867         cpuset = sfmmup->sfmmu_cpusran;
5868         xt_sync(cpuset);
5869     }
5870     if (callback && addr_count != 0) {
5871         for (a = 0; a < addr_count; ++a) {
5872             callback->hcb_start_addr = cb_start_addr[a];
5873             callback->hcb_end_addr = cb_end_addr[a];
5874             callback->hcb_function(callback);
5875         }
5876     }

5878     /*
5879     * Check TSB and TLB page sizes if the process isn't exiting.
5880     */
5881     if (!sfmmup->sfmmu_free)
5882         sfmmu_check_page_sizes(sfmmup, 0);
5883 }

5885 /*
5886 * Unload all the mappings in the range [addr..addr+len). addr and len must
5887 * be MMU_PAGESIZE aligned.
5888 */
5889 void
5890 hat_unload(struct hat *sfmmup, caddr_t addr, size_t len, uint_t flags)
5891 {
5892     if (sfmmup->sfmmu_xhat_provider) {
5893         XHAT_UNLOAD(sfmmup, addr, len, flags);
5894         return;
5895     }
5896     hat_unload_callback(sfmmup, addr, len, flags, NULL);
5897 }
5898 }
5899 }
5900 }
5901 }
5902 }
5903 }

unchanged_portion_omitted

6215 /*
6216 * Synchronize all the mappings in the range [addr..addr+len).
6217 * Can be called with clearflag having two states:
6218 * HAT_SYNC_DONTZERO means just return the rm stats
6219 * HAT_SYNC_ZERORM means zero rm bits in the tte and return the stats
6220 */
6221 void
6222 hat_sync(struct hat *sfmmup, caddr_t addr, size_t len, uint_t clearflag)
6223 {
6224     struct hmehash_bucket *hmebp;
6225     hmeblk_tag hblktag;
6226     int hmeshift, hashno = 1;

```

```

6227     struct hme_blk *hmeblkp, *list = NULL;
6228     caddr_t endaddr;
6229     cpuset_t cpuset;

6234     ASSERT(sfmmup->sfmmu_xhat_provider == NULL);
6231     ASSERT((sfmmup == ksfcmmup) ||
6232            AS_LOCK_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));
6233     ASSERT((len & MMU_PAGEOFFSET) == 0);
6234     ASSERT((clearflag == HAT_SYNC_DONTZERO) ||
6235            (clearflag == HAT_SYNC_ZERORM));

6237     CPuset_ZERO(cpuset);

6239     endaddr = addr + len;
6240     hblktag.htag_id = sfmmup;
6241     hblktag.htag_rid = SFMMU_INVALID_SHMERID;

6243     /*
6244     * Spitfire supports 4 page sizes.
6245     * Most pages are expected to be of the smallest page
6246     * size (8K) and these will not need to be rehashed. 64K
6247     * pages also don't need to be rehashed because the an hmeblk
6248     * spans 64K of address space. 512K pages might need 1 rehash and
6249     * and 4M pages 2 rehashes.
6250     */
6251     while (addr < endaddr) {
6252         hmeshift = HME_HASH_SHIFT(hashno);
6253         hblktag.htag_bspage = HME_HASH_BSPAGE(addr, hmeshift);
6254         hblktag.htag_rehash = hashno;
6255         hmebp = HME_HASH_FUNCTION(sfmmup, addr, hmeshift);

6257         SFMMU_HASH_LOCK(hmebp);

6259         HME_HASH_SEARCH(hmebp, hblktag, hmeblkp, &list);
6260         if (hmeblkp != NULL) {
6261             ASSERT(!hmeblkp->hblk_shared);
6262             /*
6263             * We've encountered a shadow hmeblk so skip the range
6264             * of the next smaller mapping size.
6265             */
6266             if (hmeblkp->hblk_shw_bit) {
6267                 ASSERT(sfmmup != ksfcmmup);
6268                 ASSERT(hashno > 1);
6269                 addr = (caddr_t)P2END((uintptr_t)addr,
6270                                     TTEBYTES(hashno - 1));
6271             } else {
6272                 addr = sfmmu_hblk_sync(sfmmup, hmeblkp,
6273                                     addr, endaddr, clearflag);
6274             }
6275             SFMMU_HASH_UNLOCK(hmebp);
6276             hashno = 1;
6277             continue;
6278         }
6279         SFMMU_HASH_UNLOCK(hmebp);

6281         if (!HME_REHASH(sfmmup) || (hashno >= mmu_hashcnt)) {
6282             /*
6283             * We have traversed the whole list and rehashed
6284             * if necessary without finding the address to sync.
6285             * This is ok so we increment the address by the
6286             * smallest hmeblk range for kernel mappings and the
6287             * largest hmeblk range, to account for shadow hmeblks,
6288             * for user mappings and continue.
6289             */
6290             if (sfmmup == ksfcmmup)
6291                 addr = (caddr_t)P2END((uintptr_t)addr,

```

```

6292             TTEBYTES(1));
6293     else
6294         addr = (caddr_t)P2END((uintptr_t)addr,
6295             TTEBYTES(hashno));
6296     hashno = 1;
6297 } else {
6298     hashno++;
6299 }
6300 }
6301 sfmmu_hblks_list_purge(&list, 0);
6302 cpuset = sfmmup->sfmmu_cpuseran;
6303 xt_sync(cpuset);
6304 }
    unchanged_portion_omitted

7022 /*
7023  * Remove all mappings to page 'pp'.
7024  */
7025 int
7026 hat_pageunload(struct page *pp, uint_t forceflag)
7027 {
7028     struct page *origpp = pp;
7029     struct sf_hment *sfhme, *tmphme;
7030     struct hme_blk *hmeblkp;
7031     kmutex_t *pml;
7032 #ifdef VAC
7033     kmutex_t *pmtx;
7034 #endif
7035     cpuset_t cpuset, tset;
7036     int index, cons;
7037     int xhme_blks;
7038     int pa_hments;

7039     ASSERT(PAGE_EXCL(pp));

7040
7041     retry_xhat:
7042     tmphme = NULL;
7043     xhme_blks = 0;
7044     pa_hments = 0;
7045     CPuset_ZERO(cpuset);

7046     pml = sfmmu_mlist_enter(pp);

7047 #ifdef VAC
7048     if (pp->p_kpmref)
7049         sfmmu_kpm_pageunload(pp);
7050     ASSERT(!PP_ISMAPPED_KPM(pp));
7051 #endif
7052     /*
7053      * Clear vpm reference. Since the page is exclusively locked
7054      * vpm cannot be referencing it.
7055      */
7056     if (vpm_enable) {
7057         pp->p_vpmref = 0;
7058     }

7059
7060     index = PP_MAPINDEX(pp);
7061     cons = TTE8K;
7062     retry:
7063     for (sfhme = pp->p_mapping; sfhme; sfhme = tmphme) {
7064         tmphme = sfhme->hme_next;

7065         if (IS_PAHME(sfhme)) {
7066             ASSERT(sfhme->hme_data != NULL);
7067             pa_hments++;
7068             continue;

```

```

7070     }

7071     hmeblkp = sfmmu_hmetohblk(sfhme);
7072     if (hmeblkp->hblk_xhat_bit) {
7073         struct xhat_hme_blk *xblk =
7074             (struct xhat_hme_blk *)hmeblkp;

7075         (void) XHAT_PAGEUNLOAD(xblk->xhat_hme_blk_hat,
7076             pp, forceflag, XBLK2PROVBLK(xblk));

7077         xhme_blks = 1;
7078         continue;
7079     }

7080     /*
7081      * If there are kernel mappings don't unload them, they will
7082      * be suspended.
7083      */
7084     if (forceflag == SFMMU_KERNEL_RELOC && hmeblkp->hblk_lckcnt &&
7085         hmeblkp->hblk_tag.htag_id == ksfmmup)
7086         continue;

7087     tset = sfmmu_pageunload(pp, sfhme, cons);
7088     CPuset_OR(cpuset, tset);
7089 }

7090 while (index != 0) {
7091     index = index >> 1;
7092     if (index != 0)
7093         cons++;
7094     if (index & 0x1) {
7095         /* Go to leading page */
7096         pp = PP_GROUPLLEADER(pp, cons);
7097         ASSERT(sfmmu_mlist_held(pp));
7098         goto retry;
7099     }
7100 }

7101 /*
7102  * cpuset may be empty if the page was only mapped by segkpm,
7103  * in which case we won't actually cross-trap.
7104  */
7105 xt_sync(cpuset);

7106
7107 /*
7108  * The page should have no mappings at this point, unless
7109  * we were called from hat_page_relocate() in which case we
7110  * leave the locked mappings which will be suspended later.
7111  */
7112 ASSERT(!PP_ISMAPPED(origpp) || pa_hments ||
7113     ASSERT(!PP_ISMAPPED(origpp) || xhme_blks || pa_hments ||
7114         (forceflag == SFMMU_KERNEL_RELOC));

7115 #ifdef VAC
7116     if (PP_ISTNC(pp)) {
7117         if (cons == TTE8K) {
7118             pmtx = sfmmu_page_enter(pp);
7119             PP_CLRTNC(pp);
7120             sfmmu_page_exit(pmtx);
7121         } else {
7122             conv_tnc(pp, cons);
7123         }
7124     }
7125 #endif /* VAC */

7126 if (pa_hments && forceflag != SFMMU_KERNEL_RELOC) {

```



```

7125      /*
7126      * Unlink any pa_hments and free them, calling back
7127      * the responsible subsystem to notify it of the error.
7128      * This can occur in situations such as drivers leaking
7129      * DMA handles: naughty, but common enough that we'd like
7130      * to keep the system running rather than bringing it
7131      * down with an obscure error like "pa_hment leaked"
7132      * which doesn't aid the user in debugging their driver.
7133      */
7134      for (sfhme = pp->p_mapping; sfhme; sfhme = tmphme) {
7135          tmphme = sfhme->hme_next;
7136          if (IS_PAHME(sfhme)) {
7137              struct pa_hment *pahmep = sfhme->hme_data;
7138              sfmmu_pahment_leaked(pahmep);
7139              HME_SUB(sfhme, pp);
7140              kmem_cache_free(pa_hment_cache, pahmep);
7141          }
7142      }
7143
7144      ASSERT(!PP_ISMAPPED(origpp));
7261      ASSERT(!PP_ISMAPPED(origpp) || xhme_blks);
7145  }
7146
7147      sfmmu_mlist_exit(pml);
7148
7149      /*
7266      * XHAT may not have finished unloading pages
7267      * because some other thread was waiting for
7268      * mlist lock and XHAT_PAGEUNLOAD let it do
7269      * the job.
7270      */
7271      if (xhme_blks) {
7272          pp = origpp;
7273          goto retry_xhat;
7274      }
7149      return (0);
7150  }
7151  _____ unchanged_portion_omitted _____
7374  uint_t
7375  hat_pagesync(struct page *pp, uint_t clearflag)
7376  {
7377      struct sf_hment *sfhme, *tmphme = NULL;
7378      struct hme_blk *hmeblkp;
7379      kmutex_t *pml;
7380      cpuset_t cpuset, tset;
7381      int index, cons;
7382      extern ulong_t po_share;
7383      page_t *save_pp = pp;
7384      int stop_on_sh = 0;
7385      uint_t shcnt;
7386
7387      CPuset_ZERO(cpuset);
7388
7389      if (PP_ISRO(pp) && (clearflag & HAT_SYNC_STOPON_MOD)) {
7390          return (PP_GENERIC_ATTR(pp));
7391      }
7392
7393      if ((clearflag & HAT_SYNC_ZERORM) == 0) {
7394          if ((clearflag & HAT_SYNC_STOPON_REF) && PP_ISREF(pp)) {
7395              return (PP_GENERIC_ATTR(pp));
7396          }
7397          if ((clearflag & HAT_SYNC_STOPON_MOD) && PP_ISMOD(pp)) {
7398              return (PP_GENERIC_ATTR(pp));
7399          }

```

```

7400          if (clearflag & HAT_SYNC_STOPON_SHARED) {
7401              if (pp->p_share > po_share) {
7402                  hat_page_setattr(pp, P_REF);
7403                  return (PP_GENERIC_ATTR(pp));
7404              }
7405              stop_on_sh = 1;
7406              shcnt = 0;
7407          }
7408      }
7409
7410      clearflag &= ~HAT_SYNC_STOPON_SHARED;
7411      pml = sfmmu_mlist_enter(pp);
7412      index = PP_MAPINDEX(pp);
7413      cons = TTE8K;
7414      retry:
7415      for (sfhme = pp->p_mapping; sfhme; sfhme = tmphme) {
7416          /*
7417          * We need to save the next hment on the list since
7418          * it is possible for pagesync to remove an invalid hment
7419          * from the list.
7420          */
7421          tmphme = sfhme->hme_next;
7422          if (IS_PAHME(sfhme))
7423              continue;
7424          /*
7425          * If we are looking for large mappings and this hme doesn't
7426          * reach the range we are seeking, just ignore it.
7427          */
7428          hmeblkp = sfmmu_hmetohblk(sfhme);
7429          if (hmeblkp->hblk_xhat_bit)
7430              continue;
7431
7432          if (hme_size(sfhme) < cons)
7433              continue;
7434
7435          if (stop_on_sh) {
7436              if (hmeblkp->hblk_shared) {
7437                  sf_srd_t *srdp = hblktosrd(hmeblkp);
7438                  uint_t rid = hmeblkp->hblk_tag.htag_rid;
7439                  sf_region_t *rgnp;
7440                  ASSERT(SFMMU_IS_SHMERID_VALID(rid));
7441                  ASSERT(rid < SFMMU_MAX_HME_REGIONS);
7442                  ASSERT(srdp != NULL);
7443                  rgnp = srdp->srd_hmergnp[rid];
7444                  SFMMU_VALIDATE_SHAREDHBLK(hmeblkp, srdp,
7445                      rgnp, rid);
7446                  shcnt += rgnp->rgn_refcnt;
7447              } else {
7448                  shcnt++;
7449              }
7450              if (shcnt > po_share) {
7451                  /*
7452                  * tell the pager to spare the page this time
7453                  * around.
7454                  */
7455                  hat_page_setattr(save_pp, P_REF);
7456                  index = 0;
7457                  break;
7458              }
7459          }
7460          tset = sfmmu_pagesync(pp, sfhme,
7461              clearflag & ~HAT_SYNC_STOPON_RM);
7462          CPuset_OR(cpuset, tset);
7463      }
7464
7465      /*
7466      * If clearflag is HAT_SYNC_DONTZERO, break out as soon

```

```

7464         * as the "ref" or "mod" is set or share cnt exceeds po_share.
7465         */
7466         if ((clearflag & ~HAT_SYNC_STOPON_RM) == HAT_SYNC_DONTZERO &&
7467             ((clearflag & HAT_SYNC_STOPON_MOD) && PP_ISMOD(save_pp)) ||
7468             ((clearflag & HAT_SYNC_STOPON_REF) && PP_ISREF(save_pp))) {
7469             index = 0;
7470             break;
7471         }
7472     }
7473
7474     while (index) {
7475         index = index >> 1;
7476         cons++;
7477         if (index & 0x1) {
7478             /* Go to leading page */
7479             pp = PP_GROUPLLEADER(pp, cons);
7480             goto retry;
7481         }
7482     }
7483
7484     xt_sync(cpuset);
7485     sfmmu_mlist_exit(pml);
7486     return (PP_GENERIC_ATTR(save_pp));
7487 }

```

unchanged portion omitted

```

7560 /*
7561  * Remove write permission from a mappings to a page, so that
7562  * we can detect the next modification of it. This requires modifying
7563  * the TTE then invalidating (demap) any TLB entry using that TTE.
7564  * This code is similar to sfmmu_pagesync().
7565  */
7566 static cpuset_t
7567 sfmmu_pageclrwr(struct page *pp, struct sf_hment *sfhme)
7568 {
7569     caddr_t addr;
7570     tte_t tte;
7571     tte_t ttemod;
7572     struct hme_blk *hmeblkp;
7573     int ret;
7574     sfmmu_t *sfmmup;
7575     cpuset_t cpuset;
7576
7577     ASSERT(pp != NULL);
7578     ASSERT(sfmmu_mlist_held(pp));
7579
7580     CPUSET_ZERO(cpuset);
7581     SFMMU_STAT(sf_clrwr);
7582
7583 retry:
7584
7585     sfmmu_copytte(&sfhme->hme_tte, &tte);
7586     if (TTE_IS_VALID(&tte) && TTE_IS_WRITABLE(&tte)) {
7587         hmeblkp = sfmmu_hmetohblk(sfhme);
7588
7589
7590
7591         /*
7592          * xhat mappings should never be to a VMODSORT page.
7593          */
7594         ASSERT(hmeblkp->hblk_xhat_bit == 0);
7595
7596         sfmmup = hblktosfmmu(hmeblkp);
7597         addr = tte_to_vaddr(hmeblkp, tte);
7598
7599         ttemod = tte;
7600         TTE_CLR_WRT(&ttemod);
7601         TTE_CLR_MOD(&ttemod);

```

```

7594         ret = sfmmu_modifytte_try(&tte, &ttemod, &sfhme->hme_tte);
7595
7596         /*
7597          * if cas failed and the new value is not what
7598          * we want retry
7599          */
7600         if (ret < 0)
7601             goto retry;
7602
7603         /* we win the cas */
7604         if (ret > 0) {
7605             if (hmeblkp->hblk_shared) {
7606                 sf_srd_t *srdp = (sf_srd_t *)sfmmup;
7607                 uint_t rid = hmeblkp->hblk_tag.htag_rid;
7608                 sf_region_t *rgnp;
7609                 ASSERT(SFMMU_IS_SHMERID_VALID(rid));
7610                 ASSERT(rid < SFMMU_MAX_HME_REGIONS);
7611                 ASSERT(srdp != NULL);
7612                 rgnp = srdp->srd_hmergnp[rid];
7613                 SFMMU_VALIDATE_SHAREDHBLK(hmeblkp,
7614                     srdp, rgnp, rid);
7615                 cpuset = sfmmu_rgntlb_demap(addr,
7616                     rgnp, hmeblkp, 1);
7617             } else {
7618                 sfmmu_tlb_demap(addr, sfmmup, hmeblkp, 0, 0);
7619                 cpuset = sfmmup->sfmmu_cpusran;
7620             }
7621         }
7622     }
7623
7624     return (cpuset);
7625 }

```

unchanged portion omitted

```

7829 #endif /* DEBUG */
7830
7831 /*
7832  * Returns a page frame number for a given virtual address.
7833  * Returns PFN_INVALID to indicate an invalid mapping
7834  */
7835 pfn_t
7836 hat_getpfn(struct hat *hat, caddr_t addr)
7837 {
7838     pfn_t pfn;
7839     tte_t tte;
7840
7841     /*
7842      * We would like to
7843      * ASSERT(AS_LOCK_HELD(as, &as->a_lock));
7844      * but we can't because the iommu driver will call this
7845      * routine at interrupt time and it can't grab the as lock
7846      * or it will deadlock: A thread could have the as lock
7847      * and be waiting for io. The io can't complete
7848      * because the interrupt thread is blocked trying to grab
7849      * the as lock.
7850      */
7851
7852     ASSERT(hat->sfmmu_xhat_provider == NULL);
7853
7854     if (hat == ksfmmup) {
7855         if (IS_KMEM_VA_LARGE PAGE(addr)) {
7856             ASSERT(segkmem_lpsz > 0);
7857             pfn = sfmmu_kvaszc2pfn(addr, segkmem_lpsz);
7858             if (pfn != PFN_INVALID) {
7859                 sfmmu_check_kpfn(pfn);
7860                 return (pfn);
7861             }
7862         }
7863     }

```

```

7860         } else if (segkpm && IS_KPM_ADDR(addr)) {
7861             return (sfmmu_kpm_vatopfn(addr));
7862         }
7863         while ((pfn = sfmmu_vatopfn(addr, ksfmtup, &tte))
7864             == PFN_SUSPENDED) {
7865             sfmmu_vatopfn_suspended(addr, ksfmtup, &tte);
7866         }
7867         sfmmu_check_kpfn(pfn);
7868         return (pfn);
7869     } else {
7870         return (sfmmu_uvatopfn(addr, hat, NULL));
7871     }
7872 }

```

unchanged\_portion\_omitted

```

8027 /*
8028  * For compatability with AT&T and later optimizations
8029  */
8030 /* ARGSUSED */
8031 void
8032 hat_map(struct hat *hat, caddr_t addr, size_t len, uint_t flags)
8033 {
8034     ASSERT(hat != NULL);
8035     ASSERT(hat->sfmmu_xhat_provider == NULL);
8036 }

```

unchanged\_portion\_omitted

```

8088 /*
8089  * Return 1 if the number of mappings exceeds sh_thresh. Return 0
8090  * otherwise. Count shared hmeblks by region's refcnt.
8091  */
8092 int
8093 hat_page_checkshare(page_t *pp, ulong_t sh_thresh)
8094 {
8095     kmutex_t *pml;
8096     ulong_t cnt = 0;
8097     int index, sz = TTE8K;
8098     struct sf_hment *sfhme, *tmphme = NULL;
8099     struct hme_blk *hmeblkp;

8100     pml = sfmmu_mlist_enter(pp);

8101 #ifdef VAC
8102     if (kpm_enable)
8103         cnt = pp->p_kpmref;
8104 #endif

8105     if (vpm_enable && pp->p_vpmref) {
8106         cnt += 1;
8107     }

8108     if (pp->p_share + cnt > sh_thresh) {
8109         sfmmu_mlist_exit(pml);
8110         return (1);
8111     }

8112     index = PP_MAPINDEX(pp);

8113     again:
8114     for (sfhme = pp->p_mapping; sfhme; sfhme = tmphme) {
8115         tmphme = sfhme->hme_next;
8116         if (IS_PAHME(sfhme)) {
8117             continue;
8118         }
8119     }

```

```

8126     hmeblkp = sfmmu_hmetohblk(sfhme);
8127     if (hmeblkp->hblk_xhat_bit) {
8128         cnt++;
8129         if (cnt > sh_thresh) {
8130             sfmmu_mlist_exit(pml);
8131             return (1);
8132         }
8133         continue;
8134     }
8135     if (hme_size(sfhme) != sz) {
8136         continue;
8137     }

```

```

8138     if (hmeblkp->hblk_shared) {
8139         sf_srd_t *srdp = hblkto_srd(hmeblkp);
8140         uint_t rid = hmeblkp->hblk_tag.htag_rid;
8141         sf_region_t *rgnp;
8142         ASSERT(SFMMU_IS_SHMERID_VALID(rid));
8143         ASSERT(rid < SFMMU_MAX_HME_REGIONS);
8144         ASSERT(srdp != NULL);
8145         rgnp = srdp->srd_hmergnp[rid];
8146         SFMMU_VALIDATE_SHAREDHLK(hmeblkp, srdp,
8147             rgnp, rid);
8148         cnt += rgnp->rgn_refcnt;
8149     } else {
8150         cnt++;
8151     }
8152     if (cnt > sh_thresh) {
8153         sfmmu_mlist_exit(pml);
8154         return (1);
8155     }

```

```

8156     index >>= 1;
8157     sz++;
8158     while (index) {
8159         pp = PP_GROUPLADER(pp, sz);
8160         ASSERT(sfmmu_mlist_held(pp));
8161         if (index & 0x1) {
8162             goto again;
8163         }
8164         index >>= 1;
8165         sz++;
8166     }
8167     sfmmu_mlist_exit(pml);
8168     return (0);
8169 }

```

```

8170 /*
8171  * Unload all large mappings to the pp and reset the p_szc field of every
8172  * constituent page according to the remaining mappings.
8173  * pp must be locked SE_EXCL. Even though no other constituent pages are
8174  * locked it's legal to unload the large mappings to the pp because all
8175  * constituent pages of large locked mappings have to be locked SE_SHARED.
8176  * This means if we have SE_EXCL lock on one of constituent pages none of the
8177  * large mappings to pp are locked.
8178  * Decrease p_szc field starting from the last constituent page and ending
8179  * with the root page. This method is used because other threads rely on the
8180  * root's p_szc to find the lock to synchronize on. After a root page_t's p_szc
8181  * is demoted then other threads will succeed in sfmmu_mlspl_enter(). This
8182  * ensures that p_szc changes of the constituent pages appears atomic for all
8183  * threads that use sfmmu_mlspl_enter() to examine p_szc field.
8184  * This mechanism is only used for file system pages where it's not always

```

```

8184 * possible to get SE_EXCL locks on all constituent pages to demote the size
8185 * code (as is done for anonymous or kernel large pages).
8186 *
8187 * See more comments in front of sfmmu_mlspl_enter().
8188 */
8189 void
8190 hat_page_demote(page_t *pp)
8191 {
8192     int index;
8193     int sz;
8194     cpuset_t cpuset;
8195     int sync = 0;
8196     page_t *rootpp;
8197     struct sf_hment *sfhme;
8198     struct sf_hment *tmphme = NULL;
8199     struct hme_blk *hmeblkp;
8200     uint_t pszc;
8201     page_t *lastpp;
8202     cpuset_t tset;
8203     pgcnt_t npgs;
8204     kmutex_t *pml;
8205     kmutex_t *pmtx = NULL;

8207     ASSERT(PAGE_EXCL(pp));
8208     ASSERT(!PP_ISFREE(pp));
8209     ASSERT(!PP_ISKAS(pp));
8210     ASSERT(page_szc_lock_assert(pp));
8211     pml = sfmmu_mlist_enter(pp);

8213     pszc = pp->p_szc;
8214     if (pszc == 0) {
8215         goto out;
8216     }

8218     index = PP_MAPINDEX(pp) >> 1;

8220     if (index) {
8221         CPUSET_ZERO(cpuset);
8222         sz = TTE64K;
8223         sync = 1;
8224     }

8226     while (index) {
8227         if (!(index & 0x1)) {
8228             index >>= 1;
8229             sz++;
8230             continue;
8231         }
8232         ASSERT(sz <= pszc);
8233         rootpp = PP_GROUPLEADER(pp, sz);
8234         for (sfhme = rootpp->p_mapping; sfhme; sfhme = tmphme) {
8235             tmphme = sfhme->hme_next;
8236             ASSERT(!IS_PAHME(sfhme));
8237             hmeblkp = sfmmu_hmetohblk(sfhme);
8238             if (hme_size(sfhme) != sz) {
8239                 continue;
8240             }
8241             if (hmeblkp->hblk_xhat_bit) {
8242                 cmn_err(CE_PANIC,
8243                     "hat_page_demote: xhat hmeblk");
8244             }
8245             tset = sfmmu_pageunload(rootpp, sfhme, sz);
8246             CPUSET_OR(cpuset, tset);
8247         }
8248         if (index >>= 1) {
8249             sz++;

```

```

8246     }
8247 }

8249     ASSERT(!PP_ISMAPPED_LARGE(pp));

8251     if (sync) {
8252         xt_sync(cpuset);
8253 #ifdef VAC
8254         if (PP_ISTNC(pp)) {
8255             conv_tnc(rootpp, sz);
8256         }
8257 #endif /* VAC */
8258     }

8260     pmtx = sfmmu_page_enter(pp);

8262     ASSERT(pp->p_szc == pszc);
8263     rootpp = PP_PAGEROOT(pp);
8264     ASSERT(rootpp->p_szc == pszc);
8265     lastpp = PP_PAGENEXT_N(rootpp, TTEPAGES(pszc) - 1);

8267     while (lastpp != rootpp) {
8268         sz = PP_MAPINDEX(lastpp) ? fnd_mapping_sz(lastpp) : 0;
8269         ASSERT(sz < pszc);
8270         npgs = (sz == 0) ? 1 : TTEPAGES(sz);
8271         ASSERT(P2PHASE(lastpp->p_pagenum, npgs) == npgs - 1);
8272         while (--npgs > 0) {
8273             lastpp->p_szc = (uchar_t)sz;
8274             lastpp = PP_PAGEPREV(lastpp);
8275         }
8276         if (sz) {
8277             /*
8278              * make sure before current root's pszc
8279              * is updated all updates to constituent pages pszc
8280              * fields are globally visible.
8281              */
8282             membar_producer();
8283         }
8284         lastpp->p_szc = sz;
8285         ASSERT(IS_P2ALIGNED(lastpp->p_pagenum, TTEPAGES(sz)));
8286         if (lastpp != rootpp) {
8287             lastpp = PP_PAGEPREV(lastpp);
8288         }
8289     }
8290     if (sz == 0) {
8291         /* the loop above doesn't cover this case */
8292         rootpp->p_szc = 0;
8293     }
8294 out:
8295     ASSERT(pp->p_szc == 0);
8296     if (pmtx != NULL) {
8297         sfmmu_page_exit(pmtx);
8298     }
8299     sfmmu_mlist_exit(pml);
8300 }

      unchanged_portion_omitted

8345 /*
8346  * Yield the memory claim requirement for an address space.
8347  *
8348  * This is currently implemented as the number of bytes that have active
8349  * hardware translations that have page structures. Therefore, it can
8350  * underestimate the traditional resident set size, eg, if the
8351  * physical page is present and the hardware translation is missing;
8352  * and it can overestimate the rss, eg, if there are active
8353  * translations to a frame buffer with page structs.

```

```

8354 * Also, it does not take sharing into account.
8355 *
8356 * Note that we don't acquire locks here since this function is most often
8357 * called from the clock thread.
8358 */
8359 size_t
8360 hat_get_mapped_size(struct hat *hat)
8361 {
8362     size_t    assize = 0;
8363     int       i;

8365     if (hat == NULL)
8366         return (0);

8519     ASSERT(hat->sfmmu_xhat_provider == NULL);

8368     for (i = 0; i < mmu_page_sizes; i++)
8369         assize += ((pgcnt_t)hat->sfmmu_ttecnt[i] +
8370                 (pgcnt_t)hat->sfmmu_scdrttecnt[i]) * TTEBYTES(i);

8372     if (hat->sfmmu_iblk == NULL)
8373         return (assize);

8375     for (i = 0; i < mmu_page_sizes; i++)
8376         assize += ((pgcnt_t)hat->sfmmu_ismttecnt[i] +
8377                 (pgcnt_t)hat->sfmmu_scdismttecnt[i]) * TTEBYTES(i);

8379     return (assize);
8380 }

8382 int
8383 hat_stats_enable(struct hat *hat)
8384 {
8385     hatlock_t    *hatlockp;

8540     ASSERT(hat->sfmmu_xhat_provider == NULL);

8387     hatlockp = sfmmu_hat_enter(hat);
8388     hat->sfmmu_rmstat++;
8389     sfmmu_hat_exit(hatlockp);
8390     return (1);
8391 }

8393 void
8394 hat_stats_disable(struct hat *hat)
8395 {
8396     hatlock_t    *hatlockp;

8553     ASSERT(hat->sfmmu_xhat_provider == NULL);

8398     hatlockp = sfmmu_hat_enter(hat);
8399     hat->sfmmu_rmstat--;
8400     sfmmu_hat_exit(hatlockp);
8401 }

    unchanged_portion_omitted_

8451 /*
8452 * Hat_share()/unshare() return an (non-zero) error
8453 * when saddr and daddr are not properly aligned.
8454 *
8455 * The top level mapping element determines the alignment
8456 * requirement for saddr and daddr, depending on different
8457 * architectures.
8458 *
8459 * When hat_share()/unshare() are not supported,
8460 * HATOP_SHARE()/UNSHARE() return 0

```

```

8461 */
8462 int
8463 hat_share(struct hat *sfmmup, caddr_t addr,
8464           struct hat *ism_hatid, caddr_t sptaddr, size_t len, uint_t ismszc)
8465 {
8466     ism_blk_t    *ism_blkp;
8467     ism_blk_t    *new_iblk;
8468     ism_map_t    *ism_map;
8469     ism_ment_t    *ism_ment;
8470     int          i, added;
8471     hatlock_t    *hatlockp;
8472     int          reload_mmu = 0;
8473     uint_t       ismshift = page_get_shift(ismszc);
8474     size_t       ismpgsz = page_get_pagesize(ismszc);
8475     uint_t       ismmask = (uint_t)ismpgsz - 1;
8476     size_t       sh_size = ISM_SHIFT(ismshift, len);
8477     ushort_t     ismhatflag;
8478     hat_region_cookie_t rcookie;
8479     sf_scd_t     *old_scdp;

8481 #ifdef DEBUG
8482     caddr_t      eaddr = addr + len;
8483 #endif /* DEBUG */

8485     ASSERT(ism_hatid != NULL && sfmmup != NULL);
8486     ASSERT(sptaddr == ISMID_STARTADDR);
8487     /*
8488      * Check the alignment.
8489      */
8490     if (!ISM_ALIGNED(ismshift, addr) || !ISM_ALIGNED(ismshift, sptaddr))
8491         return (EINVAL);

8493     /*
8494      * Check size alignment.
8495      */
8496     if (!ISM_ALIGNED(ismshift, len))
8497         return (EINVAL);

8656     ASSERT(sfmmup->sfmmu_xhat_provider == NULL);

8499     /*
8500      * Allocate ism_ment for the ism_hat's mapping list, and an
8501      * ism map blk in case we need one. We must do our
8502      * allocations before acquiring locks to prevent a deadlock
8503      * in the kmem allocator on the mapping list lock.
8504      */
8505     new_iblk = kmem_cache_alloc(ism_blk_cache, KM_SLEEP);
8506     ism_ment = kmem_cache_alloc(ism_ment_cache, KM_SLEEP);

8508     /*
8509      * Serialize ISM mappings with the ISM busy flag, and also the
8510      * trap handlers.
8511      */
8512     sfmmu_ismhat_enter(sfmmup, 0);

8514     /*
8515      * Allocate an ism map blk if necessary.
8516      */
8517     if (sfmmup->sfmmu_iblk == NULL) {
8518         sfmmup->sfmmu_iblk = new_iblk;
8519         bzero(new_iblk, sizeof(*new_iblk));
8520         new_iblk->iblk_nextpa = (uint64_t)-1;
8521         membar_stst(); /* make sure next ptr visible to all CPUs */
8522         sfmmup->sfmmu_ismblkpa = va_to_pa((caddr_t)new_iblk);
8523         reload_mmu = 1;
8524         new_iblk = NULL;

```

```

8525     }
8527 #ifdef DEBUG
8528     /*
8529     * Make sure mapping does not already exist.
8530     */
8531     ism_blkp = sfmmup->sfmmu_iblk;
8532     while (ism_blkp != NULL) {
8533         ism_map = ism_blkp->iblk_maps;
8534         for (i = 0; i < ISM_MAP_SLOTS && ism_map[i].imap_ismhat; i++) {
8535             if ((addr >= ism_start(ism_map[i]) &&
8536                 addr < ism_end(ism_map[i])) ||
8537                 eaddr > ism_start(ism_map[i]) &&
8538                 eaddr <= ism_end(ism_map[i])) {
8539                 panic("sfmmu_share: Already mapped!");
8540             }
8541         }
8542         ism_blkp = ism_blkp->iblk_next;
8543     }
8544 #endif /* DEBUG */

8546     ASSERT(ismszc >= TTE4M);
8547     if (ismszc == TTE4M) {
8548         ismhatflag = HAT_4M_FLAG;
8549     } else if (ismszc == TTE32M) {
8550         ismhatflag = HAT_32M_FLAG;
8551     } else if (ismszc == TTE256M) {
8552         ismhatflag = HAT_256M_FLAG;
8553     }
8554     /*
8555     * Add mapping to first available mapping slot.
8556     */
8557     ism_blkp = sfmmup->sfmmu_iblk;
8558     added = 0;
8559     while (!added) {
8560         ism_map = ism_blkp->iblk_maps;
8561         for (i = 0; i < ISM_MAP_SLOTS; i++) {
8562             if (ism_map[i].imap_ismhat == NULL) {

8564                 ism_map[i].imap_ismhat = ism_hatid;
8565                 ism_map[i].imap_vb_shift = (uchar_t)ismshift;
8566                 ism_map[i].imap_rid = SFMMU_INVALID_ISMRID;
8567                 ism_map[i].imap_hatflags = ismhatflag;
8568                 ism_map[i].imap_sz_mask = ismmask;
8569                 /*
8570                 * imap_seg is checked in ISM_CHECK to see if
8571                 * non-NULL, then other info assumed valid.
8572                 */
8573                 membar_stst();
8574                 ism_map[i].imap_seg = (uintptr_t)addr | sh_size;
8575                 ism_map[i].imap_ment = ism_ment;

8577                 /*
8578                 * Now add ourselves to the ism_hat's
8579                 * mapping list.
8580                 */
8581                 ism_ment->iment_hat = sfmmup;
8582                 ism_ment->iment_base_va = addr;
8583                 ism_hatid->sfmmu_ismhat = 1;
8584                 mutex_enter(&ism_mlist_lock);
8585                 iment_add(ism_ment, ism_hatid);
8586                 mutex_exit(&ism_mlist_lock);
8587                 added = 1;
8588                 break;
8589             }
8590         }

```

```

8591         if (!added && ism_blkp->iblk_next == NULL) {
8592             ism_blkp->iblk_next = new_iblk;
8593             new_iblk = NULL;
8594             bzero(ism_blkp->iblk_next,
8595                 sizeof (*ism_blkp->iblk_next));
8596             ism_blkp->iblk_next->iblk_nextpa = (uint64_t)-1;
8597             membar_stst();
8598             ism_blkp->iblk_nextpa =
8599                 va_to_pa((caddr_t)ism_blkp->iblk_next);
8600         }
8601         ism_blkp = ism_blkp->iblk_next;
8602     }

8604     /*
8605     * After calling hat_join_region, sfmmup may join a new SCD or
8606     * move from the old scd to a new scd, in which case, we want to
8607     * shrink the sfmmup's private tsb size, i.e., pass shrink to
8608     * sfmmu_check_page_sizes at the end of this routine.
8609     */
8610     old_scdp = sfmmup->sfmmu_scdp;

8612     rcookie = hat_join_region(sfmmup, addr, len, (void *)ism_hatid, 0,
8613         PROT_ALL, ismszc, NULL, HAT_REGION_ISM);
8614     if (rcookie != HAT_INVALID_REGION_COOKIE) {
8615         ism_map[i].imap_rid = (uchar_t)((uint64_t)rcookie);
8616     }
8617     /*
8618     * Update our counters for this sfmmup's ism mappings.
8619     */
8620     for (i = 0; i <= ismszc; i++) {
8621         if (!(disable_ism_large_pages & (1 << i)))
8622             (void) ism_tsb_entries(sfmmup, i);
8623     }

8625     /*
8626     * For ISM and DISM we do not support 512K pages, so we only only
8627     * search the 4M and 8K/64K hashes for 4 pagesize cpus, and search the
8628     * 256M or 32M, and 4M and 8K/64K hashes for 6 pagesize cpus.
8629     *
8630     * Need to set 32M/256M ISM flags to make sure
8631     * sfmmu_check_page_sizes() enables them on Panther.
8632     */
8633     ASSERT((disable_ism_large_pages & (1 << TTE512K)) != 0);

8635     switch (ismszc) {
8636     case TTE256M:
8637         if (!SFMMU_FLAGS_ISSET(sfmmup, HAT_256M_ISM)) {
8638             hatlockp = sfmmu_hat_enter(sfmmup);
8639             SFMMU_FLAGS_SET(sfmmup, HAT_256M_ISM);
8640             sfmmu_hat_exit(hatlockp);
8641         }
8642         break;
8643     case TTE32M:
8644         if (!SFMMU_FLAGS_ISSET(sfmmup, HAT_32M_ISM)) {
8645             hatlockp = sfmmu_hat_enter(sfmmup);
8646             SFMMU_FLAGS_SET(sfmmup, HAT_32M_ISM);
8647             sfmmu_hat_exit(hatlockp);
8648         }
8649         break;
8650     default:
8651         break;
8652     }

8654     /*
8655     * If we updated the ismblkpa for this HAT we must make
8656     * sure all CPUs running this process reload their tsbmiss area.

```

```

8657      * Otherwise they will fail to load the mappings in the tsbmiss
8658      * handler and will loop calling pagefault().
8659      */
8660      if (reload_mmu) {
8661          hatlockp = sfmmu_hat_enter(sfmmup);
8662          sfmmu_sync_mmustate(sfmmup);
8663          sfmmu_hat_exit(hatlockp);
8664      }

8666      sfmmu_ismhat_exit(sfmmup, 0);

8668      /*
8669      * Free up ismblk if we didn't use it.
8670      */
8671      if (new_iblk != NULL)
8672          kmem_cache_free(ism_blk_cache, new_iblk);

8674      /*
8675      * Check TSB and TLB page sizes.
8676      */
8677      if (sfmmup->sfmmu_scdp != NULL && old_scdp != sfmmup->sfmmu_scdp) {
8678          sfmmu_check_page_sizes(sfmmup, 0);
8679      } else {
8680          sfmmu_check_page_sizes(sfmmup, 1);
8681      }
8682      return (0);
8683 }

8685 /*
8686 * hat_unshare removes exactly one ism_map from
8687 * this process's as. It expects multiple calls
8688 * to hat_unshare for multiple shm segments.
8689 */
8690 void
8691 hat_unshare(struct hat *sfmmup, caddr_t addr, size_t len, uint_t ismszc)
8692 {
8693     ism_map_t      *ism_map;
8694     ism_ment_t     *free_ment = NULL;
8695     ism_blk_t      *ism_blkp;
8696     struct hat     *ism_hatid;
8697     int             found, i;
8698     hatlock_t      *hatlockp;
8699     struct tsb_info *tsbinfo;
8700     uint_t          ismshift = page_get_shift(ismszc);
8701     size_t          sh_size = ISM_SHIFT(ismshift, len);
8702     uchar_t         ism_rid;
8703     sf_scdp_t      *old_scdp;

8705     ASSERT(ISM_ALIGNED(ismshift, addr));
8706     ASSERT(ISM_ALIGNED(ismshift, len));
8707     ASSERT(sfmmup != NULL);
8708     ASSERT(sfmmup != ksfmmup);

8709     if (sfmmup->sfmmu_xhat_provider) {
8710         XHAT_UNSHARE(sfmmup, addr, len);
8711         return;
8712     } else {
8713         /*
8714          * This must be a CPU HAT. If the address space has
8715          * XHATs attached, inform all XHATs that ISM segment
8716          * is going away
8717          */
8718         ASSERT(sfmmup->sfmmu_as != NULL);
8719         if (sfmmup->sfmmu_as->a_xhat != NULL)
8720             xhat_unshare_all(sfmmup->sfmmu_as, addr, len);
8721     }

```

```

8712     /*
8713     * Make sure that during the entire time ISM mappings are removed,
8714     * the trap handlers serialize behind us, and that no one else
8715     * can be mucking with ISM mappings. This also lets us get away
8716     * with not doing expensive cross calls to flush the TLB -- we
8717     * just discard the context, flush the entire TSB, and call it
8718     * a day.
8719     */
8720     sfmmu_ismhat_enter(sfmmup, 0);

8722     /*
8723     * Remove the mapping.
8724     */
8725     * We can't have any holes in the ism map.
8726     * The tsb miss code while searching the ism map will
8727     * stop on an empty map slot. So we must move
8728     * everyone past the hole up 1 if any.
8729     */
8730     * Also empty ism map blks are not freed until the
8731     * process exits. This is to prevent a MT race condition
8732     * between sfmmu_unshare() and sfmmu_tsbmiss_exception().
8733     */
8734     found = 0;
8735     ism_blkp = sfmmup->sfmmu_iblk;
8736     while (!found && ism_blkp != NULL) {
8737         ism_map = ism_blkp->iblk_maps;
8738         for (i = 0; i < ISM_MAP_SLOTS; i++) {
8739             if (addr == ism_start(ism_map[i]) &&
8740                 sh_size == (size_t)(ism_size(ism_map[i]))) {
8741                 found = 1;
8742                 break;
8743             }
8744         }
8745         if (!found)
8746             ism_blkp = ism_blkp->iblk_next;
8747     }

8749     if (found) {
8750         ism_hatid = ism_map[i].ism_hatid;
8751         ism_rid = ism_map[i].ism_rid;
8752         ASSERT(ism_hatid != NULL);
8753         ASSERT(ism_hatid->sfmmu_ismhat == 1);

8755         /*
8756         * After hat_leave_region, the sfmmup may leave SCD,
8757         * in which case, we want to grow the private tsb size when
8758         * calling sfmmu_check_page_sizes at the end of the routine.
8759         */
8760         old_scdp = sfmmup->sfmmu_scdp;
8761         /*
8762         * Then remove ourselves from the region.
8763         */
8764         if (ism_rid != SFMMU_INVALID_ISMRID) {
8765             hat_leave_region(sfmmup, (void *)((uint64_t)ism_rid),
8766                 HAT_REGION_ISM);
8767         }

8769         /*
8770         * And now guarantee that any other cpu
8771         * that tries to process an ISM miss
8772         * will go to tl=0.
8773         */
8774         hatlockp = sfmmu_hat_enter(sfmmup);
8775         sfmmu_invalidate_ctx(sfmmup);
8776         sfmmu_hat_exit(hatlockp);

```

```

8778      /*
8779      * Remove ourselves from the ism mapping list.
8780      */
8781      mutex_enter(&ism_mlist_lock);
8782      iment_sub(ism_map[i].imap_ment, ism_hatid);
8783      mutex_exit(&ism_mlist_lock);
8784      free_ment = ism_map[i].imap_ment;

8786      /*
8787      * We delete the ism map by copying
8788      * the next map over the current one.
8789      * We will take the next one in the maps
8790      * array or from the next ism_blk.
8791      */
8792      while (ism_blkp != NULL) {
8793          ism_map = ism_blkp->iblk_maps;
8794          while (i < (ISM_MAP_SLOTS - 1)) {
8795              ism_map[i] = ism_map[i + 1];
8796              i++;
8797          }
8798          /* i == (ISM_MAP_SLOTS - 1) */
8799          ism_blkp = ism_blkp->iblk_next;
8800          if (ism_blkp != NULL) {
8801              ism_map[i] = ism_blkp->iblk_maps[0];
8802              i = 0;
8803          } else {
8804              ism_map[i].imap_seg = 0;
8805              ism_map[i].imap_vb_shift = 0;
8806              ism_map[i].imap_rid = SFMMU_INVALID_ISMRID;
8807              ism_map[i].imap_hatflags = 0;
8808              ism_map[i].imap_sz_mask = 0;
8809              ism_map[i].imap_ismhat = NULL;
8810              ism_map[i].imap_ment = NULL;
8811          }
8812      }

8814      /*
8815      * Now flush entire TSB for the process, since
8816      * demapping page by page can be too expensive.
8817      * We don't have to flush the TLB here anymore
8818      * since we switch to a new TLB ctx instead.
8819      * Also, there is no need to flush if the process
8820      * is exiting since the TSB will be freed later.
8821      */
8822      if (!sfmmup->sfmmu_free) {
8823          hatlockp = sfmmu_hat_enter(sfmmup);
8824          for (tsbinfo = sfmmup->sfmmu_tsb; tsbinfo != NULL;
8825              tsbinfo = tsbinfo->tsb_next) {
8826              if (tsbinfo->tsb_flags & TSB_SWAPPED)
8827                  continue;
8828              if (tsbinfo->tsb_flags & TSB_RELOC_FLAG) {
8829                  tsbinfo->tsb_flags |=
8830                      TSB_FLUSH_NEEDED;
8831                  continue;
8832              }

8834              sfmmu_inv_tsb(tsbinfo->tsb_va,
8835                          TSB_BYTES(tsbinfo->tsb_szc));
8836          }
8837          sfmmu_hat_exit(hatlockp);
8838      }
8839  }

8841      /*
8842      * Update our counters for this sfmmup's ism mappings.

```

```

8843      /*
8844      for (i = 0; i <= ismszc; i++) {
8845          if (!(disable_ism_large_pages & (1 << i)))
8846              (void) ism_tsb_entries(sfmmup, i);
8847      }

8849      sfmmu_ismhat_exit(sfmmup, 0);

8851      /*
8852      * We must do our freeing here after dropping locks
8853      * to prevent a deadlock in the kmem allocator on the
8854      * mapping list lock.
8855      */
8856      if (free_ment != NULL)
8857          kmem_cache_free(ism_ment_cache, free_ment);

8859      /*
8860      * Check TSB and TLB page sizes if the process isn't exiting.
8861      */
8862      if (!sfmmup->sfmmu_free) {
8863          if (found && old_scdp != NULL && sfmmup->sfmmu_scdp == NULL) {
8864              sfmmu_check_page_sizes(sfmmup, 1);
8865          } else {
8866              sfmmu_check_page_sizes(sfmmup, 0);
8867          }
8868      }
8869  }
_____unchanged_portion_omitted_____

9078 #ifdef VAC
9079 /*
9080 * Check for conflicts.
9081 * A conflict exists if the new and existent mappings do not match in
9082 * their "shm_alignment" fields. If conflicts exist, the existent mappings
9083 * are flushed unless one of them is locked. If one of them is locked, then
9084 * the mappings are flushed and converted to non-cacheable mappings.
9085 */
9086 static void
9087 sfmmu_vac_conflict(struct hat *hat, caddr_t addr, page_t *pp)
9088 {
9089     struct hat *tmphat;
9090     struct sf_hment *sfhmep, *tmphme = NULL;
9091     struct hme_blk *hmeblkp;
9092     int vcolor;
9093     tte_t tte;

9095     ASSERT(sfmmu_mlist_held(pp));
9096     ASSERT(!PP_ISNC(pp));      /* page better be cacheable */

9098     vcolor = addr_to_vcolor(addr);
9099     if (PP_NEWPAGE(pp)) {
9100         PP_SET_VCOLOR(pp, vcolor);
9101         return;
9102     }

9104     if (PP_GET_VCOLOR(pp) == vcolor) {
9105         return;
9106     }

9108     if (!PP_ISMAPPED(pp) && !PP_ISMAPPED_KPM(pp)) {
9109         /*
9110          * Previous user of page had a different color
9111          * but since there are no current users
9112          * we just flush the cache and change the color.
9113          */
9114         SFMMU_STAT(sf_pgcolor_conflict);

```



```

9115         sfmmu_cache_flush(pp->p_pagenum, PP_GET_VCOLOR(pp));
9116         PP_SET_VCOLOR(pp, vcolor);
9117         return;
9118     }

9120 /*
9121  * If we get here we have a vac conflict with a current
9122  * mapping. VAC conflict policy is as follows.
9123  * - The default is to unload the other mappings unless:
9124  * - If we have a large mapping we uncache the page.
9125  * We need to uncache the rest of the large page too.
9126  * - If any of the mappings are locked we uncache the page.
9127  * - If the requested mapping is inconsistent
9128  * with another mapping and that mapping
9129  * is in the same address space we have to
9130  * make it non-cached. The default thing
9131  * to do is unload the inconsistent mapping
9132  * but if they are in the same address space
9133  * we run the risk of unmapping the pc or the
9134  * stack which we will use as we return to the user,
9135  * in which case we can then fault on the thing
9136  * we just unloaded and get into an infinite loop.
9137  */
9138 if (PP_ISMAPPED_LARGE(pp)) {
9139     int sz;

9141     /*
9142     * Existing mapping is for big pages. We don't unload
9143     * existing big mappings to satisfy new mappings.
9144     * Always convert all mappings to TNC.
9145     */
9146     sz = fnd_mapping_sz(pp);
9147     pp = PP_GROUPLoader(pp, sz);
9148     SFMMU_STAT_ADD(sf_uncache_conflict, TTEPAGES(sz));
9149     sfmmu_page_cache_array(pp, HAT_TMPNC, CACHE_FLUSH,
9150         TTEPAGES(sz));

9152     return;
9153 }

9155 /*
9156  * check if any mapping is in same as or if it is locked
9157  * since in that case we need to uncache.
9158  */
9159 for (sfhmep = pp->p_mapping; sfhmep; sfhmep = tmphme) {
9160     tmphme = sfhmep->hme_next;
9161     if (IS_PAHME(sfhmep))
9162         continue;
9163     hmeblkp = sfmmu_hmetohblk(sfhmep);
9164     if (hmeblkp->hblk_xhat_bit)
9165         continue;
9166     tmphat = hblktosfmmu(hmeblkp);
9167     sfmmu_copytte(&sfhmep->hme_tte, &tte);
9168     ASSERT(TTE_IS_VALID(&tte));
9169     if (hmeblkp->hblk_shared || tmphat == hat ||
9170         hmeblkp->hblk_lckcnt) {
9171         /*
9172         * We have an uncache conflict
9173         */
9174         SFMMU_STAT(sf_uncache_conflict);
9175         sfmmu_page_cache_array(pp, HAT_TMPNC, CACHE_FLUSH, 1);
9176         return;
9177     }
9178 }

```

```

9179     * We have an unload conflict
9180     * We have already checked for LARGE mappings, therefore
9181     * the remaining mapping(s) must be TTE8K.
9182     */
9183     SFMMU_STAT(sf_unload_conflict);

9185     for (sfhmep = pp->p_mapping; sfhmep; sfhmep = tmphme) {
9186         tmphme = sfhmep->hme_next;
9187         if (IS_PAHME(sfhmep))
9188             continue;
9189         hmeblkp = sfmmu_hmetohblk(sfhmep);
9190         if (hmeblkp->hblk_xhat_bit)
9191             continue;
9192         ASSERT(!hmeblkp->hblk_shared);
9193         (void) sfmmu_pageunload(pp, sfhmep, TTE8K);
9194     }

9196     if (PP_ISMAPPED_KPM(pp))
9197         sfmmu_kpm_vac_unload(pp, addr);

9199     /*
9200     * Unloads only do TLB flushes so we need to flush the
9201     * cache here.
9202     */
9203     sfmmu_cache_flush(pp->p_pagenum, PP_GET_VCOLOR(pp));
9204     PP_SET_VCOLOR(pp, vcolor);
9205 }

```

unchanged portion omitted

```

9290 /*
9291  * Returns 1 if page(s) can be converted from TNC to cacheable setting,
9292  * returns 0 otherwise. Note that oaddr argument is valid for only
9293  * 8k pages.
9294  */
9295 int
9296 tst_tnc(page_t *pp, pgcnt_t npages)
9297 {
9298     struct sf_hment *sfhme;
9299     struct hme_blk *hmeblkp;
9300     tte_t tte;
9301     caddr_t vaddr;
9302     int clr_valid = 0;
9303     int color, color1, bcolor;
9304     int i, ncolors;

9306     ASSERT(pp != NULL);
9307     ASSERT(!(cache & CACHE_WRITEBACK));

9309     if (npages > 1) {
9310         ncolors = CACHE_NUM_COLOR;
9311     }

9313     for (i = 0; i < npages; i++) {
9314         ASSERT(sfmmu_mlist_held(pp));
9315         ASSERT(PP_ISTNC(pp));
9316         ASSERT(PP_GET_VCOLOR(pp) == NO_VCOLOR);

9318         if (PP_ISPNC(pp)) {
9319             return (0);
9320         }

9322         clr_valid = 0;
9323         if (PP_ISMAPPED_KPM(pp)) {
9324             caddr_t kpmvaddr;

9326             ASSERT(kpm_enable);

```

```

9327         kpmvaddr = hat_kpm_page2va(pp, 1);
9328         ASSERT(!(npages > 1 && IS_KPM_ALIAS_RANGE(kpmvaddr)));
9329         color1 = addr_to_vcolor(kpmvaddr);
9330         clr_valid = 1;
9331     }

9333     for (sfhme = pp->p_mapping; sfhme; sfhme = sfhme->hme_next) {
9334         if (IS_PAHME(sfhme))
9335             continue;
9336         hmeblkp = sfmmu_hmetohblk(sfhme);
9352         if (hmeblkp->hblk_xhat_bit)
9353             continue;

9338         sfmmu_copytte(&sfhme->hme_tte, &tte);
9339         ASSERT(TTE_IS_VALID(&tte));

9341         vaddr = tte_to_vaddr(hmeblkp, tte);
9342         color = addr_to_vcolor(vaddr);

9344         if (npages > 1) {
9345             /*
9346              * If there is a big mapping, make sure
9347              * 8K mapping is consistent with the big
9348              * mapping.
9349              */
9350             bcolor = i % ncolors;
9351             if (color != bcolor) {
9352                 return (0);
9353             }
9354         }
9355         if (!clr_valid) {
9356             clr_valid = 1;
9357             color1 = color;
9358         }

9360         if (color1 != color) {
9361             return (0);
9362         }
9363     }

9365     pp = PP_PAGENEXT(pp);
9366 }

9368     return (1);
9369 }

```

unchanged portion omitted

```

9455 /*
9456  * This function changes the virtual cacheability of all mappings to a
9457  * particular page.  When changing from uncached to cacheable the mappings will
9458  * only be changed if all of them have the same virtual color.
9459  * We need to flush the cache in all cpus.  It is possible that
9460  * a process referenced a page as cacheable but has since exited
9461  * and cleared the mapping list.  We still to flush it but have no
9462  * state so all cpus is the only alternative.
9463  */
9464 static void
9465 sfmmu_page_cache(page_t *pp, int flags, int cache_flush_flag, int bcolor)
9466 {
9467     struct sf_hment *sfhme;
9468     struct hme_blk *hmeblkp;
9469     sfmmu_t *sfmmup;
9470     tte_t tte, ttemod;
9471     caddr_t vaddr;
9472     int ret, color;
9473     pfn_t pfn;

```

```

9475         color = bcolor;
9476         pfn = pp->p_pagenum;

9478         for (sfhme = pp->p_mapping; sfhme; sfhme = sfhme->hme_next) {

9480             if (IS_PAHME(sfhme))
9481                 continue;
9482             hmeblkp = sfmmu_hmetohblk(sfhme);

9661             if (hmeblkp->hblk_xhat_bit)
9662                 continue;

9484             sfmmu_copytte(&sfhme->hme_tte, &tte);
9485             ASSERT(TTE_IS_VALID(&tte));
9486             vaddr = tte_to_vaddr(hmeblkp, tte);
9487             color = addr_to_vcolor(vaddr);

9489 #ifdef DEBUG
9490             if ((flags & HAT_CACHE) && bcolor != NO_VCOLOR) {
9491                 ASSERT(color == bcolor);
9492             }
9493 #endif

9495             ASSERT(flags != HAT_TMPNC || color == PP_GET_VCOLOR(pp));

9497             ttemod = tte;
9498             if (flags & (HAT_UNCACHE | HAT_TMPNC)) {
9499                 TTE_CLR_VCACHEABLE(&ttemod);
9500             } else { /* flags & HAT_CACHE */
9501                 TTE_SET_VCACHEABLE(&ttemod);
9502             }
9503             ret = sfmmu_modifytte_try(&tte, &ttemod, &sfhme->hme_tte);
9504             if (ret < 0) {
9505                 /*
9506                  * Since all cpus are captured modifytte should not
9507                  * fail.
9508                  */
9509                 panic("sfmmu_page_cache: write to tte failed");
9510             }

9512             sfmmup = hblkto_sfmmu(hmeblkp);
9513             if (cache_flush_flag == CACHE_FLUSH) {
9514                 /*
9515                  * Flush TSBS, TLBs and caches
9516                  */
9517                 if (hmeblkp->hblk_shared) {
9518                     sf_srd_t *srdp = (sf_srd_t *)sfmmup;
9519                     uint_t rid = hmeblkp->hblk_tag.htag_rid;
9520                     sf_region_t *rgnp;
9521                     ASSERT(SFMMU_IS_SHMERID_VALID(rid));
9522                     ASSERT(rid < SFMMU_MAX_HME_REGIONS);
9523                     ASSERT(srdp != NULL);
9524                     rgnp = srdp->srd_hmergnp[rid];
9525                     SFMMU_VALIDATE_SHAREDHBLK(hmeblkp,
9526                                                srdp, rgnp, rid);
9527                     (void) sfmmu_rgntlb_demap(vaddr, rgnp,
9528                                               hmeblkp, 0);
9529                     sfmmu_cache_flush(pfn, addr_to_vcolor(vaddr));
9530                 } else if (sfmmup->sfmmu_ismhat) {
9531                     if (flags & HAT_CACHE) {
9532                         SFMMU_STAT(sf_ism_recache);
9533                     } else {
9534                         SFMMU_STAT(sf_ism_uncache);
9535                     }
9536                 }
9537             }
9538             sfmmu_ismtlb_cache_demap(vaddr, sfmmup, hmeblkp,

```

```

9537         pfn, CACHE_FLUSH);
9538     } else {
9539         sfmmu_tlbcache_demap(vaddr, sfmmup, hmeblkp,
9540         pfn, 0, FLUSH_ALL_CPUS, CACHE_FLUSH, 1);
9541     }
9542
9543     /*
9544     * all cache entries belonging to this pfn are
9545     * now flushed.
9546     */
9547     cache_flush_flag = CACHE_NO_FLUSH;
9548 } else {
9549     /*
9550     * Flush only TSBs and TLBs.
9551     */
9552     if (hmeblkp->hblk_shared) {
9553         sf_srd_t *srdp = (sf_srd_t *)sfmmup;
9554         uint_t rid = hmeblkp->hblk_tag.htag_rid;
9555         sf_region_t *rgnp;
9556         ASSERT(SFMMU_IS_SHMERID_VALID(rid));
9557         ASSERT(rid < SFMMU_MAX_HME_REGIONS);
9558         ASSERT(srdp != NULL);
9559         rgnp = srdp->srd_hmergnp[rid];
9560         SFMMU_VALIDATE_SHAREDHBLK(hmeblkp,
9561         srdp, rgnp, rid);
9562         (void) sfmmu_rgntlb_demap(vaddr, rgnp,
9563         hmeblkp, 0);
9564     } else if (sfmmup->sfmmu_ismhat) {
9565         if (flags & HAT_CACHE) {
9566             SFMMU_STAT(sf_ism_recache);
9567         } else {
9568             SFMMU_STAT(sf_ism_uncache);
9569         }
9570         sfmmu_ismtlbcache_demap(vaddr, sfmmup, hmeblkp,
9571         pfn, CACHE_NO_FLUSH);
9572     } else {
9573         sfmmu_tlb_demap(vaddr, sfmmup, hmeblkp, 0, 1);
9574     }
9575 }
9576 }
9577
9578 if (PP_ISMAPPED_KPM(pp))
9579     sfmmu_kpm_page_cache(pp, flags, cache_flush_flag);
9580
9581 switch (flags) {
9582     default:
9583         panic("sfmmu_pagecache: unknown flags");
9584         break;
9585
9586     case HAT_CACHE:
9587         PP_CLRTNC(pp);
9588         PP_CLRPNC(pp);
9589         PP_SET_VCOLOR(pp, color);
9590         break;
9591
9592     case HAT_TMPNC:
9593         PP_SETTNC(pp);
9594         PP_SET_VCOLOR(pp, NO_VCOLOR);
9595         break;
9596
9597     case HAT_UNCACHE:
9598         PP_SETPNC(pp);
9599         PP_CLRTNC(pp);
9600         PP_SET_VCOLOR(pp, NO_VCOLOR);
9601         break;
9602

```

```

9603     }
9604 }
9605
9606 unchanged_portion_omitted
9607
9608 13222 /*
9609 13223 * This function is currently not supported on this platform. For what
9610 13224 * it's supposed to do, see hat.c and hat_srmmu.c
9611 13225 */
9612 13226 /* ARGSUSED */
9613 13227 faultcode_t
9614 13228 hat_softlock(struct hat *hat, caddr_t addr, size_t *lenp, page_t **ppp,
9615 13229         uint_t flags)
9616 13230 {
9617 13411     ASSERT(hat->sfmmu_xhat_provider == NULL);
9618 13231     return (FC_NOSUPPORT);
9619 13232 }
9620
9621 13234 /*
9622 13235 * Searches the mapping list of the page for a mapping of the same size. If not
9623 13236 * found the corresponding bit is cleared in the p_index field. When large
9624 13237 * pages are more prevalent in the system, we can maintain the mapping list
9625 13238 * in order and we don't have to traverse the list each time. Just check the
9626 13239 * next and prev entries, and if both are of different size, we clear the bit.
9627 13240 */
9628 13241 static void
9629 13242 sfmmu_rm_large_mappings(page_t *pp, int ttesz)
9630 13243 {
9631 13244     struct sfhmep *sfhmep;
9632 13245     struct hme_blk *hmeblkp;
9633 13246     int index;
9634 13247     pgcnt_t npgs;
9635
9636 13249     ASSERT(ttesz > TTE8K);
9637
9638 13251     ASSERT(sfmmu_mlist_held(pp));
9639
9640 13253     ASSERT(PP_ISMAPPED_LARGE(pp));
9641
9642 13255     /*
9643 13256     * Traverse mapping list looking for another mapping of same size.
9644 13257     * since we only want to clear index field if all mappings of
9645 13258     * that size are gone.
9646 13259     */
9647
9648 13261     for (sfhmep = pp->p_mapping; sfhmep; sfhmep = sfhmep->hme_next) {
9649 13262         if (IS_PAHME(sfhmep))
9650 13263             continue;
9651 13264         hmeblkp = sfmmu_hmetohblk(sfhmep);
9652 13446         if (hmeblkp->hblk_xhat_bit)
9653 13447             continue;
9654 13265         if (hme_size(sfhmep) == ttesz) {
9655 13266             /*
9656 13267             * another mapping of the same size. don't clear index.
9657 13268             */
9658 13269             return;
9659 13270         }
9660 13271     }
9661
9662 13273     /*
9663 13274     * Clear the p_index bit for large page.
9664 13275     */
9665 13276     index = PAGESZ_TO_INDEX(ttesz);
9666 13277     npgs = TTEPAGES(ttesz);
9667 13278     while (npgs-- > 0) {
9668 13279         ASSERT(pp->p_index & index);
9669 13280         pp->p_index &= ~index;

```

```

13281         pp = PP_PAGENEXT(pp);
13282     }
13283 }
_____unchanged_portion_omitted_____
13798 /*
13799 * The caller makes sure hat_join_region()/hat_leave_region() can't be called
13800 * at the same time for the same process and address range. This is ensured by
13801 * the fact that address space is locked as writer when a process joins the
13802 * regions. Therefore there's no need to hold an srd lock during the entire
13803 * execution of hat_join_region()/hat_leave_region().
13804 */

13806 #define RGN_HASH_FUNCTION(obj) (((uintptr_t)(obj)) >> 4) ^ \
13807                               (((uintptr_t)(obj)) >> 11) & \
13808                               srd_rgn_hashmask)
13809 /*
13810 * This routine implements the shared context functionality required when
13811 * attaching a segment to an address space. It must be called from
13812 * hat_share() for D(ISM) segments and from segvn_create() for segments
13813 * with the MAP_PRIVATE and MAP_TEXT flags set. It returns a region_cookie
13814 * which is saved in the private segment data for hme segments and
13815 * the ism_map structure for ism segments.
13816 */
13817 hat_region_cookie_t
13818 hat_join_region(struct hat *sfmmup,
13819                caddr_t r_saddr,
13820                size_t r_size,
13821                void *r_obj,
13822                u_offset_t r_objoff,
13823                uchar_t r_perm,
13824                uchar_t r_pgsz,
13825                hat_rgn_cb_func_t r_cb_function,
13826                uint_t flags)
13827 {
13828     sf_srd_t *srdp = sfmmup->sfmmu_srdp;
13829     uint_t rhash;
13830     uint_t rid;
13831     hatlock_t *hatlockp;
13832     sf_region_t *rgnp;
13833     sf_region_t *new_rgnp = NULL;
13834     int i;
13835     uint16_t *nextidp;
13836     sf_region_t **freelistp;
13837     int maxids;
13838     sf_region_t **rarrp;
13839     uint16_t *busyrgnsp;
13840     ulong_t rttecnt;
13841     uchar_t tteflag;
13842     uchar_t r_type = flags & HAT_REGION_TYPE_MASK;
13843     int text = (r_type == HAT_REGION_TEXT);

13845     if (srdp == NULL || r_size == 0) {
13846         return (HAT_INVALID_REGION_COOKIE);
13847     }

14032     ASSERT(sfmmup->sfmmu_xhat_provider == NULL);
13849     ASSERT(sfmmup != ksffmmup);
13850     ASSERT(AS_WRITE_HELD(sfmmup->sfmmu_as, &sfmmup->sfmmu_as->a_lock));
13851     ASSERT(srdp->srd_refcnt > 0);
13852     ASSERT(!(flags & ~HAT_REGION_TYPE_MASK));
13853     ASSERT(flags == HAT_REGION_TEXT || flags == HAT_REGION_ISM);
13854     ASSERT(r_pgsz < mmu_page_sizes);
13855     if (!IS_P2ALIGNED(r_saddr, TTEBYTES(r_pgsz)) ||
13856         !IS_P2ALIGNED(r_size, TTEBYTES(r_pgsz))) {
13857         panic("hat_join_region: region addr or size is not aligned\n");

```

```

13858     }

13861     r_type = (r_type == HAT_REGION_ISM) ? SFMMU_REGION_ISM :
13862           SFMMU_REGION_HME;
13863     /*
13864     * Currently only support shared hmes for the read only main text
13865     * region.
13866     */
13867     if (r_type == SFMMU_REGION_HME && ((r_obj != srdp->srd_esp) ||
13868         (r_perm & PROT_WRITE))) {
13869         return (HAT_INVALID_REGION_COOKIE);
13870     }

13872     rhash = RGN_HASH_FUNCTION(r_obj);

13874     if (r_type == SFMMU_REGION_ISM) {
13875         nextidp = &srdp->srd_next_ismrid;
13876         freelistp = &srdp->srd_ismrgnfree;
13877         maxids = SFMMU_MAX_ISM_REGIONS;
13878         rarrp = srdp->srd_ismrgnp;
13879         busyrgnsp = &srdp->srd_ismbusyrgns;
13880     } else {
13881         nextidp = &srdp->srd_next_hmerid;
13882         freelistp = &srdp->srd_hmergnfree;
13883         maxids = SFMMU_MAX_HME_REGIONS;
13884         rarrp = srdp->srd_hmergnp;
13885         busyrgnsp = &srdp->srd_hmebusyrgns;
13886     }

13888     mutex_enter(&srdp->srd_mutex);

13890     for (rgnp = srdp->srd_rgnhash[rhash]; rgnp != NULL;
13891         rgnp = rgnp->rgn_hash) {
13892         if (rgnp->rgn_saddr == r_saddr && rgnp->rgn_size == r_size &&
13893             rgnp->rgn_obj == r_obj && rgnp->rgn_objoff == r_objoff &&
13894             rgnp->rgn_perm == r_perm && rgnp->rgn_pgsz == r_pgsz) {
13895             break;
13896         }
13897     }

13899     rfound:
13900     if (rgnp != NULL) {
13901         ASSERT((rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK) == r_type);
13902         ASSERT(rgnp->rgn_cb_function == r_cb_function);
13903         ASSERT(rgnp->rgn_refcnt >= 0);
13904         rid = rgnp->rgn_id;
13905         ASSERT(rid < maxids);
13906         ASSERT(rarrp[rid] == rgnp);
13907         ASSERT(rid < *nextidp);
13908         atomic_inc_32((volatile uint_t *)&rgnp->rgn_refcnt);
13909         mutex_exit(&srdp->srd_mutex);
13910         if (new_rgnp != NULL) {
13911             kmem_cache_free(region_cache, new_rgnp);
13912         }
13913         if (r_type == SFMMU_REGION_HME) {
13914             int myjoin =
13915                 (sfmmup == astosfmmu(curthread->t_procp->p_as));

13917             sfmmu_link_to_hmeregion(sfmmup, rgnp);
13918             /*
13919             * bitmap should be updated after linking sfmmu on
13920             * region list so that pageunload() doesn't skip
13921             * TSB/TLB flush. As soon as bitmap is updated another
13922             * thread in this process can already start accessing
13923             * this region.

```

```

13924  */
13925  /*
13926  * Normally ttecnt accounting is done as part of
13927  * pagefault handling. But a process may not take any
13928  * pagefaults on shared hmeblks created by some other
13929  * process. To compensate for this assume that the
13930  * entire region will end up faulted in using
13931  * the region's pagesize.
13932  *
13933  */
13934  if (r_pgsz > TTE8K) {
13935      tteflag = 1 << r_pgsz;
13936      if (disable_large_pages & tteflag) {
13937          tteflag = 0;
13938      }
13939  } else {
13940      tteflag = 0;
13941  }
13942  if (tteflag && !(sfmmup->sfmmu_rtteflags & tteflag)) {
13943      hatlockp = sfmmu_hat_enter(sfmmup);
13944      sfmmup->sfmmu_rtteflags |= tteflag;
13945      sfmmu_hat_exit(hatlockp);
13946  }
13947  hatlockp = sfmmu_hat_enter(sfmmup);
13948
13949  /*
13950  * Preallocate 1/4 of ttecnt's in 8K TSB for >= 4M
13951  * region to allow for large page allocation failure.
13952  */
13953  if (r_pgsz >= TTE4M) {
13954      sfmmup->sfmmu_tsb0_4minflcnt +=
13955          r_size >> (TTE_PAGE_SHIFT(TTE8K) + 2);
13956  }
13957
13958  /* update sfmmu_ttecnt with the shme rgn ttecnt */
13959  rttecnt = r_size >> TTE_PAGE_SHIFT(r_pgsz);
13960  atomic_add_long(&sfmmup->sfmmu_ttecnt[r_pgsz],
13961      rttecnt);
13962
13963  if (text && r_pgsz >= TTE4M &&
13964      (tteflag || ((disable_large_pages >> TTE4M) &
13965          ((1 << (r_pgsz - TTE4M + 1)) - 1))) &&
13966      !SFMMU_FLAGS_ISSET(sfmmup, HAT_4MTEXT_FLAG)) {
13967      SFMMU_FLAGS_SET(sfmmup, HAT_4MTEXT_FLAG);
13968  }
13969
13970  sfmmu_hat_exit(hatlockp);
13971  /*
13972  * On Panther we need to make sure TLB is programmed
13973  * to accept 32M/256M pages. Call
13974  * sfmmu_check_page_sizes() now to make sure TLB is
13975  * setup before making hmeregions visible to other
13976  * threads.
13977  */
13978  sfmmu_check_page_sizes(sfmmup, 1);
13979  hatlockp = sfmmu_hat_enter(sfmmup);
13980  SF_RGNMAP_ADD(sfmmup->sfmmu_hmeregion_map, rid);
13981
13982  /*
13983  * if context is invalid tsb miss exception code will
13984  * call sfmmu_check_page_sizes() and update tsbmiss
13985  * area later.
13986  */
13987  kpreempt_disable();
13988  if (myjoin &&
13989      (sfmmup->sfmmu_ctxs[CPU_MMU_IDX(CPU)].cnun

```

```

13990      != INVALID_CONTEXT)) {
13991          struct tsbmiss *tsbmp;
13992
13993          tsbmp = &tsbmiss_area[CPU->cpu_id];
13994          ASSERT(sfmmup == tsbmp->usfmmup);
13995          BT_SET(tsbmp->shmermap, rid);
13996          if (r_pgsz > TTE64K) {
13997              tsbmp->uhat_rtteflags |= tteflag;
13998          }
13999      }
14000  }
14001  kpreempt_enable();
14002
14003  sfmmu_hat_exit(hatlockp);
14004  ASSERT((hat_region_cookie_t)((uint64_t)rid) !=
14005      HAT_INVALID_REGION_COOKIE);
14006  } else {
14007      hatlockp = sfmmu_hat_enter(sfmmup);
14008      SF_RGNMAP_ADD(sfmmup->sfmmu_ismregion_map, rid);
14009      sfmmu_hat_exit(hatlockp);
14010  }
14011  ASSERT(rid < maxids);
14012
14013  if (r_type == SFMMU_REGION_ISM) {
14014      sfmmu_find_scd(sfmmup);
14015  }
14016  return ((hat_region_cookie_t)((uint64_t)rid));
14017  }
14018
14019  ASSERT(new_rgnp == NULL);
14020
14021  if (*busyrngnp >= maxids) {
14022      mutex_exit(&srddp->srd_mutex);
14023      return (HAT_INVALID_REGION_COOKIE);
14024  }
14025
14026  ASSERT(MUTEX_HELD(&srddp->srd_mutex));
14027  if (*freelistp != NULL) {
14028      rgnp = *freelistp;
14029      *freelistp = rgnp->rgn_next;
14030      ASSERT(rgnp->rgn_id < *nextidp);
14031      ASSERT(rgnp->rgn_id < maxids);
14032      ASSERT(rgnp->rgn_flags & SFMMU_REGION_FREE);
14033      ASSERT((rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK)
14034          == r_type);
14035      ASSERT(rarrp[rgnp->rgn_id] == rgnp);
14036      ASSERT(rgnp->rgn_hmeflags == 0);
14037  } else {
14038      /*
14039      * release local locks before memory allocation.
14040      */
14041      mutex_exit(&srddp->srd_mutex);
14042
14043      new_rgnp = kmem_cache_alloc(region_cache, KM_SLEEP);
14044
14045      mutex_enter(&srddp->srd_mutex);
14046      for (rgnp = srddp->srd_rgnhash[rhash]; rgnp != NULL;
14047          rgnp = rgnp->rgn_hash) {
14048          if (rgnp->rgn_saddr == r_saddr &&
14049              rgnp->rgn_size == r_size &&
14050              rgnp->rgn_obj == r_obj &&
14051              rgnp->rgn_objoff == r_objoff &&
14052              rgnp->rgn_perm == r_perm &&
14053              rgnp->rgn_pgsz == r_pgsz) {
14054              break;
14055          }

```

```

14056     }
14057     if (rgnp != NULL) {
14058         goto rfound;
14059     }

14061     if (*nextidp >= maxids) {
14062         mutex_exit(&srdp->srdd_mutex);
14063         goto fail;
14064     }
14065     rgnp = new_rgnp;
14066     new_rgnp = NULL;
14067     rgnp->rgn_id = (*nextidp)++;
14068     ASSERT(rgnp->rgn_id < maxids);
14069     ASSERT(rarrp[rgnp->rgn_id] == NULL);
14070     rarrp[rgnp->rgn_id] = rgnp;
14071 }

14073     ASSERT(rgnp->rgn_sfmmu_head == NULL);
14074     ASSERT(rgnp->rgn_hmeflags == 0);
14075 #ifdef DEBUG
14076     for (i = 0; i < MMU_PAGE_SIZES; i++) {
14077         ASSERT(rgnp->rgn_ttecnt[i] == 0);
14078     }
14079 #endif
14080     rgnp->rgn_saddr = r_saddr;
14081     rgnp->rgn_size = r_size;
14082     rgnp->rgn_obj = r_obj;
14083     rgnp->rgn_objoff = r_objoff;
14084     rgnp->rgn_perm = r_perm;
14085     rgnp->rgn_pgsz = r_pgsz;
14086     rgnp->rgn_flags = r_type;
14087     rgnp->rgn_refcnt = 0;
14088     rgnp->rgn_cb_function = r_cb_function;
14089     rgnp->rgn_hash = srdp->srdd_rgnhash[rhash];
14090     srdp->srdd_rgnhash[rhash] = rgnp;
14091     (*busyrgnsp)++;
14092     ASSERT(*busyrgnsp <= maxids);
14093     goto rfound;

14095 fail:
14096     ASSERT(new_rgnp != NULL);
14097     kmem_cache_free(region_cache, new_rgnp);
14098     return (HAT_INVALID_REGION_COOKIE);
14099 }

14101 /*
14102 * This function implements the shared context functionality required
14103 * when detaching a segment from an address space. It must be called
14104 * from hat_unshare() for all D(ISM) segments and from segvn_unmap(),
14105 * for segments with a valid region_cookie.
14106 * It will also be called from all segvn routines which change a
14107 * segment's attributes such as segvn_setprot(), segvn_setpagesize(),
14108 * segvn_clrsize() & segvn_advise(), as well as in the case of COW fault
14109 * from segvn_fault().
14110 */
14111 void
14112 hat_leave_region(struct hat *sfmmup, hat_region_cookie_t rcookie, uint_t flags)
14113 {
14114     sf_srd_t *srdp = sfmmup->sfmmu_srdp;
14115     sf_scd_t *scdp;
14116     uint_t rhash;
14117     uint_t rid = (uint_t)((uint64_t)rcookie);
14118     hatlock_t *hatlockp = NULL;
14119     sf_region_t *rgnp;
14120     sf_region_t **prev_rgnpp;
14121     sf_region_t *cur_rgnp;

```

```

14122     void *r_obj;
14123     int i;
14124     caddr_t r_saddr;
14125     caddr_t r_eaddr;
14126     size_t r_size;
14127     uchar_t r_pgsz;
14128     uchar_t r_type = flags & HAT_REGION_TYPE_MASK;

14130     ASSERT(sfmmup != ksfsmmup);
14131     ASSERT(srdp != NULL);
14132     ASSERT(srdp->srdd_refcnt > 0);
14133     ASSERT(!(flags & ~HAT_REGION_TYPE_MASK));
14134     ASSERT(flags == HAT_REGION_TEXT || flags == HAT_REGION_ISM);
14135     ASSERT(!sfmmup->sfmmu_free || sfmmup->sfmmu_scdp == NULL);

14137     r_type = (r_type == HAT_REGION_ISM) ? SFMMU_REGION_ISM :
14138         SFMMU_REGION_HME;

14140     if (r_type == SFMMU_REGION_ISM) {
14141         ASSERT(SFMMU_IS_ISMRID_VALID(rid));
14142         ASSERT(rid < SFMMU_MAX_ISM_REGIONS);
14143         rgnp = srdp->srdd_ismrgnp[rid];
14144     } else {
14145         ASSERT(SFMMU_IS_SHMERID_VALID(rid));
14146         ASSERT(rid < SFMMU_MAX_HME_REGIONS);
14147         rgnp = srdp->srdd_hmergnp[rid];
14148     }
14149     ASSERT(rgnp != NULL);
14150     ASSERT(rgnp->rgn_id == rid);
14151     ASSERT((rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK) == r_type);
14152     ASSERT(!(rgnp->rgn_flags & SFMMU_REGION_FREE));
14153     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     }

14155     if (sfmmup->sfmmu_free) {
14156         ulong_t rttecnt;
14157         r_pgsz = rgnp->rgn_pgsz;
14158         r_size = rgnp->rgn_size;

14160         ASSERT(sfmmup->sfmmu_scdp == NULL);
14161         if (r_type == SFMMU_REGION_ISM) {
14162             SF_RGNMAP_DEL(sfmmup->sfmmu_ismregion_map, rid);
14163         } else {
14164             /* update shme rgns ttecnt in sfmmu_ttecnt */
14165             rttecnt = r_size >> TTE_PAGE_SHIFT(r_pgsz);
14166             ASSERT(sfmmup->sfmmu_ttecnt[r_pgsz] >= rttecnt);

14168             atomic_add_long(&sfmmup->sfmmu_ttecnt[r_pgsz],
14169                 -rttecnt);

14171             SF_RGNMAP_DEL(sfmmup->sfmmu_hmeregion_map, rid);
14172         }
14173     } else if (r_type == SFMMU_REGION_ISM) {
14174         hatlockp = sfmmu_hat_enter(sfmmup);
14175         ASSERT(rid < srdp->srdd_next_ismrid);
14176         SF_RGNMAP_DEL(sfmmup->sfmmu_ismregion_map, rid);
14177         scdp = sfmmup->sfmmu_scdp;
14178         if (scdp != NULL &&
14179             SF_RGNMAP_TEST(scdp->scdd_ismregion_map, rid)) {
14180             sfmmu_leave_scd(sfmmup, r_type);
14181             ASSERT(sfmmu_hat_lock_held(sfmmup));

```

```

14182     }
14183     sfmmu_hat_exit(hatlockp);
14184 } else {
14185     ulong_t rttecnt;
14186     r_pgsz = rgnp->rgn_pgsz;
14187     r_saddr = rgnp->rgn_saddr;
14188     r_size = rgnp->rgn_size;
14189     r_eaddr = r_saddr + r_size;

14191     ASSERT(r_type == SFMMU_REGION_HME);
14192     hatlockp = sfmmu_hat_enter(sfmmup);
14193     ASSERT(rid < srdp->srd_next_hmerid);
14194     SF_RGNMAP_DEL(sfmmup->sfmmu_hmeregion_map, rid);

14196     /*
14197     * If region is part of an SCD call sfmmu_leave_scd().
14198     * Otherwise if process is not exiting and has valid context
14199     * just drop the context on the floor to lose stale TLB
14200     * entries and force the update of tsb miss area to reflect
14201     * the new region map. After that clean our TSB entries.
14202     */
14203     scdp = sfmmup->sfmmu_scdp;
14204     if (scdp != NULL &&
14205         SF_RGNMAP_TEST(scdp->scd_hmeregion_map, rid)) {
14206         sfmmu_leave_scd(sfmmup, r_type);
14207         ASSERT(sfmmu_hat_lock_held(sfmmup));
14208     }
14209     sfmmu_invalidate_ctx(sfmmup);

14211     i = TTE8K;
14212     while (i < mmu_page_sizes) {
14213         if (rgnp->rgn_ttecnt[i] != 0) {
14214             sfmmu_unload_tsb_range(sfmmup, r_saddr,
14215                 r_eaddr, i);
14216             if (i < TTE4M) {
14217                 i = TTE4M;
14218             } else {
14219                 continue;
14220             }
14221             break;
14222         }
14223         i++;
14224     }
14225     /* Remove the preallocated 1/4 8k ttecnt for 4M regions. */
14226     if (r_pgsz >= TTE4M) {
14227         rttecnt = r_size >> (TTE_PAGE_SHIFT(TTE8K) + 2);
14228         ASSERT(sfmmup->sfmmu_tsb0_4minflcnt >=
14229             rttecnt);
14230         sfmmup->sfmmu_tsb0_4minflcnt -= rttecnt;
14231     }

14233     /* update shme rgns ttecnt in sfmmu_ttecnt */
14234     rttecnt = r_size >> TTE_PAGE_SHIFT(r_pgsz);
14235     ASSERT(sfmmup->sfmmu_ttecnt[r_pgsz] >= rttecnt);
14236     atomic_add_long(&sfmmup->sfmmu_ttecnt[r_pgsz], -rttecnt);

14238     sfmmu_hat_exit(hatlockp);
14239     if (scdp != NULL && sfmmup->sfmmu_scdp == NULL) {
14240         /* sfmmup left the scd, grow private tsb */
14241         sfmmu_check_page_sizes(sfmmup, 1);
14242     } else {
14243         sfmmu_check_page_sizes(sfmmup, 0);
14244     }
14245 }

14247 if (r_type == SFMMU_REGION_HME) {

```

```

14248         sfmmu_unlink_from_hmeregion(sfmmup, rgnp);
14249     }

14251     r_obj = rgnp->rgn_obj;
14252     if (atomic_dec_32_nv((volatile uint_t *)&rgnp->rgn_refcnt)) {
14253         return;
14254     }

14256     /*
14257     * looks like nobody uses this region anymore. Free it.
14258     */
14259     rhash = RGN_HASH_FUNCTION(r_obj);
14260     mutex_enter(&srdp->srd_mutex);
14261     for (prev_rgnpp = &srdp->srd_rgnhash[rhash];
14262         (cur_rgnp = *prev_rgnpp) != NULL;
14263         prev_rgnpp = &cur_rgnp->rgn_hash) {
14264         if (cur_rgnp == rgnp && cur_rgnp->rgn_refcnt == 0) {
14265             break;
14266         }
14267     }

14269     if (cur_rgnp == NULL) {
14270         mutex_exit(&srdp->srd_mutex);
14271         return;
14272     }

14274     ASSERT((rgnp->rgn_flags & SFMMU_REGION_TYPE_MASK) == r_type);
14275     *prev_rgnpp = rgnp->rgn_hash;
14276     if (r_type == SFMMU_REGION_ISM) {
14277         rgnp->rgn_flags |= SFMMU_REGION_FREE;
14278         ASSERT(rid < srdp->srd_next_ismrid);
14279         rgnp->rgn_next = srdp->srd_ismrgnfree;
14280         srdp->srd_ismrgnfree = rgnp;
14281         ASSERT(srdp->srd_ismbusyrngns > 0);
14282         srdp->srd_ismbusyrngns--;
14283         mutex_exit(&srdp->srd_mutex);
14284         return;
14285     }
14286     mutex_exit(&srdp->srd_mutex);

14288     /*
14289     * Destroy region's hmeblks.
14290     */
14291     sfmmu_unload_hmeregion(srdp, rgnp);

14293     rgnp->rgn_hmeflags = 0;

14295     ASSERT(rgnp->rgn_sfmmu_head == NULL);
14296     ASSERT(rgnp->rgn_id == rid);
14297     for (i = 0; i < MMU_PAGE_SIZES; i++) {
14298         rgnp->rgn_ttecnt[i] = 0;
14299     }
14300     rgnp->rgn_flags |= SFMMU_REGION_FREE;
14301     mutex_enter(&srdp->srd_mutex);
14302     ASSERT(rid < srdp->srd_next_hmerid);
14303     rgnp->rgn_next = srdp->srd_hmergnfree;
14304     srdp->srd_hmergnfree = rgnp;
14305     ASSERT(srdp->srd_hmebusyrngns > 0);
14306     srdp->srd_hmebusyrngns--;
14307     mutex_exit(&srdp->srd_mutex);
14308 }

```

\_\_\_\_\_unchanged\_portion\_omitted\_\_\_\_\_

new/usr/src/uts/sfmmu/vm/hat\_sfmmu.h

1

```
*****
86609 Fri May 8 18:02:57 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.
664 */

664 struct hat {
665     void *sfmmu_xhat_provider; /* NULL for CPU hat */
666     cpuset_t sfmmu_cpusran; /* cpu bit mask for efficient xcalls */
667     struct as *sfmmu_as; /* as this hat provides mapping for */
668     /* per pgsz private ttecnt + shme rgns ttecnt for rgns not in SCD */
669     ulong_t sfmmu_ttecnt[MMU_PAGE_SIZES];
670     /* shme rgns ttecnt for rgns in SCD */
671     ulong_t sfmmu_scdrttecnt[MMU_PAGE_SIZES];
672     /* est. ism ttes that are NOT in a SCD */
673     /* ttecnt for isms that are in a SCD */
674     ulong_t sfmmu_ismttecnt[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 */
```

new/usr/src/uts/sfmmu/vm/hat\_sfmmu.h

2

```
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_hvbblock;
703 #endif
704 /*
705 * sfmmu_ctxs is a variable length array of max_mmu_ctxs # of
706 * elements. max_mmu_ctxs is determined at run-time.
707 * sfmmu_ctxs[1] is just the first 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 * maintains. 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_hmect 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.
1241 *
1242 * Note that xhat_hme_blk's layout follows this structure: hme_blk_misc
1243 * and sf_hment are at the same offsets in both structures. Whenever
1244 * hme_blk is changed, xhat_hme_blk may need to be updated as well.
1245 */

1243 struct hme_blk_misc {
1244     uint_t notused:26;
1245     uint_t notused:25;
1246     uint_t shared_bit:1; /* set for SRD shared hmeblk */
1247     uint_t xhat_bit:1; /* set for an xhat hme_blk */
1248     uint_t shadow_bit:1; /* set for a shadow hme_blk */
1249     uint_t nucleus_bit:1; /* set for a nucleus hme_blk */
1250     uint_t ttesize:3; /* contains ttesz of hmeblk */
1251 };
unchanged_portion_omitted_

1284 #define hblk_shared hblk_misc.shared_bit
```



```

1293 #define hblk_xhat_bit    hblk_misc.xhat_bit
1285 #define hblk_shw_bit     hblk_misc.shadow_bit
1286 #define hblk_nuc_bit     hblk_misc.nucleus_bit
1287 #define hblk_ttesz      hblk_misc.ttesize
1288 #define hblk_hmecnt      hblk_un.hblk_counts.hblk_hmecount
1289 #define hblk_vcncnt      hblk_un.hblk_counts.hblk_validcnt
1290 #define hblk_shw_mask     hblk_un.hblk_shadow_mask

1292 #define MAX_HBLK_LCKCNT  0xFFFFFFFF
1293 #define HMEBLK_ALIGN    0x8          /* hmeblk has to be double aligned */

1295 #ifdef HBLK_TRACE

1297 #define HBLK_STACK_TRACE(hmeblkp, lock) \
1298 { \
1299     int flag = lock;          /* to pacify lint */ \
1300     int audit_index; \
1301 \
1302     mutex_enter(&hmeblkp->hblk_audit_lock); \
1303     audit_index = hmeblkp->hblk_audit_index; \
1304     hmeblkp->hblk_audit_index = ((hmeblkp->hblk_audit_index + 1) & \
1305     (HBLK_AUDIT_CACHE_SIZE - 1)); \
1306     mutex_exit(&hmeblkp->hblk_audit_lock); \
1307 \
1308     if (flag) \
1309         hmeblkp->hblk_audit_cache[audit_index].flag = \
1310         HBLK_LOCK_PATTERN; \
1311     else \
1312         hmeblkp->hblk_audit_cache[audit_index].flag = \
1313         HBLK_UNLOCK_PATTERN; \
1314 \
1315     hmeblkp->hblk_audit_cache[audit_index].thread = curthread; \
1316     hmeblkp->hblk_audit_cache[audit_index].depth = \
1317     getpstack(hmeblkp->hblk_audit_cache[audit_index].stack, \
1318     HBLK_STACK_DEPTH); \
1319 }

1320 #endif

```

unchanged portion omitted

```

1826 extern size_t    tsb_slab_size;
1827 extern uint_t    tsb_slab_shift;
1828 extern size_t    tsb_slab_mask;

1830 #endif /* !_ASM */

1832 /*
1833  * Flags for TL kpm tsbmiss handler
1834  */
1835 #define KPMTSBM_ENABLE_FLAG    0x01    /* bit copy of kpm_enable */
1836 #define KPMTSBM_TLTSBM_FLAG    0x02    /* use TL tsbmiss handler */
1837 #define KPMTSBM_TSBPHYS_FLAG    0x04    /* use ASI_MEM for TSB update */

1839 /*
1840  * The TSB
1841  * All TSB sizes supported by the hardware are now supported (8K - 1M).
1842  * For kernel TSBs we may go beyond the hardware supported sizes and support
1843  * larger TSBs via software.
1844  * All TTE sizes are supported in the TSB; the manner in which this is
1845  * done is cpu dependent.
1846  */
1847 #define TSB_MIN_SZCODE        TSB_8K_SZCODE    /* min. supported TSB size */
1848 #define TSB_MIN_OFFSET_MASK    (TSB_OFFSET_MASK(TSB_MIN_SZCODE))

1850 #ifdef sun4v
1851 #define UTSB_MAX_SZCODE        TSB_256M_SZCODE    /* max. supported TSB size */
1852 #else /* sun4u */
1853 #define UTSB_MAX_SZCODE        TSB_1M_SZCODE    /* max. supported TSB size */

```

```

1854 #endif /* sun4v */

1856 #define UTSB_MAX_OFFSET_MASK    (TSB_OFFSET_MASK(UTSB_MAX_SZCODE))

1858 #define TSB_FREEMEM_MIN        0x1000    /* 32 mb */
1859 #define TSB_FREEMEM_LARGE    0x10000    /* 512 mb */
1860 #define TSB_8K_SZCODE        0    /* 512 entries */
1861 #define TSB_16K_SZCODE        1    /* 1k entries */
1862 #define TSB_32K_SZCODE        2    /* 2k entries */
1863 #define TSB_64K_SZCODE        3    /* 4k entries */
1864 #define TSB_128K_SZCODE        4    /* 8k entries */
1865 #define TSB_256K_SZCODE        5    /* 16k entries */
1866 #define TSB_512K_SZCODE        6    /* 32k entries */
1867 #define TSB_1M_SZCODE        7    /* 64k entries */
1868 #define TSB_2M_SZCODE        8    /* 128k entries */
1869 #define TSB_4M_SZCODE        9    /* 256k entries */
1870 #define TSB_8M_SZCODE        10    /* 512k entries */
1871 #define TSB_16M_SZCODE        11    /* 1M entries */
1872 #define TSB_32M_SZCODE        12    /* 2M entries */
1873 #define TSB_64M_SZCODE        13    /* 4M entries */
1874 #define TSB_128M_SZCODE        14    /* 8M entries */
1875 #define TSB_256M_SZCODE        15    /* 16M entries */
1876 #define TSB_ENTRY_SHIFT        4    /* each entry = 128 bits = 16 bytes */
1877 #define TSB_ENTRY_SIZE        (1 << 4)
1878 #define TSB_START_SIZE        9
1879 #define TSB_ENTRIES(tbsz)    (1 << (TSB_START_SIZE + tbsz))
1880 #define TSB_BYTES(tbsz)    (TSB_ENTRIES(tbsz) << TSB_ENTRY_SHIFT)
1881 #define TSB_OFFSET_MASK(tbsz)    (TSB_ENTRIES(tbsz) - 1)
1882 #define TSB_BASEADDR_MASK    ((1 << 12) - 1)

1884 /*
1885  * sun4u platforms
1886  * -----
1887  * We now support two user TSBs with one TSB base register.
1888  * Hence the TSB base register is split up as follows:
1889  *
1890  * When only one TSB present:
1891  *   [63  62..42  41..13  12..4  3..0]
1892  *   ^      ^      ^      ^      ^
1893  *   |      |      |      |      |
1894  *   |      |      |      |      |
1895  *   |      |      |      |      |
1896  *   |      |      |      |      |
1897  *   |      |      |      |      |
1898  *   |      |      |      |      |
1899  *   |      |      |      |      |
1900  *   |      |      |      |      |
1901  *   |      |      |      |      |
1902  *   |      |      |      |      |
1903  *   |      |      |      |      |
1904  *   |      |      |      |      |
1905  *   |      |      |      |      |
1906  *   |      |      |      |      |
1907  *   |      |      |      |      |
1908  *   |      |      |      |      |
1909  *   |      |      |      |      |
1910  *   |      |      |      |      |
1911  *   |      |      |      |      |
1912  *   |      |      |      |      |
1913  *   |      |      |      |      |
1914  *   |      |      |      |      |
1915  *   |      |      |      |      |
1916  *   |      |      |      |      |
1917  *   |      |      |      |      |
1918  *   |      |      |      |      |
1919  *   |      |      |      |      |

```

```

1920 *      |      |  _ VA hole (Spitfire) / ones (Cheetah and beyond)
1921 *      |      |
1922 *      |      |  _ 1
1923 *
1924 * Note that since we store 21..13 of each TSB's VA, TSBs and their slabs
1925 * may be up to 4M in size. For now, only hardware supported TSB sizes
1926 * are supported, though the slabs are usually 4M in size.
1927 *
1928 * sun4u platforms that define UTSB_PHYS use physical addressing to access
1929 * the user TSBs at TL>0. The first user TSB base is in the MMU I/D TSB Base
1930 * registers. The second TSB base uses a dedicated scratchpad register which
1931 * requires a definition of SCRATCHPAD_UTSBREG2 in mach_sfmmu.h. The layout for
1932 * both registers is equivalent to sun4v below, except the TSB PA range is
1933 * [46..13] for sun4u.
1934 *
1935 * sun4v platforms
1936 * -----
1937 * On sun4v platforms, we use two dedicated scratchpad registers as pseudo
1938 * hardware TSB base registers to hold up to two different user TSBs.
1939 *
1940 * Each register contains TSB's physical base and size code information
1941 * as follows:
1942 *
1943 *      [63..56  55..13  12..4  3..0]
1944 *      ^         ^         ^         ^
1945 *      |         |         |         |
1946 *      |         |         |         |  _ TSB size code
1947 *      |         |         |         |
1948 *      |         |         |         |  _ Reserved 0
1949 *      |         |         |         |
1950 *      |         |         |         |  _ TSB PA[55..13]
1951 *      |         |         |         |
1952 *      |         |         |         |
1953 *      |         |         |         |
1954 *      |         |         |         |  _ 0 for valid TSB
1955 *
1956 * Absence of a user TSB (primarily the second user TSB) is indicated by
1957 * storing a negative value in the TSB base register. This allows us to
1958 * check for presence of a user TSB by simply checking bit# 63.
1959 */
1960 #define TSBREG_MSB_SHIFT      32          /* set upper bits */
1961 #define TSBREG_MSB_CONST      0xfffff800 /* set bits 63..43 */
1962 #define TSBREG_FIRTSB_SHIFT   42          /* to clear bits 63:22 */
1963 #define TSBREG_SECTSB_MKSHIFT 20          /* 21:13 --> 41:33 */
1964 #define TSBREG_SECTSB_LSHIFT  22          /* to clear bits 63:42 */
1965 #define TSBREG_SECTSB_RSHIFT  (TSBREG_SECTSB_MKSHIFT + TSBREG_SECTSB_LSHIFT)
1966                                     /* sectsb va -> bits 21:13 */
1967                                     /* after clearing upper bits */
1968 #define TSBREG_SECSZ_SHIFT    29          /* to get sectsb szc to 3:0 */
1969 #define TSBREG_VAMASK_SHIFT   13          /* set up VA mask */
1970
1971 #define BIGKTSB_SZ_MASK        0xf
1972 #define TSB_SOFTSZ_MASK        BIGKTSB_SZ_MASK
1973 #define MIN_BIGKTSB_SZCODE    9          /* 256k entries */
1974 #define MAX_BIGKTSB_SZCODE    11         /* 1024k entries */
1975 #define MAX_BIGKTSB_TTES      (TSB_BYTES(MAX_BIGKTSB_SZCODE) / MMU_PAGESIZE4M)
1976
1977 #define TAG_VALO_SHIFT         22          /* tag's va are bits 63-22 */
1978 /*
1979 * sw bits used on tsb_tag - bit masks used only in assembly
1980 * use only a sethi for these fields.
1981 */
1982 #define TSBTAG_INVALID        0x00008000 /* tsb_tag.tag_invalid */
1983 #define TSBTAG_LOCKED         0x00004000 /* tsb_tag.tag_locked */
1984
1985 #ifndef _ASM

```

```

1987 /*
1988 * Marker to indicate that this instruction will be hot patched at runtime
1989 * to some other value.
1990 * This value must be zero since it fills in the imm bits of the target
1991 * instructions to be patched
1992 */
1993 #define RUNTIME_PATCH      (0)
1994
1995 /*
1996 * V9 defines nop instruction as the following, which we use
1997 * at runtime to nullify some instructions we don't want to
1998 * execute in the trap handlers on certain platforms.
1999 */
2000 #define MAKE_NOP_INSTR(reg) \
2001     sethi    %hi(0x1000000), reg
2002
2003 /*
2004 * This macro constructs a SPARC V9 "jmpl <source reg>, %g0"
2005 * instruction, with the source register specified by the jump_reg_number.
2006 * The jmp opcode [24:19] = 11 1000 and source register is bits [18:14].
2007 * The instruction is returned in reg. The macro is used to patch in a jmp!
2008 * instruction at runtime.
2009 */
2010 #define MAKE_JMP_INSTR(jump_reg_number, reg, tmp) \
2011     sethi    %hi(0x81c00000), reg; \
2012     mov      jump_reg_number, tmp; \
2013     sll     tmp, 14, tmp; \
2014     or      reg, tmp, reg
2015
2016 /*
2017 * Macro to get hat per-MMU cnum on this CPU.
2018 * sfmmu - In, pass in "sfmmup" from the caller.
2019 * cnum - Out, return 'cnum' to the caller
2020 * scr - scratch
2021 */
2022 #define SFMMU_CPU_CNUM(sfmmu, cnum, scr) \
2023     CPU_ADDR(scr, cnum); /* scr = load CPU struct addr */ \
2024     ld      [scr + CPU_MMU_IDX], cnum; /* cnum = mmuid */ \
2025     add     sfmmu, SFMMU_CTXS, scr; /* scr = sfmmup->sfmmu_ctxs[] */ \
2026     sllx   cnum, SFMMU_MMU_CTX_SHIFT, cnum; \
2027     add     scr, cnum, scr; /* scr = sfmmup->sfmmu_ctxs[id] */ \
2028     ldx    [scr + SFMMU_MMU_GC_NUM], scr; /* sfmmu_ctxs[id].gcnum */ \
2029     sllx   scr, SFMMU_MMU_CNUM_LSHIFT, scr; \
2030     srlx   scr, SFMMU_MMU_CNUM_LSHIFT, cnum; /* cnum = sfmmu cnum */
2031
2032 /*
2033 * Macro to get hat gnum & cnum associated with sfmmu_ctx[mmuid] entry
2034 * entry - In, pass in (&sfmmu_ctxs[mmuid] - SFMMU_CTXS) from the caller.
2035 * gnum - Out, return sfmmu gnum
2036 * cnum - Out, return sfmmu cnum
2037 * reg - scratch
2038 */
2039 #define SFMMU_MMUID_GNUM_CNUM(entry, gnum, cnum, reg) \
2040     ldx    [entry + SFMMU_CTXS], reg; /* reg = sfmmu (gnum | cnum) */ \
2041     srlx   reg, SFMMU_MMU_GNUM_RSHIFT, gnum; /* gnum = sfmmu gnum */ \
2042     sllx   reg, SFMMU_MMU_CNUM_LSHIFT, cnum; \
2043     srlx   cnum, SFMMU_MMU_CNUM_LSHIFT, cnum; /* cnum = sfmmu cnum */
2044
2045 /*
2046 * Macro to get this CPU's tsbmiss area.
2047 */
2048 #define CPU_TSBMISS_AREA(tsbmiss, tmp1) \
2049     CPU_INDEX(tmp1, tsbmiss); /* tmp1 = cpu idx */ \
2050     sethi  %hi(tsbmiss_area), tsbmiss; /* tsbmiss base ptr */ \
2051     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)  ((pp)->p_vcolor = ((color) + 1))
2174
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_PAGE_SHIFT) & 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 corresponding 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 * example, 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_GROUPLER(pp, sz) \
2223     (&(pp)[-1*(int)(pp->p_pagenum & (TTPAGES(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_GROUPLER(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 mapping 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_tsbttag(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_ttopfn(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_kvaszcpfn(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_membt(tte_t *, pfn_t, uint_t, int);
2284 extern void sfmmu_tload(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_tsbbsz;
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 kpmtsbn kpmtsbn_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_ttload8k;           /* calls to sfmmu_ttload */
2472     int     sf_ttload64k;          /* calls to sfmmu_ttload */
2473     int     sf_ttload512k;         /* calls to sfmmu_ttload */
2474     int     sf_ttload4m;           /* calls to sfmmu_ttload */
2475     int     sf_ttload32m;          /* calls to sfmmu_ttload */
2476     int     sf_ttload256m;         /* calls to sfmmu_ttload */

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 ttload */
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

new/usr/src/uts/sun4u/Makefile.files

1

\*\*\*\*\*

6586 Fri May 8 18:02:57 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

\*\*\*\*\*

```
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 #
23 # Copyright 2009 Sun Microsystems, Inc. All rights reserved.
24 # Use is subject to license terms.
25 #
26 # This Makefile defines all file modules for the directory uts/sun4u
27 # and it's children. These are the source files which are sun4u
28 # "implementation architecture" dependent.
29 #
30 #
31 #
32 # object lists
33 #
34 CORE_OBJS += atomic.o
35 CORE_OBJS += bootops.o
36 CORE_OBJS += cmp.o
37 CORE_OBJS += cpc_hwreg.o
38 CORE_OBJS += cpc_subr.o
39 CORE_OBJS += cpupm.o
40 CORE_OBJS += mach_cpu_states.o
41 CORE_OBJS += mach_ddi_impl.o
42 CORE_OBJS += ecc.o
43 CORE_OBJS += fillsysinfo.o
44 CORE_OBJS += forthdebug.o
45 CORE_OBJS += hardclk.o
46 CORE_OBJS += hat_sfmmu.o
47 CORE_OBJS += hat_kdi.o
48 CORE_OBJS += iscsi_boot.o
49 CORE_OBJS += mach_copy.o
50 CORE_OBJS += mach_kpm.o
51 CORE_OBJS += mach_mp_startup.o
52 CORE_OBJS += mach_mp_states.o
53 CORE_OBJS += mach_sfmmu.o
54 CORE_OBJS += mach_startup.o
```

new/usr/src/uts/sun4u/Makefile.files

2

```
55 CORE_OBJS += mach_subr_asm.o
56 CORE_OBJS += mach_trap.o
57 CORE_OBJS += mach_vm_dep.o
58 CORE_OBJS += mach_xc.o
59 CORE_OBJS += mem_cage.o
60 CORE_OBJS += mem_config.o
61 CORE_OBJS += memlist_new.o
62 CORE_OBJS += memscrub.o
63 CORE_OBJS += memscrub_asm.o
64 CORE_OBJS += ppage.o
65 CORE_OBJS += sfmmu_kdi.o
66 CORE_OBJS += swtch.o
67 CORE_OBJS += xhat_sfmmu.o

68 #
69 # Some objects must be linked at the front of the image (or
70 # near other objects at the front of the image).
71 #
72 SPECIAL_OBJS += trap_table.o
73 SPECIAL_OBJS += locore.o
74 SPECIAL_OBJS += mach_locore.o
75 SPECIAL_OBJS += sfmmu_asm.o
76 SPECIAL_OBJS += mach_sfmmu_asm.o
77 SPECIAL_OBJS += interrupt.o
78 SPECIAL_OBJS += mach_interrupt.o
79 SPECIAL_OBJS += wbuf.o

81 #
82 # driver modules
83 #
84 ROOTNEX_OBJS += mach_rootnex.o
85 UPA64S_OBJS += upa64s.o
86 SYSIO_SBUS_OBJS += iommu.o sysioerr.o sysiosbus.o iocache.o
87 PX_OBJS += px_asm_4u.o px_err.o px_hlib.o px_lib4u.o px_tools_4u.o
88 PCI_COMMON_OBJS += pci.o pci_util.o pci_dma.o pci_devctl.o \
89 pci_fdvma.o pci_iommu.o pci_sc.o pci_debug.o \
90 pci_cb.o pci_ib.o pci_ecc.o pci_pbm.o pci_intr.o \
91 pci_space.o pci_counters.o pci_axq.o \
92 pci_fm.o pci_reloc.o pci_tools.o pci_asm.o
93 RMCLOMV_OBJS += rmlcmv.o

95 PSYCHO_PCI_OBJS += $(PCI_COMMON_OBJS) pcipsy.o
96 SCHIZO_PCI_OBJS += $(PCI_COMMON_OBJS) pcisch_asm.o pcisch.o pcix.o
97 SIMBA_PCI_OBJS += simba.o
98 DB21554_OBJS += db21554.o
99 US_OBJS += cpudrv.o cpudrv_mach.o
100 POWER_OBJS += power.o
101 EPIC_OBJS += epic.o
102 GRBEEP_OBJS += grbeep.o
103 ADM1031_OBJS += adm1031.o
104 ICS951601_OBJS += ics951601.o
105 PPM_OBJS += ppm_subr.o ppm.o ppm_plat.o
106 OPLCFG_OBJS += opl_cfg.o
107 PCF8584_OBJS += pcf8584.o
108 PCA9556_OBJS += pca9556.o
109 ADM1026_OBJS += adm1026.o
110 BBC_OBJS += bbc_beep.o
111 TDA8444_OBJS += tda8444.o
112 MAX1617_OBJS += max1617.o
113 SEEPROM_OBJS += seeprom.o
114 I2C_SVC_OBJS += i2c_svc.o
115 SMBUS_OBJS += smbus.o
116 SCHPPM_OBJS += schppm.o
117 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 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 SNI_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

```



new/usr/src/uts/sun4u/vm/mach\_kpm.c

1

\*\*\*\*\*

60364 Fri May 8 18:02:57 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
```

new/usr/src/uts/sun4u/vm/mach\_kpm.c

2

```
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;
1689         tmphat = hblktosfmmu(hmeblkp);
1690         sfmmu_copytte(&sfhmep->hme_tte, &tte);
1691         ASSERT(TTE_IS_VALID(&tte));
1692         if ((tmphat == ksfmmap) || hmeblkp->hblk_lckcnt) {
1693             /*
1694             * We have an uncache conflict
1695             */
1696             SFMMU_STAT(sf_uncache_conflict);
1697             sfmmu_page_cache_array(pp, HAT_TMPNC, CACHE_FLUSH, 1);
1698             return;
1699         }
1700     }

1702     /*
1703     * We have an unload conflict
1704     */
1705     SFMMU_STAT(sf_unload_conflict);

1707     for (sfhmep = pp->p_mapping; sfhmep; sfhmep = tmphme) {
1708         tmphme = sfhmep->hme_next;
1709         if (IS_PAHME(sfhmep))
1710             continue;
1711         hmeblkp = sfmmu_hmetohblk(sfhmep);
1712         if (hmeblkp->hblk_xhat_bit)
1713             continue;
1714         (void) sfmmu_pageunload(pp, sfhmep, TTE8K);
1715     }

1717     /*
1718     * Unloads only does tlb flushes so we need to flush the
1719     * dcache vcolor here.
1720     */
1721     sfmmu_cache_flush(pp->p_pagenum, PP_GET_VCOLOR(pp));
1722     PP_SET_VCOLOR(pp, vcolor);
1723 }

_____unchanged_portion_omitted_____
```

new/usr/src/uts/sun4v/Makefile.files

1

\*\*\*\*\*

6371 Fri May 8 18:02:58 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

\*\*\*\*\*

```
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 #
23 # Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
24 #
25 # This Makefile defines all file modules for the directory uts/sun4v
26 # and it's children. These are the source files which are sun4v
27 # "implementation architecture" dependent.
28 #
29 #
30 #
31 # object lists
32 #
33 CORE_OBJS += bootops.o
34 CORE_OBJS += cmp.o
35 CORE_OBJS += cpc_hwreg.o
36 CORE_OBJS += cpc_subr.o
37 CORE_OBJS += error.o
38 CORE_OBJS += fillsysinfo.o
39 CORE_OBJS += forthdebug.o
40 CORE_OBJS += hardclk.o
41 CORE_OBJS += hat_sfmmu.o
42 CORE_OBJS += hat_kdi.o
43 CORE_OBJS += hsvc.o
44 CORE_OBJS += iscsi_boot.o
45 CORE_OBJS += kldc.o
46 CORE_OBJS += lpad.o
47 CORE_OBJS += mach_cpu_states.o
48 CORE_OBJS += mach_ddi_impl.o
49 CORE_OBJS += mach_descrip.o
50 CORE_OBJS += mach_kpm.o
51 CORE_OBJS += mach_mp_startup.o
52 CORE_OBJS += mach_mp_states.o
53 CORE_OBJS += mach_proc_init.o
54 CORE_OBJS += mach_sfmmu.o
```

new/usr/src/uts/sun4v/Makefile.files

2

```
55 CORE_OBJS += mach_startup.o
56 CORE_OBJS += mach_subr_asm.o
57 CORE_OBJS += mach_trap.o
58 CORE_OBJS += mach_vm_dep.o
59 CORE_OBJS += mach_xc.o
60 CORE_OBJS += mem_cage.o
61 CORE_OBJS += mem_config.o
62 CORE_OBJS += memlist_new.o
63 CORE_OBJS += memseg.o
64 CORE_OBJS += mpo.o
65 CORE_OBJS += ppage.o
66 CORE_OBJS += promif_asr.o
67 CORE_OBJS += promif_cpu.o
68 CORE_OBJS += promif_emul.o
69 CORE_OBJS += promif_mon.o
70 CORE_OBJS += promif_io.o
71 CORE_OBJS += promif_interp.o
72 CORE_OBJS += promif_key.o
73 CORE_OBJS += promif_power_off.o
74 CORE_OBJS += promif_prop.o
75 CORE_OBJS += promif_node.o
76 CORE_OBJS += promif_reboot.o
77 CORE_OBJS += promif_stree.o
78 CORE_OBJS += promif_test.o
79 CORE_OBJS += promif_version.o
80 CORE_OBJS += sfmmu_kdi.o
81 CORE_OBJS += suspend.o
82 CORE_OBJS += swtch.o
83 CORE_OBJS += wdt.o
84 CORE_OBJS += xhat_sfmmu.o
85 CORE_OBJS += mdesc_diff.o
86 CORE_OBJS += mdesc_findname.o
87 CORE_OBJS += mdesc_findnodeprop.o
88 CORE_OBJS += mdesc_fini.o
89 CORE_OBJS += mdesc_getbinsize.o
90 CORE_OBJS += mdesc_getgen.o
91 CORE_OBJS += mdesc_getpropdata.o
92 CORE_OBJS += mdesc_getpropstr.o
93 CORE_OBJS += mdesc_getpropval.o
94 CORE_OBJS += mdesc_init_intern.o
95 CORE_OBJS += mdesc_nodecount.o
96 CORE_OBJS += mdesc_rootnode.o
97 CORE_OBJS += mdesc_scandag.o
98 #
99 #
100 # Some objects must be linked at the front of the image (or
101 # near other objects at the front of the image).
102 #
103 SPECIAL_OBJS += trap_table.o
104 SPECIAL_OBJS += locore.o
105 SPECIAL_OBJS += mach_locore.o
106 SPECIAL_OBJS += sfmmu_asm.o
107 SPECIAL_OBJS += mach_sfmmu_asm.o
108 SPECIAL_OBJS += interrupt.o
109 SPECIAL_OBJS += mach_interrupt.o
110 SPECIAL_OBJS += wbuf.o
111 SPECIAL_OBJS += hcall.o
112 SPECIAL_OBJS += intrq.o
113 #
114 #
115 # driver modules
116 #
117 ROOTNEX_OBJS += mach_rootnex.o
118 PX_OBJS += px_lib4v.o px_err.o px_tools_4v.o px_hcall.o px_libhv.o
119 FPC_OBJS += fpc-impl-4v.o fpc-asm-4v.o
```

## new/usr/src/uts/sun4v/Makefile.files

3

```

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 += niумx.o niумx_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 vdisk_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

```

## new/usr/src/uts/sun4v/Makefile.files

4

```

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

```