

```

*****
21346 Thu Feb 18 08:40:37 2016
new/usr/src/cmd/mdb/common/modules/genunix/findstack.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

583 /*ARGSUSED*/
584 int
585 stacks(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
586 {
587     size_t idx;

589     char *seen = NULL;

591     const char *caller_str = NULL;
592     const char *excl_caller_str = NULL;
593     uintptr_t caller = 0, excl_caller = 0;
594     const char *module_str = NULL;
595     const char *excl_module_str = NULL;
596     stacks_module_t module, excl_module;
597     const char *sobj = NULL;
598     const char *excl_sobj = NULL;
599     uintptr_t sobj_ops = 0, excl_sobj_ops = 0;
600     const char *tstate_str = NULL;
601     const char *excl_tstate_str = NULL;
602     uint_t tstate = -1U;
603     uint_t excl_tstate = -1U;
604     uint_t printed = 0;

606     uint_t all = 0;
607     uint_t force = 0;
608     uint_t interesting = 0;
609     uint_t verbose = 0;

611     /*
612     * We have a slight behavior difference between having piped
613     * input and 'addr::stacks'. Without a pipe, we assume the
614     * thread pointer given is a representative thread, and so
615     * we include all similar threads in the system in our output.
616     *
617     * With a pipe, we filter down to just the threads in our
618     * input.
619     */
620     uint_t addrspec = (flags & DCMD_ADDRSPEC);
621     uint_t only_matching = addrspec && (flags & DCMD_PIPE);

623     mdb_pipe_t p;

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

628     if (mdb_getopts(argc, argv,
629         'a', MDB_OPT_SETBITS, TRUE, &all,
630         'f', MDB_OPT_SETBITS, TRUE, &force,
631         'i', MDB_OPT_SETBITS, TRUE, &interesting,
632         'v', MDB_OPT_SETBITS, TRUE, &verbose,
633         'c', MDB_OPT_STR, &caller_str,
634         'C', MDB_OPT_STR, &excl_caller_str,
635         'm', MDB_OPT_STR, &module_str,
636         'M', MDB_OPT_STR, &excl_module_str,
637         's', MDB_OPT_STR, &sobj,
638         'S', MDB_OPT_STR, &excl_sobj,
639         't', MDB_OPT_STR, &tstate_str,
640         'T', MDB_OPT_STR, &excl_tstate_str,
641         NULL) != argc)

```

```

642         return (DCMD_USAGE);

644     if (interesting) {
645         if (sobj != NULL || excl_sobj != NULL ||
646             tstate_str != NULL || excl_tstate_str != NULL) {
647             mdb_warn(
648                 "stacks: -i is incompatible with -[sStT]\n");
649             return (DCMD_USAGE);
650         }
651         excl_sobj = "CV";
652         excl_tstate_str = "FREE";
653     }

655     if (caller_str != NULL) {
656         mdb_set_dot(0);
657         if (mdb_eval(caller_str) != 0) {
658             mdb_warn("stacks: evaluation of \"%s\" failed",
659                 caller_str);
660             return (DCMD_ABORT);
661         }
662         caller = mdb_get_dot();
663     }

665     if (excl_caller_str != NULL) {
666         mdb_set_dot(0);
667         if (mdb_eval(excl_caller_str) != 0) {
668             mdb_warn("stacks: evaluation of \"%s\" failed",
669                 excl_caller_str);
670             return (DCMD_ABORT);
671         }
672         excl_caller = mdb_get_dot();
673     }
674     mdb_set_dot(addr);

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

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

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

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

689     if (sobj_ops != 0 && excl_sobj_ops != 0) {
690         mdb_warn("stacks: only one of -s and -S can be specified\n");
691         return (DCMD_USAGE);
692     }

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

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

701     if (tstate != -1U && excl_tstate != -1U) {
702         mdb_warn("stacks: only one of -t and -T can be specified\n");
703         return (DCMD_USAGE);
704     }

706     /*
707     * If there's an address specified, we're going to further filter

```

```

708     * to only entries which have an address in the input. To reduce
709     * overhead (and make the sorted output come out right), we
710     * use mdb_get_pipe() to grab the entire pipeline of input, then
711     * use qsort() and bsearch() to speed up the search.
712     */
713     if (addrspec) {
714         mdb_get_pipe(&p);
715         if (p.pipe_data == NULL || p.pipe_len == 0) {
716             p.pipe_data = &addr;
717             p.pipe_len = 1;
718         }
719         qsort(p.pipe_data, p.pipe_len, sizeof (uintptr_t),
720             uintptrcomp);
721
722         /* remove any duplicates in the data */
723         idx = 0;
724         while (idx < p.pipe_len - 1) {
725             uintptr_t *data = &p.pipe_data[idx];
726             size_t len = p.pipe_len - idx;
727
728             if (data[0] == data[1]) {
729                 memmove(data, data + 1,
730                     (len - 1) * sizeof (*data));
731                 p.pipe_len--;
732                 continue; /* repeat without incrementing idx */
733             }
734             idx++;
735         }
736
737         seen = mdb_zalloc(p.pipe_len, UM_SLEEP | UM_GC);
738     }
739
740     /*
741     * Force a cleanup if we're connected to a live system. Never
742     * do a cleanup after the first invocation around the loop.
743     */
744     force |= (mdb_get_state() == MDB_STATE_RUNNING);
745     if (force && (flags & (DCMD_LOOPFIRST|DCMD_LOOP)) == DCMD_LOOP)
746         force = 0;
747
748     stacks_cleanup(force);
749
750     if (stacks_state == STACKS_STATE_CLEAN) {
751         int res = stacks_run(verbose, addrspec ? &p : NULL);
752         if (res != DCMD_OK)
753             return (res);
754     }
755
756     for (idx = 0; idx < stacks_array_size; idx++) {
757         stacks_entry_t *sep = stacks_array[idx];
758         stacks_entry_t *cur = sep;
759         int frame;
760         size_t count = sep->se_count;
761
762         if (addrspec) {
763             stacks_entry_t *head = NULL, *tail = NULL, *sp;
764             size_t foundcount = 0;
765             /*
766             * We use the now-unused hash chain field se_next to
767             * link together the dups which match our list.
768             */
769             for (sp = sep; sp != NULL; sp = sp->se_dup) {
770                 uintptr_t *entry = bsearch(&sp->se_thread,
771                     p.pipe_data, p.pipe_len, sizeof (uintptr_t),
772                     uintptrcomp);
773                 if (entry != NULL) {

```

```

774             foundcount++;
775             seen[entry - p.pipe_data]++;
776             if (head == NULL)
777                 head = sp;
778             else
779                 tail->se_next = sp;
780             tail = sp;
781             sp->se_next = NULL;
782         }
783     }
784     if (head == NULL)
785         continue; /* no match, skip entry */
786
787     if (only_matching) {
788         cur = sep = head;
789         count = foundcount;
790     }
791
792     if (caller != 0 && !stacks_has_caller(sep, caller))
793         continue;
794
795     if (excl_caller != 0 && stacks_has_caller(sep, excl_caller))
796         continue;
797
798     if (module.sm_size != 0 && !stacks_has_module(sep, &module))
799         continue;
800
801     if (excl_module.sm_size != 0 &&
802         stacks_has_module(sep, &excl_module))
803         continue;
804
805     if (tstate != -1U) {
806         if (tstate == TSTATE_PANIC) {
807             if (!sep->se_panic)
808                 continue;
809         } else if (sep->se_panic || sep->se_tstate != tstate)
810             continue;
811     }
812
813     if (excl_tstate != -1U) {
814         if (excl_tstate == TSTATE_PANIC) {
815             if (sep->se_panic)
816                 continue;
817         } else if (!sep->se_panic &&
818             sep->se_tstate == excl_tstate)
819             continue;
820     }
821
822     if (sobj_ops == SOBJ_ALL) {
823         if (sep->se_sobj_ops == 0)
824             continue;
825     } else if (sobj_ops != 0) {
826         if (sobj_ops != sep->se_sobj_ops)
827             continue;
828     }
829
830     if (!(interesting && sep->se_panic)) {
831         if (excl_sobj_ops == SOBJ_ALL) {
832             if (sep->se_sobj_ops != 0)
833                 continue;
834         } else if (excl_sobj_ops != 0) {
835             if (excl_sobj_ops == sep->se_sobj_ops)
836                 continue;
837         }
838     }

```

```

840     if (flags & DCMD_PIPE_OUT) {
841         while (sep != NULL) {
842             mdb_printf("%lx\n", sep->se_thread);
843             sep = only_matching ?
844                 sep->se_next : sep->se_dup;
845         }
846         continue;
847     }
848
849     if (all || !printed) {
850         mdb_printf("%<u>%-?s %-8s %-?s %8s%</u>\n",
851             "THREAD", "STATE", "SOBJ", "COUNT");
852         printed = 1;
853     }
854
855     do {
856         char state[20];
857         char sobj[100];
858
859         tstate_to_text(cur->se_tstate, cur->se_panic,
860             state, sizeof (state));
861         sobj_to_text(cur->se_sobj_ops,
862             sobj, sizeof (sobj));
863
864         if (cur == sep)
865             mdb_printf("%-?p %-8s %-?s %8d\n",
866                 cur->se_thread, state, sobj, count);
867         else
868             mdb_printf("%-?p %-8s %-?s %8s\n",
869                 cur->se_thread, state, sobj, "-");
870
871         cur = only_matching ? cur->se_next : cur->se_dup;
872     } while (all && cur != NULL);
873
874     if (sep->se_failed != 0) {
875         char *reason;
876         switch (sep->se_failed) {
877             case FSI_FAIL_NOTINMEMORY:
878                 reason = "thread not in memory";
879                 break;
880             case FSI_FAIL_THREADCORRUPT:
881                 reason = "thread structure stack info corrupt";
882                 break;
883             case FSI_FAIL_STACKNOTFOUND:
884                 reason = "no consistent stack found";
885                 break;
886             default:
887                 reason = "unknown failure";
888                 break;
889         }
890         mdb_printf("%?s <s>\n", "", reason);
891     }
892
893     for (frame = 0; frame < sep->se_depth; frame++)
894         mdb_printf("%?s %a\n", "", sep->se_stack[frame]);
895     if (sep->se_overflow)
896         mdb_printf("%?s ... truncated ...\n", "");
897     mdb_printf("\n");
898
899     if (flags & DCMD_ADDRSPEC) {
900         for (idx = 0; idx < p.pipe_len; idx++)
901             if (seen[idx] == 0)
902                 mdb_warn("stacks: %p not in thread list\n",
903                     p.pipe_data[idx]);
904     }

```

```

903         return (DCMD_OK);
904     }

```

---

unchanged\_portion\_omitted

new/usr/src/cmd/mdb/common/modules/genunix/findstack.h

1

\*\*\*\*\*

2801 Thu Feb 18 08:40:37 2016

new/usr/src/cmd/mdb/common/modules/genunix/findstack.h

6583 remove whole-process swapping

\*\*\*\*\*

unchanged portion omitted

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

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

unchanged portion omitted

```

*****
11424 Thu Feb 18 08:40:37 2016
new/usr/src/cmd/mdb/common/modules/genunix/findstack_subr.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

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

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

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

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

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

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

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

179 #ifdef __amd64
180     /*
181      * The stack on amd64 is intentionally misaligned, so ignore the top
182      * half-frame. See thread_stk_init(). When handling traps, the frame
183      * is automatically aligned by the hardware, so we only alter ktop if
184      * needed.
185      */
186     if ((ktop & (STACK_ALIGN - 1)) != 0)
187         ktop -= STACK_ENTRY_ALIGN;
188 #endif

190     /*
191      * If the stack size is larger than a meg, assume that it's bogus.

```

```

192     /*
193     if (stksz > TOO_BIG_FOR_A_STACK) {
194         if (print_warnings)
195             mdb_warn("stack size for thread %p is too big to be "
196                 "reasonable\n", addr);
197         fsip->fsi_failed = FSI_FAIL_THREADCORRUPT;
198         return (DCMD_ERR);
199     }

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

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

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

249     /*
250     * We didn't conclusively find the stack. So we'll take another lap,
251     * and print out anything that looks possible.
252     */
253     if (print_warnings)
254         mdb_printf("Possible stack pointers for thread %p:\n", addr);
255     (void) mdb_vread((caddr_t)ubase, stksz, kbase);

257     for (win = ubase;

```

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

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

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

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

\*\*\*\*\*

109418 Thu Feb 18 08:40:37 2016

new/usr/src/cmd/mdb/common/modules/genunix/kmem.c

6583 remove whole-process swapping

\*\*\*\*\*

unchanged\_portion\_omitted\_

```
4316 static int
4317 whatthread_walk_thread(uintptr_t addr, const kthread_t *t, whatthread_t *w)
4318 {
4319     uintptr_t current, data;
4320
4321     if (t->t_stkbase == NULL)
4322         return (WALK_NEXT);
4323
4324     /*
4325      * Warn about swapped out threads, but drive on anyway
4326      */
4327     if (!(t->t_schedflag & TS_LOAD)) {
4328         mdb_warn("thread %p's stack swapped out\n", addr);
4329         return (WALK_NEXT);
4330     }
4331
4332     /*
4333      * Search the thread's stack for the given pointer. Note that it would
4334      * be more efficient to follow ::kgrep's lead and read in page-sized
4335      * chunks, but this routine is already fast and simple.
4336      */
4337     for (current = (uintptr_t)t->t_stkbase; current < (uintptr_t)t->t_stk;
4338          current += sizeof (uintptr_t)) {
4339         if (mdb_vread(&data, sizeof (data), current) == -1) {
4340             mdb_warn("couldn't read thread %p's stack at %p",
4341                    addr, current);
4342             return (WALK_ERR);
4343         }
4344         if (data == w->wt_target) {
4345             if (w->wt_verbose) {
4346                 mdb_printf("%p in thread %p's stack%s\n",
4347                            current, addr, stack_active(t, current));
4348             } else {
4349                 mdb_printf("%#lx\n", addr);
4350                 return (WALK_NEXT);
4351             }
4352         }
4353     }
4354
4355     return (WALK_NEXT);
4356 }
4357
4358 }
4359
4360 unchanged_portion_omitted_
```

```

*****
21512 Thu Feb 18 08:40:37 2016
new/usr/src/cmd/mdb/common/modules/genunix/leaky_subr.c
6583 remove whole-process swapping
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License, Version 1.0 only
6  * (the "License"). You may not use this file except in compliance
7  * with the License.
8  *
9  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
10 * or http://www.opensolaris.org/os/licensing.
11 * See the License for the specific language governing permissions
12 * and limitations under the License.
13 *
14 * When distributing Covered Code, include this CDDL HEADER in each
15 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
16 * If applicable, add the following below this CDDL HEADER, with the
17 * fields enclosed by brackets "[]" replaced with your own identifying
18 * information: Portions Copyright [yyyy] [name of copyright owner]
19 *
20 * CDDL HEADER END
21 */
22 /*
23 * Copyright 2006 Sun Microsystems, Inc. All rights reserved.
24 * Use is subject to license terms.
25 */

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

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

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

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

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

53 #define TYPE_VMEM      0          /* lkb_data is the vmem_seg's size */
54 #define TYPE_CACHE    1          /* lkb_cid is the bufctl's cache */
55 #define TYPE_KMEM     2          /* lkb_cid is the bufctl's cache */

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

```

```

60 #define LKM_CTL_MASK   3L

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

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

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

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

77     return (WALK_NEXT);
78 }

    unchanged_portion_omitted

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

286     /*
287      * If this thread isn't in memory, we can't look at its stack. This
288      * may result in false positives, so we print a warning.
289      */
290     if (!(t->t_schedflag & TS_LOAD)) {
291         mdb_printf("findleaks: thread %p's stack swapped out; "
292                 "false positives possible\n", addr);
293         return (WALK_NEXT);
294     }

295     if (t->t_state != TS_FREE)
296         leaky_grep(base, stk - base);

297     /*
298      * There is always gunk hanging out between t_stk and the page
299      * boundary. If this thread structure wasn't kmem allocated,
300      * this will include the thread structure itself. If the thread
301      * _is_ kmem allocated, we'll be able to get to it via allthreads.
302      */
303     size = *pagesize - (stk & (*pagesize - 1));

304     leaky_grep(stk, size);

305     return (WALK_NEXT);
306 }

    unchanged_portion_omitted

```



new/usr/src/cmd/rpcsvc/rstat\_proc.c

1

```
*****
26490 Thu Feb 18 08:40:37 2016
new/usr/src/cmd/rpcsvc/rstat_proc.c
patch relling-feedback
*****
```

```
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 2008 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */
```

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

```
26 /*
27  * rstat service: built with rstat.x
28 */
```

```
30 #include <stdio.h>
31 #include <stdlib.h>
32 #include <stdarg.h>
33 #include <string.h>
34 #include <signal.h>
35 #include <utmpx.h>
36 #include <nlist.h>
37 #include <fcntl.h>
38 #include <syslog.h>
39 #include <kstat.h>
```

```
41 #include <rpc/rpc.h>
```

```
43 #include <sys/socket.h>
44 #include <sys/cpuvar.h>
45 #include <sys/sysinfo.h>
46 #include <sys/system.h>
47 #include <errno.h>
48 #include <sys/stropts.h>
49 #include <sys/tihdr.h>
50 #include <sys/sysmacros.h>
```

```
52 #include <net/if.h>
```

```
53 #include <inet/mib2.h>
```

```
55 #include "rstat.h"
56 #include "rstat_v2.h"
```

```
58 typedef struct {
59     kstat_t sys;
```

new/usr/src/cmd/rpcsvc/rstat\_proc.c

2

```
60     kstat_t vm;
61 } _cpu_stats_t;
_____unchanged_portion_omitted_____
```

```
294 void
295 updatestat(int ignored)
296 {
297     extern int _rpcpmstart;          /* Started by a port monitor ? */
298     extern int _rpcsvdirty;         /* Still serving ? */

300 #ifdef DEBUG
301     fprintf(stderr, "entering updatestat\n");
302 #endif
303 #ifdef EXIT_WHEN_IDLE
304     if (_rpcpmstart && sinclastreq >= CLOSEDOWN && !_rpcsvdirty) {
305 #ifdef DEBUG
306         fprintf(stderr, "about to closedown\n");
307 #endif
308         exit(0);
309     }
310     sinclastreq++;
311 #endif /* def EXIT_WHEN_IDLE */

313     (void) alarm(0);
314 #ifdef DEBUG
315     fprintf(stderr, "boottime: %d %d\n", stats_s3.boottime.tv_sec,
316         stats_s3.boottime.tv_usec);
317 #endif
318     while (system_stat_load() || diskinfo_load() || netinfo_load()) {
319         (void) kstat_chain_update(kc);
320         system_stat_init();
321         init_disks();
322         init_net();
323     }
324     stats_s4.cp_time.cp_time_len = CPU_STATES;
325     if (stats_s4.cp_time.cp_time_val == NULL)
326         stats_s4.cp_time.cp_time_val =
327             malloc(stats_s4.cp_time.cp_time_len * sizeof(int));
328     stats_s2.cp_time[RSTAT_CPU_USER] =
329         stats_s3.cp_time[RSTAT_CPU_USER] =
330         stats_s4.cp_time.cp_time_val[RSTAT_CPU_USER] =
331         CPU_STAT(&cpu_stats_all.sys, "cpu_ticks_user");
332     stats_s2.cp_time[RSTAT_CPU_NICE] =
333         stats_s3.cp_time[RSTAT_CPU_NICE] =
334         stats_s4.cp_time.cp_time_val[RSTAT_CPU_NICE] =
335         CPU_STAT(&cpu_stats_all.sys, "cpu_ticks_wait");
336     stats_s2.cp_time[RSTAT_CPU_SYS] =
337         stats_s3.cp_time[RSTAT_CPU_SYS] =
338         stats_s4.cp_time.cp_time_val[RSTAT_CPU_SYS] =
339         CPU_STAT(&cpu_stats_all.sys, "cpu_ticks_kernel");
340     stats_s2.cp_time[RSTAT_CPU_IDLE] =
341         stats_s3.cp_time[RSTAT_CPU_IDLE] =
342         stats_s4.cp_time.cp_time_val[RSTAT_CPU_IDLE] =
343         CPU_STAT(&cpu_stats_all.sys, "cpu_ticks_idle");

345 #ifdef DEBUG
346     fprintf(stderr, "cpu: %d %d %d %d\n",
347         CPU_STAT(&cpu_stats_all.sys, "cpu_ticks_user"),
348         CPU_STAT(&cpu_stats_all.sys, "cpu_ticks_wait"),
349         CPU_STAT(&cpu_stats_all.sys, "cpu_ticks_kernel"),
350         CPU_STAT(&cpu_stats_all.sys, "cpu_ticks_idle"));
351     fprintf(stderr, "cp_time: %d %d %d %d\n",
352         stats_s3.cp_time[RSTAT_CPU_USER],
353         stats_s3.cp_time[RSTAT_CPU_NICE],
354         stats_s3.cp_time[RSTAT_CPU_SYS],
355         stats_s3.cp_time[RSTAT_CPU_IDLE]);
```

```

356 #endif

358 /* current time */
359 gettimeofday((struct timeval *)&stats_s3.curtime, NULL);
360 stats_s4.curtime = stats_s3.curtime;

362 stats_s2.v_pgpgin =
363 stats_s3.v_pgpgin =
364 stats_s4.v_pgpgin = CPU_STAT(&cpu_stats_all.vm, "pgpgin");
365 stats_s2.v_pgpgout =
366 stats_s3.v_pgpgout =
367 stats_s4.v_pgpgout = CPU_STAT(&cpu_stats_all.vm, "pgpgout");
368 stats_s2.v_pswpin =
369 stats_s3.v_pswpin =
370 stats_s4.v_pswpin = 0;
371 stats_s4.v_pswpin = CPU_STAT(&cpu_stats_all.vm, "pgswpin");
372 stats_s2.v_pswpout =
373 stats_s3.v_pswpout =
374 stats_s4.v_pswpout = 0;
375 stats_s4.v_pswpout = CPU_STAT(&cpu_stats_all.vm, "pgswpout");
376 stats_s3.v_intr = CPU_STAT(&cpu_stats_all.sys, "intr");
377 stats_s3.v_intr -= hz*(stats_s3.curtime.tv_sec - btm.tv_sec) +
378 hz*(stats_s3.curtime.tv_usec - btm.tv_usec)/1000000;
379 stats_s2.v_intr =
380 stats_s4.v_intr = stats_s3.v_intr;
381 /* swtch not in V1 */
382 stats_s2.v_swtch =
383 stats_s3.v_swtch =
384 stats_s4.v_swtch = CPU_STAT(&cpu_stats_all.sys, "pswtch");

384 #ifdef DEBUG
385 fprintf(stderr,
386 "pgin: %d pgout: %d swpin: %d swpout: %d intr: %d swtch: %d\n",
387 stats_s3.v_pgpgin,
388 stats_s3.v_pgpgout,
389 stats_s3.v_pswpin,
390 stats_s3.v_pswpout,
391 stats_s3.v_intr,
392 stats_s3.v_swtch);
393 #endif
394 /*
395 * V2 and V3 of rstat are limited to RSTAT_DK_NDRIVE drives
396 */
397 memcpy(stats_s3.dk_xfer, stats_s4.dk_xfer.dk_xfer_val,
398 RSTAT_DK_NDRIVE * sizeof(int));
399 memcpy(stats_s2.dk_xfer, stats_s4.dk_xfer.dk_xfer_val,
400 RSTAT_DK_NDRIVE * sizeof(int));
401 #ifdef DEBUG
402 fprintf(stderr, "dk_xfer: %d %d %d %d\n",
403 stats_s4.dk_xfer.dk_xfer_val[0],
404 stats_s4.dk_xfer.dk_xfer_val[1],
405 stats_s4.dk_xfer.dk_xfer_val[2],
406 stats_s4.dk_xfer.dk_xfer_val[3]);
407 #endif

409 stats_s2.if_ipackets =
410 stats_s3.if_ipackets = stats_s4.if_ipackets;
411 /* no s2 opackets */
412 stats_s3.if_opackets = stats_s4.if_opackets;
413 stats_s2.if_ierrors =
414 stats_s3.if_ierrors = stats_s4.if_ierrors;
415 stats_s2.if_oerrors =
416 stats_s3.if_oerrors = stats_s4.if_oerrors;
417 stats_s2.if_collisions =
418 stats_s3.if_collisions = stats_s4.if_collisions;

```

```

420 stats_s2.avenrun[0] =
421 stats_s3.avenrun[0] =
422 stats_s4.avenrun[0] = avenrun_lmin_knp->value.ul;
423 stats_s2.avenrun[1] =
424 stats_s3.avenrun[1] =
425 stats_s4.avenrun[1] = avenrun_5min_knp->value.ul;
426 stats_s2.avenrun[2] =
427 stats_s3.avenrun[2] =
428 stats_s4.avenrun[2] = avenrun_15min_knp->value.ul;
429 #ifdef DEBUG
430 fprintf(stderr, "avenrun: %d %d %d\n", stats_s3.avenrun[0],
431 stats_s3.avenrun[1], stats_s3.avenrun[2]);
432 #endif
433 signal(SIGALRM, updatestat);
434 alarm(1);
435 }

```

unchanged portion omitted

```

*****
13563 Thu Feb 18 08:40:38 2016
new/usr/src/cmd/stat/vmstat/vmstat.c
patch relling-feedback
*****
_____unchanged_portion_omitted_____

241 static void
242 dovmstats(struct snapshot *old, struct snapshot *new)
243 {
244     kstat_t *oldsys = NULL;
245     kstat_t *newsys = &new->s_sys.ss_agg_sys;
246     kstat_t *oldvm = NULL;
247     kstat_t *newvm = &new->s_sys.ss_agg_vm;
248     double percent_factor;
249     ulong_t sys_updates, vm_updates;
250     int count;

252     adj = 0;

254     if (old) {
255         oldsys = &old->s_sys.ss_agg_sys;
256         oldvm = &old->s_sys.ss_agg_vm;
257     }

259     etime = cpu_ticks_delta(oldsys, newsys);

261     percent_factor = 100.0 / denom(etime);
262     /*
263      * If any time has passed, convert etime to seconds per CPU
264      */
265     etime = etime >= 1.0 ? (etime / nr_active_cpus(new)) / hz : 1.0;
266     sys_updates = denom(DELTA(s_sys.ss_sysinfo.updates));
267     vm_updates = denom(DELTA(s_sys.ss_vminfo.updates));

269     if (timestamp_fmt != NODATE) {
270         print_timestamp(timestamp_fmt);
271         lines--;
272     }

274     if (--lines <= 0)
275         printhdr(0);

277     adj = 0;

279     if (pflag) {
280         adjprintf("%*u", 6,
281             pgtok((int)(DELTA(s_sys.ss_vminfo.swap_avail)
282                 / vm_updates)));
283         adjprintf("%*u", 5,
284             pgtok((int)(DELTA(s_sys.ss_vminfo.freemem) / vm_updates)));
285         adjprintf("%*.0f", 3, kstat_delta(oldvm, newvm, "pgrec")
286             / etime);
287         adjprintf("%*.0f", 3, (kstat_delta(oldvm, newvm, "hat_fault") +
288             kstat_delta(oldvm, newvm, "as_fault")) / etime);
289         adjprintf("%*.0f", 3, pgtok(kstat_delta(oldvm, newvm, "dfree"))
290             / etime);
291         adjprintf("%*ld", 3, pgtok(new->s_sys.ss_deficit));
292         adjprintf("%*.0f", 3, kstat_delta(oldvm, newvm, "scan")
293             / etime);
294         adjprintf("%*.0f", 4,
295             pgtok(kstat_delta(oldvm, newvm, "execpgin")) / etime);
296         adjprintf("%*.0f", 4,
297             pgtok(kstat_delta(oldvm, newvm, "execpgout")) / etime);
298         adjprintf("%*.0f", 4,
299             pgtok(kstat_delta(oldvm, newvm, "execfree")) / etime);

```

```

300         adjprintf("%*.0f", 4,
301             pgtok(kstat_delta(oldvm, newvm, "anonpgin")) / etime);
302         adjprintf("%*.0f", 4,
303             pgtok(kstat_delta(oldvm, newvm, "anonpgout")) / etime);
304         adjprintf("%*.0f", 4,
305             pgtok(kstat_delta(oldvm, newvm, "anonfree")) / etime);
306         adjprintf("%*.0f", 4,
307             pgtok(kstat_delta(oldvm, newvm, "fspgin")) / etime);
308         adjprintf("%*.0f", 4,
309             pgtok(kstat_delta(oldvm, newvm, "fspgout")) / etime);
310         adjprintf("%*.0f\n", 4,
311             pgtok(kstat_delta(oldvm, newvm, "fsfree")) / etime);
312         (void) fflush(stdout);
313         return;
314     }

316     adjprintf("%*lu", 1, DELTA(s_sys.ss_sysinfo.runque) / sys_updates);
317     adjprintf("%*lu", 1, DELTA(s_sys.ss_sysinfo.waiting) / sys_updates);
318     adjprintf("%*lu", 1, DELTA(s_sys.ss_sysinfo.swpque) / sys_updates);
319     adjprintf("%*u", 6, pgtok((int)(DELTA(s_sys.ss_vminfo.swap_avail)
320         / vm_updates)));
321     adjprintf("%*u", 5, pgtok((int)(DELTA(s_sys.ss_vminfo.freemem)
322         / vm_updates)));
323     adjprintf("%*.0f", 3, swflag? 0 :
324         adjprintf("%*.0f", 3, swflag?
325             kstat_delta(oldvm, newvm, "swpin") / etime :
326             kstat_delta(oldvm, newvm, "pgrec") / etime);
327     adjprintf("%*.0f", 3, swflag? 0 :
328         adjprintf("%*.0f", 3, swflag?
329             kstat_delta(oldvm, newvm, "swpout") / etime :
330             (kstat_delta(oldvm, newvm, "hat_fault")
331             + kstat_delta(oldvm, newvm, "as_fault"))
332             / etime);
333     adjprintf("%*.0f", 2, pgtok(kstat_delta(oldvm, newvm, "pgpgin"))
334         / etime);
335     adjprintf("%*.0f", 2, pgtok(kstat_delta(oldvm, newvm, "pgpgout"))
336         / etime);
337     adjprintf("%*.0f", 2, pgtok(kstat_delta(oldvm, newvm, "dfree"))
338         / etime);
339     adjprintf("%*ld", 2, pgtok(new->s_sys.ss_deficit));
340     adjprintf("%*.0f", 2, kstat_delta(oldvm, newvm, "scan") / etime);

342     (void) snapshot_walk(SNAP_IODEVFS, old, new, show_disk, NULL);

344     count = df.if_max_iodevs - new->s_nr_iodevs;
345     while (count-- > 0)
346         adjprintf("%*d", 2, 0);

348     adjprintf("%*.0f", 4, kstat_delta(oldsys, newsys, "intr") / etime);
349     adjprintf("%*.0f", 4, kstat_delta(oldsys, newsys, "syscall") / etime);
350     adjprintf("%*.0f", 4, kstat_delta(oldsys, newsys, "pswitch") / etime);
351     adjprintf("%*.0f", 2,
352         kstat_delta(oldsys, newsys, "cpu_ticks_user") * percent_factor);
353     adjprintf("%*.0f", 2, kstat_delta(oldsys, newsys, "cpu_ticks_kernel")
354         * percent_factor);
355     adjprintf("%*.0f\n", 2, (kstat_delta(oldsys, newsys, "cpu_ticks_idle")
356         + kstat_delta(oldsys, newsys, "cpu_ticks_wait"))
357         * percent_factor);
358     (void) fflush(stdout);
359 }

_____unchanged_portion_omitted_____

416 static void
417 zero_out(char const *pretty)
418 {
419     (void) printf("%9llu %s\n", 0, pretty);

```

```

420 }

422 static void
423 #endif /* ! codereview */
424 dosum(struct sys_snapshot *ss)
425 {
426     uint64_t total_faults;
427     kstat_named_t *ksn;
428     long double nchttotal;
429     uint64_t nchhits;

431     zero_out("swap ins");
432     zero_out("swap outs");
433     zero_out("pages swapped in");
434     zero_out("pages swapped out");
435     sum_out("swap ins", &ss->ss_agg_vm, "swpin");
436     sum_out("swap outs", &ss->ss_agg_vm, "swpout");
437     sum_out("pages swapped in", &ss->ss_agg_vm, "pgswpin");
438     sum_out("pages swapped out", &ss->ss_agg_vm, "pgswpout");

439     ksn = kstat_data_lookup(&ss->ss_agg_vm, "hat_fault");
440     if (ksn == NULL) {
441         fail(0, "kstat_data_lookup('%s', 'hat_fault') failed",
442             ss->ss_agg_vm.ks_name);
443     }
444     total_faults = ksn->value.ui64;
445     ksn = kstat_data_lookup(&ss->ss_agg_vm, "as_fault");
446     if (ksn == NULL) {
447         fail(0, "kstat_data_lookup('%s', 'as_fault') failed",
448             ss->ss_agg_vm.ks_name);
449     }
450     total_faults += ksn->value.ui64;

451     (void) printf("%9llu total address trans. faults taken\n",
452         total_faults);

453     sum_out("page ins", &ss->ss_agg_vm, "pgin");
454     sum_out("page outs", &ss->ss_agg_vm, "pgout");
455     sum_out("pages paged in", &ss->ss_agg_vm, "pgpgin");
456     sum_out("pages paged out", &ss->ss_agg_vm, "pgpgout");
457     sum_out("total reclaims", &ss->ss_agg_vm, "pgrec");
458     sum_out("reclaims from free list", &ss->ss_agg_vm, "pgfrec");
459     sum_out("micro (hat) faults", &ss->ss_agg_vm, "hat_fault");
460     sum_out("minor (as) faults", &ss->ss_agg_vm, "as_fault");
461     sum_out("major faults", &ss->ss_agg_vm, "maj_fault");
462     sum_out("copy-on-write faults", &ss->ss_agg_vm, "cow_fault");
463     sum_out("zero fill page faults", &ss->ss_agg_vm, "zfod");
464     sum_out("pages examined by the clock daemon", &ss->ss_agg_vm, "scan");
465     sum_out("revolutions of the clock hand", &ss->ss_agg_vm, "rev");
466     sum_out("pages freed by the clock daemon", &ss->ss_agg_vm, "dfree");
467     sum_out("forks", &ss->ss_agg_sys, "sysfork");
468     sum_out("vforks", &ss->ss_agg_sys, "sysvfork");
469     sum_out("execs", &ss->ss_agg_sys, "sysexec");
470     sum_out("cpu context switches", &ss->ss_agg_sys, "pswitch");
471     sum_out("device interrupts", &ss->ss_agg_sys, "intr");
472     sum_out("traps", &ss->ss_agg_sys, "trap");
473     sum_out("system calls", &ss->ss_agg_sys, "syscall");

474     nchttotal = (long double) ss->ss_nc.ncs_hits.value.ui64 +
475         (long double) ss->ss_nc.ncs_misses.value.ui64;
476     nchhits = ss->ss_nc.ncs_hits.value.ui64;
477     (void) printf("%9.0Lf total name lookups (cache hits %.0Lf%%)\n",
478         nchttotal, nchhits / denom(nchttotal) * 100);

479     sum_out("user cpu", &ss->ss_agg_sys, "cpu_ticks_user");
480     sum_out("system cpu", &ss->ss_agg_sys, "cpu_ticks_kernel");

```

```

482     sum_out("idle cpu", &ss->ss_agg_sys, "cpu_ticks_idle");
483     sum_out("wait cpu", &ss->ss_agg_sys, "cpu_ticks_wait");
484 }
_____unchanged_portion_omitted_____

```

```

*****
66813 Thu Feb 18 08:40:38 2016
new/usr/src/uts/common/disp/disp.c
6583 remove whole-process swapping
*****
_____unchanged portion omitted_____
75 static void disp_dq_alloc(struct disp_queue_info *dptr, int numpris,
76 disp_t *dp);
77 static void disp_dq_assign(struct disp_queue_info *dptr, int numpris);
78 static void disp_dq_free(struct disp_queue_info *dptr);

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

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

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

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

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

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

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

123 /*
124 * Parameter that determines how long (in nanoseconds) a thread must
125 * be sitting on a run queue before it can be stolen by another CPU
126 * to reduce migrations. The interval is in nanoseconds.
127 */
128 * The nsteal_nsec should be set by platform code cmp_set_nsteal_interval()
129 * to an appropriate value. nsteal_nsec is set to NOSTEAL_UNINITIALIZED
130 * here indicating it is uninitialized.
131 * Setting nsteal_nsec to 0 effectively disables the nsteal 'protection'.
132 *

```

```

133 */
134 #define NOSTEAL_UNINITIALIZED (-1)
135 hrtime_t nsteal_nsec = NOSTEAL_UNINITIALIZED;
136 extern void cmp_set_nsteal_interval(void);

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

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

143 static void cpu_dispqalloc(int numpris);

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

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

156 void swtch_to(kthread_t *);

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

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

171 ASSERT(MUTEX_HELD(&cpu_lock));

173 newnglobpris = maxglobpri + 1 + LOCK_LEVEL;

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

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

186 /*
187 * compute new interrupt thread base priority
188 */
189 intr_pri = maxglobpri;
190 if (only_intr_kpreempt) {
191 kpreemptpri = intr_pri + 1;
192 if (kpqpri == KPQPRI)
193 kpqpri = kpreemptpri;
194 }
195 v.v_nglobpris = newnglobpris;
196 }
197 }
_____unchanged portion omitted_____

```

```

694 extern kthread_t *thread_unpin();

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

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

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

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

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

739     /*
740     * If there is nothing to run, look at what's runnable on other queues.
741     * Choose the idle thread if the CPU is quiesced.
742     * Note that CPUs that have the CPU_OFFLINE flag set can still run
743     * interrupt threads, which will be the only threads on the CPU's own
744     * queue, but cannot run threads from other queues.
745     */
746     if (pri == -1) {
747         if (!(cpup->cpu_flags & CPU_OFFLINE)) {
748             disp_lock_exit(&dp->disp_lock);
749             if ((tp = disp_getwork(cpup)) == NULL ||
750                 tp == T_DONTSTEAL) {
751                 tp = cpup->cpu_idle_thread;
752                 (void) splhigh();
753                 THREAD_ONPROC(tp, cpup);
754                 cpup->cpu_dispthread = tp;
755                 cpup->cpu_dispatch_pri = -1;
756                 cpup->cpu_runrun = cpup->cpu_kprunrun = 0;
757                 cpup->cpu_chosen_level = -1;
758             }

```

```

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

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

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

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

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

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

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

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

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

818     /*
819     * Set TS_DONT_SWAP flag to prevent another processor from swapping
820     * out this thread before we have a chance to run it.
821     * While running, it is protected against swapping by t_lock.
822     */
823     tp->t_schedflag |= TS_DONT_SWAP;
824     cpup->cpu_dispthread = tp; /* protected by spl only */
825     cpup->cpu_dispatch_pri = pri;
826     ASSERT(pri == DISP_PRIO(tp));
827 }

```

```

818     thread_onproc(tp, cpup);          /* set t_state to TS_ONPROC */
819     disp_lock_exit_high(&dp->disp_lock); /* drop run queue lock */

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

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

828     return (tp);
829 }

```

unchanged portion omitted

```

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

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

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

1194     /*
1195      * If thread is "swapped" or on the swap queue don't
1196      * queue it, but wake sched.
1197      */
1198     if ((tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ)) != TS_LOAD) {
1199         disp_swapped_setrun(tp);
1200         return;
1201     }

```

```

1185     self = (tp == curthread);

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

1192     tpri = DISP_PRI0(tp);
1193     if (ncpus == 1)
1194         cp = tp->t_cpu;
1195     else if (!bound) {
1196         if (tpri >= kpqpri) {
1197             setkpdq(tp, SETKP_BACK);
1198             return;
1199         }

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

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

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

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

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

1243             if (RUNQ_LEN(newcp, tpri) < qlen) {
1244                 DTRACE_PROBE3(runq_balance,
1245                               kthread_t *, tp,
1246                               cpu_t *, cp, cpu_t *, newcp);
1247                 cp = newcp;
1248             }
1249         }
1250     } else {

```

```

1251         /*
1252          * Migrate to a cpu in the new partition.
1253          */
1254         cp = disp_lowpri_cpu(tp->t_cpupart->cp_cpulist,
1255                             tp->t_lpl, tp->t_pri, NULL);
1256     }
1257     ASSERT((cp->cpu_flags & CPU_QUIESCED) == 0);
1258 } else {
1259     /*
1260      * It is possible that t_weakbound_cpu != t_bound_cpu (for
1261      * a short time until weak binding that existed when the
1262      * strong binding was established has dropped) so we must
1263      * favour weak binding over strong.
1264      */
1265     cp = tp->t_weakbound_cpu ?
1266         tp->t_weakbound_cpu : tp->t_bound_cpu;
1267 }
1268 /*
1269  * A thread that is ONPROC may be temporarily placed on the run queue
1270  * but then chosen to run again by disp. If the thread we're placing on
1271  * the queue is in TS_ONPROC state, don't set its t_waitrq until a
1272  * replacement process is actually scheduled in swtch(). In this
1273  * situation, curthread is the only thread that could be in the ONPROC
1274  * state.
1275  */
1276 if ((!self) && (tp->t_waitrq == 0)) {
1277     hrttime_t curtime;
1278
1279     curtime = gethrtime_unscaled();
1280     (void) cpu_update_pct(tp, curtime);
1281     tp->t_waitrq = curtime;
1282 } else {
1283     (void) cpu_update_pct(tp, gethrtime_unscaled());
1284 }
1285
1286 dp = cp->cpu_disp;
1287 disp_lock_enter_high(&dp->disp_lock);
1288
1289 DTRACE_SCHED3(enqueue, kthread_t *, tp, disp_t *, dp, int, 0);
1290 TRACE_3(TR_FAC_DISP, TR_BACKQ, "setbackdq:pri %d cpu %p tid %p",
1291         tpri, cp, tp);
1292
1293 #ifndef NPROBE
1294     /* Kernel probe */
1295     if (tnf_tracing_active)
1296         tn timer_queue(tp, cp, tpri);
1297 #endif /* NPROBE */
1298
1299     ASSERT(tpri >= 0 && tpri < dp->disp_npri);
1300
1301     THREAD_RUN(tp, &dp->disp_lock); /* set t_state to TS_RUN */
1302     tp->t_disp_queue = dp;
1303     tp->t_link = NULL;
1304
1305     dq = &dp->disp_q[tpri];
1306     dp->disp_nrunnable++;
1307     if (!bound)
1308         dp->disp_steal = 0;
1309     membar_enter();
1310
1311     if (dq->dq_sruncnt++ != 0) {
1312         ASSERT(dq->dq_first != NULL);
1313         dq->dq_last->t_link = tp;
1314         dq->dq_last = tp;
1315     } else {
1316         ASSERT(dq->dq_first == NULL);

```

```

1317         ASSERT(dq->dq_last == NULL);
1318         dq->dq_first = dq->dq_last = tp;
1319         BT_SET(dp->disp_gactmap, tpri);
1320         if (tpri > dp->disp_maxrunpri) {
1321             dp->disp_maxrunpri = tpri;
1322             membar_enter();
1323             cpu_resched(cp, tpri);
1324         }
1325     }
1326
1327     if (!bound && tpri > dp->disp_max_unbound_pri) {
1328         if (self && dp->disp_max_unbound_pri == -1 && cp == CPU) {
1329             /*
1330              * If there are no other unbound threads on the
1331              * run queue, don't allow other CPUs to steal
1332              * this thread while we are in the middle of a
1333              * context switch. We may just switch to it
1334              * again right away. CPU_DISP_DONTSTEAL is cleared
1335              * in swtch and swtch_to.
1336              */
1337             cp->cpu_disp_flags |= CPU_DISP_DONTSTEAL;
1338         }
1339         dp->disp_max_unbound_pri = tpri;
1340     }
1341     (*disp_enq_thread)(cp, bound);
1342 }
1343
1344 /*
1345  * Put the specified thread on the front of the dispatcher
1346  * queue corresponding to its current priority.
1347  *
1348  * Called with the thread in transition, onproc or stopped state
1349  * and locked (transition implies locked) and at high spl.
1350  * Returns with the thread in TS_RUN state and still locked.
1351  */
1352 void
1353 setfrontdq(kthread_t *tp)
1354 {
1355     disp_t      *dp;
1356     dispq_t     *dq;
1357     cpu_t       *cp;
1358     pri_t       tpri;
1359     int         bound;
1360
1361     ASSERT(THREAD_LOCK_HELD(tp));
1362     ASSERT((tp->t_schedflag & TS_ALLSTART) == 0);
1363     ASSERT(!thread_on_queue(tp)); /* make sure tp isn't on a runq */
1364
1365     /*
1366      * If thread is "swapped" or on the swap queue don't
1367      * queue it, but wake sched.
1368      */
1369     if ((tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ)) != TS_LOAD) {
1370         disp_swapped_setrun(tp);
1371         return;
1372     }
1373
1374     if (tp->t_bound_cpu || tp->t_weakbound_cpu)
1375         bound = 1;
1376     else
1377         bound = 0;
1378
1379     tpri = DISP_PRIO(tp);
1380     if (ncpus == 1)
1381         cp = tp->t_cpu;
1382     else if (!bound) {

```



```

1374     if (tpri >= kqppri) {
1375         setkpdq(tp, SETKP_FRONT);
1376         return;
1377     }
1378     cp = tp->t_cpu;
1379     if (tp->t_cpupart == cp->cpu_part) {
1380         /*
1381          * We'll generally let this thread continue to run
1382          * where it last ran, but will consider migration if:
1383          * - The thread last ran outside it's home lgroup.
1384          * - The CPU where it last ran is the target of an
1385          *   offline request (a thread_nomigrate() on the in
1386          *   motion CPU relies on this when forcing a preempt).
1387          * - The thread isn't the highest priority thread where
1388          *   it last ran, and it is considered not likely to
1389          *   have significant cache warmth.
1390          */
1391         if ((!LGRP_CONTAINS_CPU(tp->t_lpl->lpl_lgrp, cp)) ||
1392             (cp == cpu_inmotion)) {
1393             cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri,
1394                                 (tp == curthread) ? cp : NULL);
1395         } else if ((tpri < cp->cpu_disp->disp_maxrunpri) &&
1396                    (!THREAD_HAS_CACHE_WARMTH(tp))) {
1397             cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri,
1398                                 NULL);
1399         }
1400     } else {
1401         /*
1402          * Migrate to a cpu in the new partition.
1403          */
1404         cp = disp_lowpri_cpu(tp->t_cpupart->cp_cpulist,
1405                             tp->t_lpl, tp->t_pri, NULL);
1406     }
1407     ASSERT((cp->cpu_flags & CPU_QUIESCED) == 0);
1408 } else {
1409     /*
1410      * It is possible that t_weakbound_cpu != t_bound_cpu (for
1411      * a short time until weak binding that existed when the
1412      * strong binding was established has dropped) so we must
1413      * favour weak binding over strong.
1414      */
1415     cp = tp->t_weakbound_cpu ?
1416         tp->t_weakbound_cpu : tp->t_bound_cpu;
1417 }
1418
1419 /*
1420  * A thread that is ONPROC may be temporarily placed on the run queue
1421  * but then chosen to run again by disp.  If the thread we're placing on
1422  * the queue is in TS_ONPROC state, don't set its t_waitrq until a
1423  * replacement process is actually scheduled in swtch().  In this
1424  * situation, curthread is the only thread that could be in the ONPROC
1425  * state.
1426  */
1427 if ((tp != curthread) && (tp->t_waitrq == 0)) {
1428     hrttime_t curtime;
1429
1430     curtime = gethrtime_unscaled();
1431     (void) cpu_update_pct(tp, curtime);
1432     tp->t_waitrq = curtime;
1433 } else {
1434     (void) cpu_update_pct(tp, gethrtime_unscaled());
1435 }
1436
1437 dp = cp->cpu_disp;
1438 disp_lock_enter_high(&dp->disp_lock);

```

```

1440     TRACE_2(TR_FAC_DISP, TR_FRONTQ, "frontq:pri %d tid %p", tpri, tp);
1441     DTRACE_SCHED3(enqueue, kthread_t *, tp, disp_t *, dp, int, 1);
1442
1443 #ifndef NPROBE
1444     /* Kernel probe */
1445     if (tnf_tracing_active)
1446         tnf_thread_queue(tp, cp, tpri);
1447 #endif /* NPROBE */
1448
1449     ASSERT(tpri >= 0 && tpri < dp->disp_npri);
1450
1451     THREAD_RUN(tp, &dp->disp_lock);      /* set TS_RUN state and lock */
1452     tp->t_disp_queue = dp;
1453
1454     dq = &dp->disp_q[tpri];
1455     dp->disp_nrunnable++;
1456     if (!bound)
1457         dp->disp_steal = 0;
1458     membar_enter();
1459
1460     if (dq->dq_sruncnt++ != 0) {
1461         ASSERT(dq->dq_last != NULL);
1462         tp->t_link = dq->dq_first;
1463         dq->dq_first = tp;
1464     } else {
1465         ASSERT(dq->dq_last == NULL);
1466         ASSERT(dq->dq_first == NULL);
1467         tp->t_link = NULL;
1468         dq->dq_first = dq->dq_last = tp;
1469         BT_SET(dp->disp_qactmap, tpri);
1470         if (tpri > dp->disp_maxrunpri) {
1471             dp->disp_maxrunpri = tpri;
1472             membar_enter();
1473             cpu_resched(cp, tpri);
1474         }
1475     }
1476
1477     if (!bound && tpri > dp->disp_max_unbound_pri) {
1478         if (tp == curthread && dp->disp_max_unbound_pri == -1 &&
1479             cp == CPU) {
1480             /*
1481              * If there are no other unbound threads on the
1482              * run queue, don't allow other CPUs to steal
1483              * this thread while we are in the middle of a
1484              * context switch. We may just switch to it
1485              * again right away. CPU_DISP_DONTSTEAL is cleared
1486              * in swtch and swtch_to.
1487              */
1488             cp->cpu_disp_flags |= CPU_DISP_DONTSTEAL;
1489         }
1490         dp->disp_max_unbound_pri = tpri;
1491     }
1492     (*disp_enq_thread)(cp, bound);
1493 }
1494
1495 unchanged_portion_omitted
1496
1497 /*
1498  * Remove a thread from the dispatcher queue if it is on it.
1499  * It is not an error if it is not found but we return whether
1500  * or not it was found in case the caller wants to check.
1501  */
1502 int
1503 dispdq(kthread_t *tp)
1504 {
1505     disp_t      *dp;
1506     dispq_t     *dq;

```

```

1583     kthread_t      *rp;
1584     kthread_t      *trp;
1585     kthread_t      **ptp;
1586     int             tpri;

1588     ASSERT(THREAD_LOCK_HELD(tp));

1590     if (tp->t_state != TS_RUN)
1591         return (0);

1620     /*
1621     * The thread is "swapped" or is on the swap queue and
1622     * hence no longer on the run queue, so return true.
1623     */
1624     if ((tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ)) != TS_LOAD)
1625         return (1);

1593     tpri = DISP_PRI0(tp);
1594     dp = tp->t_disp_queue;
1595     ASSERT(tpri < dp->disp_npri);
1596     dq = &dp->disp_q[tpri];
1597     ptp = &dq->dq_first;
1598     rp = *ptp;
1599     trp = NULL;

1601     ASSERT(dq->dq_last == NULL || dq->dq_last->t_link == NULL);

1603     /*
1604     * Search for thread in queue.
1605     * Double links would simplify this at the expense of disp/setrun.
1606     */
1607     while (rp != tp && rp != NULL) {
1608         trp = rp;
1609         ptp = &trp->t_link;
1610         rp = trp->t_link;
1611     }

1613     if (rp == NULL) {
1614         panic("dispdeq: thread not on queue");
1615     }

1617     DTRACE_SCHED2(dequeue, kthread_t *, tp, disp_t *, dp);

1619     /*
1620     * Found it so remove it from queue.
1621     */
1622     if ((*ptp = rp->t_link) == NULL)
1623         dq->dq_last = trp;

1625     dp->disp_nrunnable--;
1626     if (--dq->dq_sruncnt == 0) {
1627         dp->disp_qactmap[tpri >> BT_ULSHIFT] &= ~BT_BIW(tpri);
1628         if (dp->disp_nrunnable == 0) {
1629             dp->disp_max_unbound_pri = -1;
1630             dp->disp_maxrunpri = -1;
1631         } else if (tpri == dp->disp_maxrunpri) {
1632             int ipri;

1634             ipri = bt_gethighbit(dp->disp_qactmap,
1635                 dp->disp_maxrunpri >> BT_ULSHIFT);
1636             if (ipri < dp->disp_max_unbound_pri)
1637                 dp->disp_max_unbound_pri = ipri;
1638             dp->disp_maxrunpri = ipri;
1639         }
1640     }
1641     tp->t_link = NULL;

```

```

1642     THREAD_TRANSITION(tp);          /* put in intermediate state */
1643     return (1);
1644 }

1681 /*
1682 * dq_sruncnt and dq_srundec are public functions for
1683 * incrementing/decrementing the sruncnts when a thread on
1684 * a dispatcher queue is made schedulable/unschedulable by
1685 * resetting the TS_LOAD flag.
1686 *
1687 * The caller MUST have the thread lock and therefore the dispatcher
1688 * queue lock so that the operation which changes
1689 * the flag, the operation that checks the status of the thread to
1690 * determine if it's on a disp queue AND the call to this function
1691 * are one atomic operation with respect to interrupts.
1692 */

1694 /*
1695 * Called by sched AFTER TS_LOAD flag is set on a swapped, runnable thread.
1696 */
1697 void
1698 dq_sruncnt(kthread_t *t)
1699 {
1700     ASSERT(t->t_state == TS_RUN);
1701     ASSERT(t->t_schedflag & TS_LOAD);

1703     THREAD_TRANSITION(t);
1704     setfrontdq(t);
1705 }

1707 /*
1708 * See comment on calling conventions above.
1709 * Called by sched BEFORE TS_LOAD flag is cleared on a runnable thread.
1710 */
1711 void
1712 dq_srundec(kthread_t *t)
1713 {
1714     ASSERT(t->t_schedflag & TS_LOAD);

1716     (void) dispdeq(t);
1717     disp_swapped_enq(t);
1718 }

1720 /*
1721 * Change the dispatcher lock of thread to the "swapped_lock"
1722 * and return with thread lock still held.
1723 *
1724 * Called with thread_lock held, in transition state, and at high spl.
1725 */
1726 void
1727 disp_swapped_enq(kthread_t *tp)
1728 {
1729     ASSERT(THREAD_LOCK_HELD(tp));
1730     ASSERT(tp->t_schedflag & TS_LOAD);

1732     switch (tp->t_state) {
1733     case TS_RUN:
1734         disp_lock_enter_high(&swapped_lock);
1735         THREAD_SWAP(tp, &swapped_lock); /* set TS_RUN state and lock */
1736         break;
1737     case TS_ONPROC:
1738         disp_lock_enter_high(&swapped_lock);
1739         THREAD_TRANSITION(tp);
1740         wake_sched_sec = 1; /* tell clock to wake sched */
1741         THREAD_SWAP(tp, &swapped_lock); /* set TS_RUN state and lock */

```

```

1742         break;
1743     default:
1744         panic("disp_swapped: tp: %p bad t_state", (void *)tp);
1745     }
1746 }

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

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

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

1665     ASSERT(THREAD_LOCK_HELD(tp));

1667     if (tp->t_state != TS_ONPROC)
1668         return;
1669     cpup = tp->t_disp_queue->disp_cpu; /* CPU thread dispatched to */

```

```

1670     max_pri = cpup->cpu_disp->disp_maxrunpri; /* best pri of that CPU */
1671     max_run_pri = CP_MAXRUNPRI(cpup->cpu_part);
1672     if (max_pri < max_run_pri)
1673         max_pri = max_run_pri;

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

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

1692     DTRACE_SCHED1(surrender, kthread_t *, tp);

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

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

2015     ASSERT(THREAD_LOCK_HELD(tp));

2017     /*
2018      * Don't do anything if the thread is not bound, or
2019      * currently not runnable.
2020      * currently not runnable or swapped out.
2021      */
2021     if (tp->t_bound_cpu == NULL ||
2022         tp->t_state != TS_RUN)
2023         return;
2024     tpri = DISP_PRIO(tp);
2025     dp = tp->t_bound_cpu->cpu_disp;

```

```

2027     ASSERT(tpri >= 0 && tpri < dp->disp_npri);
2028     if (tpri > dp->disp_max_unbound_pri)
2029         dp->disp_max_unbound_pri = tpri;
2030 }

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

2052     disp_lock_enter(&dp->disp_lock);

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

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

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

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

2086         /*
2087          * We've got some unbound threads on this queue, so turn
2088          * the allbound flag off now.
2089          */
2090         allbound = B_FALSE;

2092         /*

```

```

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

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

2134         nsteal = nsteal_nsec;
2135         if (nsteal == 0)
2136             break;

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

2145         /*
2146          * Steal immediately if the time spent on this run queue is more
2147          * than allowed nsteal delay.
2148          *
2149          * Negative rqtime check is needed here to avoid infinite
2150          * stealing delays caused by unlikely but not impossible
2151          * drifts between CPU times on different CPUs.
2152          */
2153         if (rqtime > nsteal || rqtime < 0)
2154             break;

2156         DTRACE_PROBE4(nsteal, kthread_t *, tp,
2157                     cpu_t *, tcp, cpu_t *, cp, hrttime_t, rqtime);
2158         scalehrtime(&now);

```

```

2159      /*
2160      * Calculate when this thread becomes stealable
2161      */
2162      now += (nosteal - rqtime);

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

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

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

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

2199 #ifdef DEBUG
2200     {
2201         int    thread_was_on_queue;

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

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

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

2356    tp->t_schedflag |= TS_DONT_SWAP;

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

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

```

```

2223    cp->cpu_dispatch_pri = pri;

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

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

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

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

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

2258    return (tp);
2259 }
_____unchanged_portion_omitted_____

```

```

*****
79977 Thu Feb 18 08:40:38 2016
new/usr/src/uts/common/disp/fss.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

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

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

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

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

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

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

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

350 static id_t      fss_cid;

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

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

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

369 static int      fss_enterclass(kthread_t *, id_t, void *, cred_t *, void *);
370 static void     fss_exitclass(void *);
371 static int      fss_canexit(kthread_t *, cred_t *);
372 static int      fss_fork(kthread_t *, kthread_t *, void *);

```

```

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

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

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

2136 /*
2143  * fss_swapin() returns -1 if the thread is loaded or is not eligible to be
2144  * swapped in. Otherwise, it returns the thread's effective priority based
2145  * on swapout time and size of process (0 <= epri <= 0 SHRT_MAX).
2146  */

```

```

2147 /*ARGSUSED*/
2148 static pri_t
2149 fss_swapin(kthread_t *t, int flags)
2150 {
2151     fssproc_t *fssproc = FSSPROC(t);
2152     long epri = -1;
2153     proc_t *pp = ttoproc(t);
2154
2155     ASSERT(THREAD_LOCK_HELD(t));
2156
2157     if (t->t_state == TS_RUN && (t->t_schedflag & TS_LOAD) == 0) {
2158         time_t swapout_time;
2159
2160         swapout_time = (ddi_get_lbolt() - t->t_stime) / hz;
2161         if (INHERITED(t) || (fssproc->fss_flags & FSSKPRI)) {
2162             epri = (long)DISP_PRIO(t) + swapout_time;
2163         } else {
2164             /*
2165              * Threads which have been out for a long time,
2166              * have high user mode priority and are associated
2167              * with a small address space are more deserving.
2168              */
2169             epri = fssproc->fss_umdpr;
2170             ASSERT(epri >= 0 && epri <= fss_maxumdpr);
2171             epri += swapout_time - pp->p_swrss / nz(maxpgio)/2;
2172         }
2173         /*
2174          * Scale epri so that SHRT_MAX / 2 represents zero priority.
2175          */
2176         epri += SHRT_MAX / 2;
2177         if (epri < 0)
2178             epri = 0;
2179         else if (epri > SHRT_MAX)
2180             epri = SHRT_MAX;
2181     }
2182     return ((pri_t)epri);
2183 }
2184
2185 /*
2186 * fss_swapout() returns -1 if the thread isn't loaded or is not eligible to
2187 * be swapped out. Otherwise, it returns the thread's effective priority
2188 * based on if the swapper is in softswap or hardswap mode.
2189 */
2190 static pri_t
2191 fss_swapout(kthread_t *t, int flags)
2192 {
2193     fssproc_t *fssproc = FSSPROC(t);
2194     long epri = -1;
2195     proc_t *pp = ttoproc(t);
2196     time_t swapin_time;
2197
2198     ASSERT(THREAD_LOCK_HELD(t));
2199
2200     if (INHERITED(t) ||
2201         (fssproc->fss_flags & FSSKPRI) ||
2202         (t->t_proc_flag & TP_LWPEXIT) ||
2203         (t->t_state & (TS_ZOMB|TS_FREE|TS_STOPPED|TS_ONPROC|TS_WAIT)) ||
2204         !(t->t_schedflag & TS_LOAD) ||
2205         !(SWAP_OK(t)))
2206         return (-1);
2207
2208     ASSERT(t->t_state & (TS_SLEEP | TS_RUN));
2209
2210     swapin_time = (ddi_get_lbolt() - t->t_stime) / hz;
2211
2212     if (flags == SOFTSWAP) {

```

```

2213         if (t->t_state == TS_SLEEP && swapin_time > maxslp) {
2214             epri = 0;
2215         } else {
2216             return ((pri_t)epri);
2217         }
2218     } else {
2219         pri_t pri;
2220
2221         if ((t->t_state == TS_SLEEP && swapin_time > fss_minslp) ||
2222             (t->t_state == TS_RUN && swapin_time > fss_minrun)) {
2223             pri = fss_maxumdpr;
2224             epri = swapin_time -
2225                 (rm_asrss(pp->p_as) / nz(maxpgio)/2) - (long)pri;
2226         } else {
2227             return ((pri_t)epri);
2228         }
2229     }
2230
2231     /*
2232     * Scale epri so that SHRT_MAX / 2 represents zero priority.
2233     */
2234     epri += SHRT_MAX / 2;
2235     if (epri < 0)
2236         epri = 0;
2237     else if (epri > SHRT_MAX)
2238         epri = SHRT_MAX;
2239
2240     return ((pri_t)epri);
2241 }
2242
2243 /*
2244 * If thread is currently at a kernel mode priority (has slept) and is
2245 * returning to the userland we assign it the appropriate user mode priority
2246 * and time quantum here. If we're lowering the thread's priority below that
2247 * of other runnable threads then we will set runrun via cpu_surrender() to
2248 * cause preemption.
2249 */
2250 static void
2251 fss_trapret(kthread_t *t)
2252 {
2253     fssproc_t *fssproc = FSSPROC(t);
2254     cpu_t *cp = CPU;
2255
2256     ASSERT(THREAD_LOCK_HELD(t));
2257     ASSERT(t == curthread);
2258     ASSERT(cp->cpu_dispthread == t);
2259     ASSERT(t->t_state == TS_ONPROC);
2260
2261     t->t_kpri_req = 0;
2262     if (fssproc->fss_flags & FSSKPRI) {
2263         /*
2264          * If thread has blocked in the kernel
2265          */
2266         THREAD_CHANGE_PRI(t, fssproc->fss_umdpr);
2267         cp->cpu_dispatch_pri = DISP_PRIO(t);
2268         ASSERT(t->t_pri >= 0 && t->t_pri <= fss_maxglobpri);
2269         fssproc->fss_flags &= ~FSSKPRI;
2270     }
2271     if (DISP_MUST_SURRENDER(t))
2272         cpu_surrender(t);
2273
2274     /*
2275     * Swapout lwp if the swapper is waiting for this thread to reach
2276     * a safe point.
2277     */

```

```

2279     if (t->t_schedflag & TS_SWAPENQ) {
2280         thread_unlock(t);
2281         swapout_lwp(ttolwp(t));
2282         thread_lock(t);
2283     }
2167 }

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

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

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

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

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

2210     /*
2211      * If preempted in user-land mark the thread as swappable because it
2212      * cannot be holding any kernel locks.
2213      */
2214     ASSERT(t->t_schedflag & TS_DONT_SWAP);
2215     if (lwp != NULL && lwp->lwp_state == LWP_USER)
2216         t->t_schedflag &= ~TS_DONT_SWAP;

2235     /*
2211     * Check to see if we're doing "preemption control" here. If
2212     * we are, and if the user has requested that this thread not
2213     * be preempted, and if preemptions haven't been put off for
2214     * too long, let the preemption happen here but try to make
2215     * sure the thread is rescheduled as soon as possible. We do
2216     * this by putting it on the front of the highest priority run
2217     * queue in the FSS class. If the preemption has been put off
2218     * for too long, clear the "nopreempt" bit and let the thread
2219     * be preempted.

```

```

2220     /*
2221     if (t->t_schedctl && schedctl_get_nopreempt(t)) {
2222         if (fssproc->fss_timeleft > -SC_MAX_TICKS) {
2223             DTRACE_SCHED1(schedctl_nopreempt, kthread_t *, t);
2224             if (!(fssproc->fss_flags & FSSKPRI)) {
2225                 /*
2226                 * If not already remembered, remember current
2227                 * priority for restoration in fss_yield().
2228                 */
2229                 if (!(fssproc->fss_flags & FSSRESTORE)) {
2230                     fssproc->fss_scpr = t->t_pri;
2231                     fssproc->fss_flags |= FSSRESTORE;
2232                 }
2233                 THREAD_CHANGE_PRI(t, fss_maxumdpr);
2234                 t->t_schedflag |= TS_DONT_SWAP;
2235             }
2236             schedctl_set_yield(t, 1);
2237             setfrontdq(t);
2238             return;
2239         } else {
2240             if (fssproc->fss_flags & FSSRESTORE) {
2241                 THREAD_CHANGE_PRI(t, fssproc->fss_scpr);
2242                 fssproc->fss_flags &= ~FSSRESTORE;
2243             }
2244             schedctl_set_nopreempt(t, 0);
2245             DTRACE_SCHED1(schedctl_preempt, kthread_t *, t);
2246             /*
2247             * Fall through and be preempted below.
2248             */
2249         }
2250     }

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

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

_____unchanged_portion_omitted_____

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

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

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

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

```



```

2313     fss_inactive(t);

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

2341     /*
2342     * A tick interrupt has occurred on a running thread. Check to see if our
2343     * time slice has expired.
2344     * time slice has expired. We must also clear the TS_DONT_SWAP flag in
2345     * t_schedflag if the thread is eligible to be swapped out.
2346     */
2347     static void
2348     fss_tick(kthread_t *t)
2349     {
2350         fssproc_t *fssproc;
2351         fssproj_t *fssproj;
2352         klwp_t *lwp;
2353         boolean_t call_cpu_surrender = B_FALSE;
2354         boolean_t cpucaps_enforce = B_FALSE;

2355         ASSERT(MUTEX_HELD(&(ttoproc(t))->p_lock));

2356         /*
2357         * It's safe to access fsspsset and fssproj structures because we're
2358         * holding our p_lock here.
2359         */
2360         thread_lock(t);
2361         fssproc = FSSPROC(t);
2362         fssproj = FSSPROC2FSSPROJ(fssproc);
2363         if (fssproj != NULL) {
2364             fsspsset_t *fsspsset = FSSPROJ2FSSPSET(fssproj);
2365             disp_lock_enter_high(&fsspsset->fssps_displ);
2366             fssproj->fssp_ticks += fss_nice_tick[fssproc->fss_nice];
2367             fssproj->fssp_tick_cnt++;
2368             fssproc->fss_ticks++;
2369             disp_lock_exit_high(&fsspsset->fssps_displ);
2370         }

2371         /*
2372         * Keep track of thread's project CPU usage. Note that projects
2373         * get charged even when threads are running in the kernel.
2374         * Do not surrender CPU if running in the SYS class.

```

```

2375     /*
2376     if (CPUCAPS_ON()) {
2377         cpucaps_enforce = cpucaps_charge(t,
2378             &fssproc->fss_caps, CPUCAPS_CHARGE_ENFORCE) &&
2379             !(fssproc->fss_flags & FSSKPRI);
2380     }

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

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

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

2414             /*
2415             * When the priority of a thread is changed, it may
2416             * be necessary to adjust its position on a sleep queue
2417             * or dispatch queue. The function thread_change_pri
2418             * accomplishes this.
2419             */
2420             if (thread_change_pri(t, new_pri, 0)) {
2421                 if ((t->t_schedflag & TS_LOAD) &&
2422                     (lwp = t->t_lwp) &&
2423                     lwp->lwp_state == LWP_USER)
2424                     t->t_schedflag &= ~TS_DONT_SWAP;
2425                 fssproc->fss_timeleft = fss_quantum;
2426             } else {
2427                 call_cpu_surrender = B_TRUE;
2428             }
2429         } else if (t->t_state == TS_ONPROC &&
2430             t->t_pri < t->t_disp_queue->disp_maxrunpri) {
2431             /*
2432             * If there is a higher-priority thread which is
2433             * waiting for a processor, then thread surrenders
2434             * the processor.
2435             */
2436             call_cpu_surrender = B_TRUE;
2437         }
2438     }

2439     if (cpucaps_enforce && 2 * fssproc->fss_timeleft > fss_quantum) {

```

```

2437      /*
2438      * The thread used more than half of its quantum, so assume that
2439      * it used the whole quantum.
2440      *
2441      * Update thread's priority just before putting it on the wait
2442      * queue so that it gets charged for the CPU time from its
2443      * quantum even before that quantum expires.
2444      */
2445      fss_newpri(fssproc, B_FALSE);
2446      if (t->t_pri != fssproc->fss_umdpri)
2447          fss_change_priority(t, fssproc);
2448
2449      /*
2450      * We need to call cpu_surrender for this thread due to cpucaps
2451      * enforcement, but fss_change_priority may have already done
2452      * so. In this case FSSBACKQ is set and there is no need to call
2453      * cpu-surrender again.
2454      */
2455      if (!(fssproc->fss_flags & FSSBACKQ))
2456          call_cpu_surrender = B_TRUE;
2457  }
2458
2459  if (call_cpu_surrender) {
2460      fssproc->fss_flags |= FSSBACKQ;
2461      cpu_surrender(t);
2462  }
2463
2464      thread_unlock_nopreempt(t);    /* clock thread can't be preempted */
2465  }
2466
2467  /*
2468  * Processes waking up go to the back of their queue. We don't need to assign
2469  * a time quantum here because thread is still at a kernel mode priority and
2470  * the time slicing is not done for threads running in the kernel after
2471  * sleeping. The proper time quantum will be assigned by fss_trapret before the
2472  * thread returns to user mode.
2473  */
2474  static void
2475  fss_wakeup(kthread_t *t)
2476  {
2477      fssproc_t *fssproc;
2478
2479      ASSERT(THREAD_LOCK_HELD(t));
2480      ASSERT(t->t_state == TS_SLEEP);
2481
2482      fss_active(t);
2483
2484      t->t_stime = ddi_get_lbolt();    /* time stamp for the swapper */
2485      fssproc = FSSPROC(t);
2486      fssproc->fss_flags &= ~FSSBACKQ;
2487
2488      if (fssproc->fss_flags & FSSKPRI) {
2489          /*
2490           * If we already have a kernel priority assigned, then we
2491           * just use it.
2492           */
2493          setbackdq(t);
2494      } else if (t->t_kpri_req) {
2495          /*
2496           * Give thread a priority boost if we were asked.
2497           */
2498          fssproc->fss_flags |= FSSKPRI;
2499          THREAD_CHANGE_PRI(t, minclsyspri);
2500          setbackdq(t);
2501          t->t_trapret = 1;    /* so that fss_trapret will run */
2502          aston(t);

```

```

2502      } else {
2503          /*
2504           * Otherwise, we recalculate the priority.
2505           */
2506          if (t->t_disp_time == ddi_get_lbolt()) {
2507              setfrontdq(t);
2508          } else {
2509              fssproc->fss_timeleft = fss_quantum;
2510              THREAD_CHANGE_PRI(t, fssproc->fss_umdpri);
2511              setbackdq(t);
2512          }
2513      }
2514  }

```

unchanged\_portion\_omitted

\*\*\*\*\*

42975 Thu Feb 18 08:40:38 2016

new/usr/src/uts/common/disp/fx.c

6583 remove whole-process swapping

\*\*\*\*\*

unchanged portion omitted

```

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

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

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

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

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

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

196 extern fxdpent_t *fx_getdpttbl(void);

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

```

```

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

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

unchanged portion omitted

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

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

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

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

```

```

1231 /*
1232  * Return Values:
1233  *
1234  *     -1 if the thread is loaded or is not eligible to be swapped in.
1235  *
1236  * FX and RT threads are designed so that they don't swapout; however,
1237  * it is possible that while the thread is swapped out and in another class, it
1238  * can be changed to FX or RT. Since these threads should be swapped in
1239  * as soon as they're runnable, rt_swapin returns SHRT_MAX, and fx_swapin
1240  * returns SHRT_MAX - 1, so that it gives deference to any swapped out
1241  * RT threads.
1242  */
1243 /* ARGSUSED */
1244 static pri_t
1245 fx_swapin(kthread_t *t, int flags)
1246 {
1247     pri_t      tpri = -1;
1248
1249     ASSERT(THREAD_LOCK_HELD(t));
1250
1251     if (t->t_state == TS_RUN && (t->t_schedflag & TS_LOAD) == 0) {
1252         tpri = (pri_t)SHRT_MAX - 1;
1253     }
1254
1255     return (tpri);
1256 }
1257
1258 /*
1259  * Return Values
1260  *     -1 if the thread isn't loaded or is not eligible to be swapped out.
1261  */
1262 /* ARGSUSED */
1263 static pri_t
1264 fx_swapout(kthread_t *t, int flags)
1265 {
1266     ASSERT(THREAD_LOCK_HELD(t));
1267
1268     return (-1);
1269 }
1270 }
  
```

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

```

1342 /*
1343  * Processes waking up go to the back of their queue.
1344  */
1345 static void
1346 fx_wakeup(kthread_t *t)
1347 {
1348     fxproc_t      *fxpp = (fxproc_t *) (t->t_cldata);
1349
1350     ASSERT(THREAD_LOCK_HELD(t));
1351
1352     t->t_stime = ddi_get_lbolt();           /* time stamp for the swapper */
1353     if (FX_HAS_CB(fxpp)) {
1354         clock_t new_quantum = (clock_t)fxpp->fx_pquantum;
1355         pri_t newpri = fxpp->fx_pri;
1356         FX_CB_WAKEUP(FX_CALLBACK(fxpp), fxpp->fx_cookie,
1357             &new_quantum, &newpri);
1358         FX_ADJUST_QUANTUM(new_quantum);
1359         if ((int)new_quantum != fxpp->fx_pquantum) {
1360             fxpp->fx_pquantum = (int)new_quantum;
1361             fxpp->fx_timeleft = fxpp->fx_pquantum;
1362         }
1363     }
1364 }
1365 }
  
```

```

1363         FX_ADJUST_PRI(newpri);
1364         if (newpri != fxpp->fx_pri) {
1365             fxpp->fx_pri = newpri;
1366             THREAD_CHANGE_PRI(t, fx_dptbl[fxpp->fx_pri].fx_globpri);
1367         }
1368     }
1369
1370     fxpp->fx_flags &= ~FXBACKQ;
1371
1372     if (t->t_disp_time != ddi_get_lbolt())
1373         setbackdq(t);
1374     else
1375         setfrontdq(t);
1376 }
  
```

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

new/usr/src/uts/common/disp/rt.c

1

```
*****
25930 Thu Feb 18 08:40:38 2016
new/usr/src/uts/common/disp/rt.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

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

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

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

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

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

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

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

146 extern rtdpent_t *rt_getdpttbl(void);

148 static struct classfuncs rt_classfuncs = {
149 /* class ops */
```

new/usr/src/uts/common/disp/rt.c

2

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

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

902 ASSERT(THREAD_LOCK_HELD(t));

909 /*
910 * If the state is user I allow swapping because I know I won't
911 * be holding any locks.
912 */
913 if ((lwp = curthread->t_lwp) != NULL && lwp->lwp_state == LWP_USER)
914 t->t_schedflag &= ~TS_DONT_SWAP;
915 if ((rtpp->rt_flags & RTBACKQ) != 0) {
916 rtpp->rt_timeleft = rtpp->rt_pquantum;
917 rtpp->rt_flags &= ~RTBACKQ;
918 setbackdq(t);
919 } else
920 setfrontdq(t);

911 }
_____unchanged_portion_omitted_____
```

```
923 static void
924 rt_setrun(kthread_t *)
925 {
926     rtproc_t *rtpp = (rtproc_t *) (t->t_cldata);
927
928     ASSERT(THREAD_LOCK_HELD(t));
929
930     rtp->rt_timeleft = rtp->rt_pquantum;
931     rtp->rt_flags &= ~RTBACKQ;
932     setbackdq(t);
933 }
934
935 /*
936 * Returns the priority of the thread, -1 if the thread is loaded or ineligible
937 * for swapin.
938 *
939 * FX and RT threads are designed so that they don't swapout; however, it
940 * is possible that while the thread is swapped out and in another class, it
941 * can be changed to FX or RT. Since these threads should be swapped in as
942 * soon as they're runnable, rt_swapin returns SHRT_MAX, and fx_swapin
943 * returns SHRT_MAX - 1, so that it gives deference to any swapped out RT
944 * threads.
945 */
946 /* ARGSUSED */
947 static pri_t
948 rt_swapin(kthread_t *t, int flags)
949 {
950     pri_t     tpri = -1;
951
952     ASSERT(THREAD_LOCK_HELD(t));
953
954     if (t->t_state == TS_RUN && (t->t_schedflag & TS_LOAD) == 0) {
955         tpri = (pri_t)SHRT_MAX;
956     }
957
958     return (tpri);
959 }
960
961 /*
962 * Return an effective priority for swapout.
963 */
964 /* ARGSUSED */
965 static pri_t
966 rt_swapout(kthread_t *t, int flags)
967 {
968     ASSERT(THREAD_LOCK_HELD(t));
969
970     return (-1);
971 }
972
973 _____unchanged_portion_omitted_____
```

```

*****
4804 Thu Feb 18 08:40:38 2016
new/usr/src/uts/common/disp/sysclass.c
6583 remove whole-process swapping
*****
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 2008 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25  */
26 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
27 /*      All Rights Reserved      */
28
29 #pragma ident      "%Z%%M% %I%      %E% SMI"      /* from SVr4.0 1.12 */
30
31 #include <sys/types.h>
32 #include <sys/param.h>
33 #include <sys/sysmacros.h>
34 #include <sys/signal.h>
35 #include <sys/pcb.h>
36 #include <sys/user.h>
37 #include <sys/systm.h>
38 #include <sys/sysinfo.h>
39 #include <sys/var.h>
40 #include <sys/errno.h>
41 #include <sys/cmn_err.h>
42 #include <sys/proc.h>
43 #include <sys/debug.h>
44 #include <sys/inline.h>
45 #include <sys/disp.h>
46 #include <sys/class.h>
47 #include <sys/kmem.h>
48 #include <sys/cpuvar.h>
49 #include <sys/priocntl.h>
50
51 /*
52  * Class specific code for the sys class. There are no
53  * class specific data structures associated with
54  * the sys class and the scheduling policy is trivially
55  * simple. There is no time slicing.
56  */
57
58 pri_t      sys_init(id_t, int, classfuncs_t **);
59 static int  sys_getclpri(pcpri_t *);
60 static int  sys_fork(kthread_t *, kthread_t *, void *);

```

```

60 static int  sys_enterclass(kthread_t *, id_t, void *, cred_t *, void *);
61 static int  sys_canexit(kthread_t *, cred_t *);
62 static int  sys_nosys();
63 static int  sys_donice(kthread_t *, cred_t *, int, int *);
64 static int  sys_doprio(kthread_t *, cred_t *, int, int *);
65 static void sys_forkret(kthread_t *, kthread_t *);
66 static void sys_nullsys();
67 static int  sys_swappri(kthread_t *, int);
68 static int  sys_alloc(void **, int);
69
70 struct classfuncs sys_classfuncs = {
71     /* messages to class manager */
72     {
73         sys_nosys,      /* admin */
74         sys_nosys,      /* getclinfo */
75         sys_nosys,      /* parmsin */
76         sys_nosys,      /* parmsout */
77         sys_nosys,      /* vaparmsin */
78         sys_nosys,      /* vaparmsout */
79         sys_getclpri,   /* getclpri */
80         sys_alloc,      /* free */
81         sys_nullsys,
82     },
83     /* operations on threads */
84     {
85         sys_enterclass, /* enterclass */
86         sys_nullsys,    /* exitclass */
87         sys_canexit,
88         sys_fork,
89         sys_forkret,    /* forkret */
90         sys_nullsys,    /* parmsget */
91         sys_nosys,      /* parmsset */
92         sys_nullsys,    /* stop */
93         sys_nullsys,    /* exit */
94         sys_nullsys,    /* active */
95         sys_nullsys,    /* inactive */
96         sys_swappri,    /* swapin */
97         sys_swappri,    /* swapout */
98         sys_nullsys,    /* trapret */
99         setfrontdq,     /* preempt */
100        setbackdq,      /* setrun */
101        sys_nullsys,    /* sleep */
102        sys_nullsys,    /* tick */
103        setbackdq,     /* wakeup */
104        sys_donice,
105        (pri_t (*)())sys_nosys, /* globpri */
106        sys_nullsys,    /* set_process_group */
107        sys_nullsys,    /* yield */
108        sys_doprio,
109    }
110 };
111
112 /*
113  * unchanged portion omitted
114  */
115
116 /* ARGSUSED */
117 static void
118 sys_forkret(t, ct)
119     kthread_t *t;
120     kthread_t *ct;
121 {
122     register proc_t *pp = ttoproc(t);
123     register proc_t *cp = ttoproc(ct);
124
125     ASSERT(t == curthread);
126     ASSERT(MUTEX_HELD(&pidlock));

```

```
178     /*
179     * Grab the child's p_lock before dropping pidlock to ensure
180     * the process does not disappear before we set it running.
181     */
182     mutex_enter(&cp->p_lock);
183     mutex_exit(&pidlock);
184     continuelwps(cp);
185     mutex_exit(&cp->p_lock);
186
187     mutex_enter(&pp->p_lock);
188     continuelwps(pp);
189     mutex_exit(&pp->p_lock);
190 }
191
192
193
194
195
196
197 /* ARGSUSED */
198 static pri_t
199 sys_swappri(t, flags)
200     kthread_t    *t;
201     int          flags;
202 {
203     return (-1);
204 }
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

unchanged portion omitted



new/usr/src/uts/common/disp/sysdc.c

1

```
*****
37694 Thu Feb 18 08:40:39 2016
new/usr/src/uts/common/disp/sysdc.c
6583 remove whole-process swapping
*****
_unchanged_portion_omitted_
```

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

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

unchanged\_portion\_omitted\_

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

```
1171 static struct classfuncs sysdc_classfuncs = {
1172     /* messages to class manager */
1173     {
1174         sysdc_enosys, /* admin */
1175         sysdc_getclinfo,
1176         sysdc_enosys, /* parmsin */
1177         sysdc_enosys, /* parmsout */
1178         sysdc_enosys, /* vaparmsin */
1179         sysdc_enosys, /* vaparmsout */
1180         sysdc_getclpri,
1181         sysdc_alloc,
1182         sysdc_free,
1183     },
1184     /* operations on threads */
1185     {
1186         sysdc_enterclass,
1187         sysdc_exitclass,
1188         sysdc_canexit,
1189         sysdc_fork,
1190         sysdc_forkret,
1191         sysdc_nullsys, /* parmsget */
1192         sysdc_enosys, /* parmsset */
1193         sysdc_nullsys, /* stop */
1194         sysdc_exit,
1195         sysdc_nullsys, /* active */
1196         sysdc_nullsys, /* inactive */
1205         sysdc_no_swap, /* swapin */
1206         sysdc_no_swap, /* swapout */
1197         sysdc_nullsys, /* trapret */
1198         sysdc_preempt,
1199         sysdc_setrun,
1200         sysdc_sleep,
1201         sysdc_tick,
1202         sysdc_wakeup,
1203         sysdc_einval, /* donice */

```

new/usr/src/uts/common/disp/sysdc.c

2

```
1204         sysdc_globpri,
1205         sysdc_nullsys, /* set_process_group */
1206         sysdc_nullsys, /* yield */
1207         sysdc_einval, /* doprio */
1208     }
1209 };
_unchanged_portion_omitted_
```

```

*****
53361 Thu Feb 18 08:40:39 2016
new/usr/src/uts/common/disp/thread.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

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

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

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

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

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

363         ASSERT(stk != NULL);

365         /*
366          * The machine-dependent mutex code may require that
367          * thread pointers (since they may be used for mutex owner
368          * fields) have certain alignment requirements.
369          * PTR24_ALIGN is the size of the alignment quanta.
370          * XXX - assumes stack grows toward low addresses.
371          */
372         if (stksize <= sizeof(kthread_t) + PTR24_ALIGN)

```

```

373         cmn_err(CE_PANIC, "thread_create: proposed stack size"
374             " too small to hold thread.");
375 #ifdef STACK_GROWTH_DOWN
376     stksize -= SA(sizeof(kthread_t) + PTR24_ALIGN - 1);
377     stksize &= -PTR24_ALIGN; /* make thread aligned */
378     t = (kthread_t*)(stk + stksize);
379     bzero(t, sizeof(kthread_t));
380     if (audit_active)
381         audit_thread_create(t);
382     t->t_stk = stk + stksize;
383     t->t_stkbase = stk;
384 #else /* stack grows to larger addresses */
385     stksize -= SA(sizeof(kthread_t));
386     t = (kthread_t*)(stk);
387     bzero(t, sizeof(kthread_t));
388     t->t_stk = stk + sizeof(kthread_t);
389     t->t_stkbase = stk + stksize + sizeof(kthread_t);
390 #endif /* STACK_GROWTH_DOWN */
391     t->t_flag |= T_TALLOCSTK;
392     t->t_swap = stk;
393 } else {
394     t = kmem_cache_alloc(thread_cache, KM_SLEEP);
395     bzero(t, sizeof(kthread_t));
396     ASSERT(((uintptr_t)t & (PTR24_ALIGN - 1)) == 0);
397     if (audit_active)
398         audit_thread_create(t);
399     /*
400      * Initialize t_stk to the kernel stack pointer to use
401      * upon entry to the kernel
402      */
403 #ifdef STACK_GROWTH_DOWN
404     t->t_stk = stk + stksize;
405     t->t_stkbase = stk;
406 #else
407     t->t_stk = stk; /* 3b2-like */
408     t->t_stkbase = stk + stksize;
409 #endif /* STACK_GROWTH_DOWN */
410 }

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

416     t->t_ts = ts;

418     /*
419     * p_cred could be NULL if it thread_create is called before cred_init
420     * is called in main.
421     */
422     mutex_enter(&pp->p_crlock);
423     if (pp->p_cred)
424         crhold(t->t_cred = pp->p_cred);
425     mutex_exit(&pp->p_crlock);
426     t->t_start = gethrestime_sec();
427     t->t_startpc = proc;
428     t->t_procp = pp;
429     t->t_clfuncs = &sys_classfuncs.thread;
430     t->t_cid = syscid;
431     t->t_pri = pri;
432     t->t_schedflag = 0;
433     t->t_stime = ddi_get_lbolt();
434     t->t_schedflag = TS_LOAD | TS_DONT_SWAP;
435     t->t_bind_cpu = PBIND_NONE;
436     t->t_bindflag = (uchar_t)default_binding_mode;
437     t->t_bind_pset = PS_NONE;
438     t->t_plockp = &pp->p_lock;

```

```

437     t->t_copyops = NULL;
438     t->t_taskq = NULL;
439     t->t_anttime = 0;
440     t->t_hatdepth = 0;

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

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

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

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

467     lgrp_affinity_init(&t->t_lgrp_affinity);

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

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

487     /*
488     * Threads should never have a NULL t_cpu pointer so assign it
489     * here. If the thread is being created with state TS_RUN a
490     * better CPU may be chosen when it is placed on the run queue.
491     *
492     * We need to keep kernel preemption disabled when setting all
493     * three fields to keep them in sync. Also, always create in
494     * the default partition since that's where kernel threads go
495     * (if this isn't a kernel thread, t_cpupart will be changed
496     * in lwp_create before setting the thread runnable).
497     */
498     t->t_cpupart = &cp_default;

500     /*
501     * For now, affiliate this thread with the root lgroup.
502     * Since the kernel does not (presently) allocate its memory

```

```

503     * in a locality aware fashion, the root is an appropriate home.
504     * If this thread is later associated with an lwp, it will have
505     * it's lgroup re-assigned at that time.
506     */
507     lgrp_move_thread(t, &cp_default.cp_lgrploads[LGRP_ROOTID], 1);

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

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

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

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

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

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

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

```

unchanged portion omitted

```

*****
57791 Thu Feb 18 08:40:39 2016
new/usr/src/uts/common/disp/ts.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

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

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

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

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

236 static gid_t     IA_gid = 0;

238 static struct classfuncs ts_classfuncs = {
239     /* class functions */

```

```

240     ts_admin,
241     ts_getclinfo,
242     ts_parmsin,
243     ts_parmsout,
244     ts_vaparmsin,
245     ts_vaparmsout,
246     ts_getclpri,
247     ts_alloc,
248     ts_free,

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

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

293     /* thread functions */
294     ts_enterclass,
295     ts_exitclass,
296     ts_canexit,
297     ts_fork,
298     ts_forkret,
299     ia_parmsget,
300     ia_parmsset,
301     ts_nullsys, /* stop */
302     ts_exit,
303     ts_nullsys, /* active */

```

```

304     ts_nullsys,      /* inactive */
309     ts_swapin,
310     ts_swapout,
305     ts_trapret,
306     ts_preempt,
307     ts_setrun,
308     ts_sleep,
309     ts_tick,
310     ts_wakeup,
311     ts_donice,
312     ts_globpri,
313     ia_set_process_group,
314     ts_yield,
315     ts_doprio,
316 };
    _____
    unchanged_portion_omitted

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

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

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

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

1400     /*
1401     * If thread got preempted in the user-land then we know
1402     * it isn't holding any locks. Mark it as swappable.
1403     */
1404     ASSERT(t->t_schedflag & TS_DONT_SWAP);
1405     if (lwp != NULL && lwp->lwp_state == LWP_USER)
1406         t->t_schedflag &= ~TS_DONT_SWAP;

1414     /*

```

```

1401     * Check to see if we're doing "preemption control" here.  If
1402     * we are, and if the user has requested that this thread not
1403     * be preempted, and if preemptions haven't been put off for
1404     * too long, let the preemption happen here but try to make
1405     * sure the thread is rescheduled as soon as possible.  We do
1406     * this by putting it on the front of the highest priority run
1407     * queue in the TS class.  If the preemption has been put off
1408     * for too long, clear the "nopreempt" bit and let the thread
1409     * be preempted.
1410     */
1411     if (t->t_schedctl && schedctl_get_nopreempt(t)) {
1412         if (tspp->ts_timeleft > -SC_MAX_TICKS) {
1413             DTRACE_SCHED1(schedctl__nopreempt, kthread_t *, t);
1414             if (!(tspp->ts_flags & TSKPRI)) {
1415                 /*
1416                 * If not already remembered, remember current
1417                 * priority for restoration in ts_yield().
1418                 */
1419                 if (!(tspp->ts_flags & TSRESTORE)) {
1420                     tspp->ts_scpr = t->t_pri;
1421                     tspp->ts_flags |= TSRESTORE;
1422                 }
1423                 THREAD_CHANGE_PRI(t, ts_maxumdpr);
1438                 t->t_schedflag |= TS_DONT_SWAP;
1439             }
1440             schedctl_set_yield(t, 1);
1441             setfrontdq(t);
1442             goto done;
1443         } else {
1444             if (tspp->ts_flags & TSRESTORE) {
1445                 THREAD_CHANGE_PRI(t, tspp->ts_scpr);
1446                 tspp->ts_flags &= ~TSRESTORE;
1447             }
1448             schedctl_set_nopreempt(t, 0);
1449             DTRACE_SCHED1(schedctl__preempt, kthread_t *, t);
1450             TNF_PROBE_2(schedctl__preempt, "schedctl TS ts_preempt",
1451                 /* CSTYLED */, tnf_pid, pid, ttoproc(t)->p_pid,
1452                 tnf_lwpid, lwpid, t->t_tid);
1453             /*
1454             * Fall through and be preempted below.
1455             */
1456         }
1457     }

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

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

1496 /*
1497  * Prepare thread for sleep. We reset the thread priority so it will
1498  * run at the kernel priority level when it wakes up.

```

```

1499 */
1500 static void
1501 ts_sleep(kthread_t *t)
1502 {
1503     tsproc_t      *tspp = (tsproc_t *) (t->t_cldata);
1504     int            flags;
1505     pri_t          old_pri = t->t_pri;

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

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

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

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

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

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

1571 /*
1572 * Return Values:
1573 *
1574 * -1 if the thread is loaded or is not eligible to be swapped in.
1575 *
1576 * effective priority of the specified thread based on swapout time
1577 * and size of process (epri >= 0 , epri <= SHRT_MAX).
1578 */
1579 /* ARGSUSED */

```

```

1580 static pri_t
1581 ts_swapin(kthread_t *t, int flags)
1582 {
1583     tsproc_t      *tspp = (tsproc_t *) (t->t_cldata);
1584     long           epri = -1;
1585     proc_t        *pp = ttoproc(t);

1587     ASSERT(THREAD_LOCK_HELD(t));

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

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

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

1641 static pri_t
1642 ts_swapout(kthread_t *t, int flags)
1643 {
1644     tsproc_t      *tspp = (tsproc_t *) (t->t_cldata);
1645     long           epri = -1;

```

```

1646     proc_t      *pp = ttoproc(t);
1647     time_t      swapin_time;

1649     ASSERT(THREAD_LOCK_HELD(t));

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

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

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

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

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

1694     return ((pri_t)epri);
1695 }

1554 /*
1555 * Check for time slice expiration.  If time slice has expired
1556 * move thread to priority specified in tsdptbl for time slice expiration
1557 * and set runrun to cause preemption.
1558 */
1559 static void
1560 ts_tick(kthread_t *t)
1561 {
1562     tsproc_t *tspp = (tsproc_t *) (t->t_cldata);
1563     klwp_t *lwp;
1564     boolean_t call_cpu_surrender = B_FALSE;
1565     pri_t oldpri = t->t_pri;

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

```

```

1568     thread_lock(t);

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

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

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

1598                 TNF_PROBE_2(schedctl_failsafe,
1599                     "schedctl TS ts_tick", /* CSTYLED */,
1600                     tnf_pid, pid, ttoproc(t)->p_pid,
1601                     tnf_lwpid, lwpid, t->t_tid);
1602             }
1603             tspp->ts_flags &= ~TSRESTORE;
1604             tspp->ts_cpupri = ts_dptbl[tspp->ts_cpupri].ts_tqexp;
1605             TS_NEWUMDPRI(tspp);
1606             tspp->ts_dispwait = 0;
1607             new_pri = ts_dptbl[tspp->ts_umdpr].ts_globpri;
1608             ASSERT(new_pri >= 0 && new_pri <= ts_maxglobpri);
1609             /*
1610             * When the priority of a thread is changed,
1611             * it may be necessary to adjust its position
1612             * on a sleep queue or dispatch queue.
1613             * The function thread_change_pri accomplishes
1614             * this.
1615             */
1616             if (thread_change_pri(t, new_pri, 0) {
1617                 if ((t->t_schedflag & TS_LOAD) &&
1618                     (lwp = t->t_lwp) &&
1619                     lwp->lwp_state == LWP_USER)
1620                     t->t_schedflag &= ~TS_DONT_SWAP;
1621                 tspp->ts_timeleft =
1622                     ts_dptbl[tspp->ts_cpupri].ts_quantum;
1623             } else {
1624                 call_cpu_surrender = B_TRUE;
1625             }
1626             TRACE_2(TR_FAC_DISP, TR_TICK,
1627                 "tick:tid %p old pri %d", t, oldpri);
1628         } else if (t->t_state == TS_ONPROC &&
1629             t->t_pri < t->t_disp_queue->disp_maxrunpri) {
1630             call_cpu_surrender = B_TRUE;
1631         }
1632     }

```

```

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

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

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

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

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

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

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

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

1838     /*
1839     * Swapout lwp if the swapper is waiting for this thread to
1840     * reach a safe point.
1841     */
1842     if ((t->t_schedflag & TS_SWAPENQ) && !(tspp->ts_flags & TSIASET)) {
1843         thread_unlock(t);

```

```

1844         swapout_lwp(ttolwp(t));
1845         thread_lock(t);
1846     }

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

    unchanged_portion_omitted

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

1825     ASSERT(THREAD_LOCK_HELD(t));

1885     t->t_stime = ddi_get_lbolt();    /* time stamp for the swapper */

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

1854         tspp->ts_flags &= ~TSBACKQ;

1856         if (tspp->ts_flags & TSIA) {
1857             if (tspp->ts_flags & TSIASET)
1858                 setfrontdq(t);
1859             else
1860                 setbackdq(t);
1861         } else {
1862             if (t->t_disp_time != ddi_get_lbolt())
1863                 setbackdq(t);
1864             else
1865                 setfrontdq(t);
1866         }

```



new/usr/src/uts/common/disp/ts.c

11

```
1867     }  
1868 }  
_____unchanged_portion_omitted_
```

```

*****
67734 Thu Feb 18 08:40:39 2016
new/usr/src/uts/common/fs/nfs/nfs_srv.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

1151 static struct rfs_async_write_list *rfs_async_write_head = NULL;
1152 static kmutex_t rfs_async_write_lock;
1153 static int rfs_write_async = 1; /* enables write clustering if == 1 */

1155 #define MAXCLIOVECS 42
1156 #define RFSWRITE_INITVAL (enum nfsstat) -1

1158 #ifdef DEBUG
1159 static int rfs_write_hits = 0;
1160 static int rfs_write_misses = 0;
1161 #endif

1163 /*
1164  * Write data to file.
1165  * Returns attributes of a file after writing some data to it.
1166  */
1167 void
1168 rfs_write(struct nfswriteargs *wa, struct nfsattrstat *ns,
1169          struct exportinfo *exi, struct svc_req *req, cred_t *cr, bool_t ro)
1170 {
1171     int error;
1172     vnode_t *vp;
1173     rlim64_t rlimit;
1174     struct vattr va;
1175     struct uio uio;
1176     struct rfs_async_write_list *lp;
1177     struct rfs_async_write_list *nlp;
1178     struct rfs_async_write *rp;
1179     struct rfs_async_write *nrp;
1180     struct rfs_async_write *trp;
1181     struct rfs_async_write *lrp;
1182     int data_written;
1183     int iovcnt;
1184     mblk_t *m;
1185     struct iovec *iovp;
1186     struct iovec *niovp;
1187     struct iovec iov[MAXCLIOVECS];
1188     int count;
1189     int rcount;
1190     uint_t off;
1191     uint_t len;
1192     struct rfs_async_write nrpsp;
1193     struct rfs_async_write_list nlpsp;
1194     ushort_t t_flag;
1195     cred_t *savecred;
1196     int in_crit = 0;
1197     caller_context_t ct;

1199     if (!rfs_write_async) {
1200         rfs_write_sync(wa, ns, exi, req, cr, ro);
1201         return;
1202     }

1204     /*
1205     * Initialize status to RFSWRITE_INITVAL instead of 0, since value of 0
1206     * is considered an OK.
1207     */
1208     ns->ns_status = RFSWRITE_INITVAL;

```

```

1210     nrp = &nrpsp;
1211     nrp->wa = wa;
1212     nrp->ns = ns;
1213     nrp->req = req;
1214     nrp->cr = cr;
1215     nrp->ro = ro;
1216     nrp->thread = curthread;

1218     ASSERT(curthread->t_schedflag & TS_DONT_SWAP);

1218     /*
1219     * Look to see if there is already a cluster started
1220     * for this file.
1221     */
1222     mutex_enter(&rfs_async_write_lock);
1223     for (lp = rfs_async_write_head; lp != NULL; lp = lp->next) {
1224         if (bcmp(&wa->wa_fhandle, lp->fhp,
1225                sizeof (fhandle_t)) == 0)
1226             break;
1227     }

1229     /*
1230     * If lp is non-NULL, then there is already a cluster
1231     * started. We need to place ourselves in the cluster
1232     * list in the right place as determined by starting
1233     * offset. Conflicts with non-blocking mandatory locked
1234     * regions will be checked when the cluster is processed.
1235     */
1236     if (lp != NULL) {
1237         rp = lp->list;
1238         trp = NULL;
1239         while (rp != NULL && rp->wa->wa_offset < wa->wa_offset) {
1240             trp = rp;
1241             rp = rp->list;
1242         }
1243         nrp->list = rp;
1244         if (trp == NULL)
1245             lp->list = nrp;
1246         else
1247             trp->list = nrp;
1248         while (nrp->ns->ns_status == RFSWRITE_INITVAL)
1249             cv_wait(&lp->cv, &rfs_async_write_lock);
1250         mutex_exit(&rfs_async_write_lock);

1252         return;
1253     }

1255     /*
1256     * No cluster started yet, start one and add ourselves
1257     * to the list of clusters.
1258     */
1259     nrp->list = NULL;

1261     nlp = &nlpsp;
1262     nlp->fhp = &wa->wa_fhandle;
1263     cv_init(&nlp->cv, NULL, CV_DEFAULT, NULL);
1264     nlp->list = nrp;
1265     nlp->next = NULL;

1267     if (rfs_async_write_head == NULL) {
1268         rfs_async_write_head = nlp;
1269     } else {
1270         lp = rfs_async_write_head;
1271         while (lp->next != NULL)
1272             lp = lp->next;
1273         lp->next = nlp;

```

```

1274     }
1275     mutex_exit(&rfs_async_write_lock);

1277     /*
1278     * Convert the file handle common to all of the requests
1279     * in this cluster to a vnode.
1280     */
1281     vp = nfs_fhtovp(&wa->wa_fhandle, exi);
1282     if (vp == NULL) {
1283         mutex_enter(&rfs_async_write_lock);
1284         if (rfs_async_write_head == nlp)
1285             rfs_async_write_head = nlp->next;
1286         else {
1287             lp = rfs_async_write_head;
1288             while (lp->next != nlp)
1289                 lp = lp->next;
1290             lp->next = nlp->next;
1291         }
1292         t_flag = curthread->t_flag & T_WOULDBLOCK;
1293         for (rp = nlp->list; rp != NULL; rp = rp->list) {
1294             rp->ns->ns_status = NFSERR_STALE;
1295             rp->thread->t_flag |= t_flag;
1296         }
1297         cv_broadcast(&nlp->cv);
1298         mutex_exit(&rfs_async_write_lock);

1300     }
1301     return;

1303     /*
1304     * Can only write regular files. Attempts to write any
1305     * other file types fail with EISDIR.
1306     */
1307     if (vp->v_type != VREG) {
1308         VN_RELE(vp);
1309         mutex_enter(&rfs_async_write_lock);
1310         if (rfs_async_write_head == nlp)
1311             rfs_async_write_head = nlp->next;
1312         else {
1313             lp = rfs_async_write_head;
1314             while (lp->next != nlp)
1315                 lp = lp->next;
1316             lp->next = nlp->next;
1317         }
1318         t_flag = curthread->t_flag & T_WOULDBLOCK;
1319         for (rp = nlp->list; rp != NULL; rp = rp->list) {
1320             rp->ns->ns_status = NFSERR_ISDIR;
1321             rp->thread->t_flag |= t_flag;
1322         }
1323         cv_broadcast(&nlp->cv);
1324         mutex_exit(&rfs_async_write_lock);

1326     }
1327     return;

1329     /*
1330     * Enter the critical region before calling VOP_RWLOCK, to avoid a
1331     * deadlock with ufs.
1332     */
1333     if (nbl_need_check(vp)) {
1334         nbl_start_crit(vp, RW_READER);
1335         in_crit = 1;
1336     }

1338     ct.cc_sysid = 0;
1339     ct.cc_pid = 0;

```

```

1340     ct.cc_caller_id = nfs2_srv_caller_id;
1341     ct.cc_flags = CC_DONTBLOCK;

1343     /*
1344     * Lock the file for writing. This operation provides
1345     * the delay which allows clusters to grow.
1346     */
1347     error = VOP_RWLOCK(vp, V_WRITELOCK_TRUE, &ct);

1349     /* check if a monitor detected a delegation conflict */
1350     if (error == EAGAIN && (ct.cc_flags & CC_WOULDBLOCK)) {
1351         if (in_crit)
1352             nbl_end_crit(vp);
1353         VN_RELE(vp);
1354         /* mark as wouldblock so response is dropped */
1355         curthread->t_flag |= T_WOULDBLOCK;
1356         mutex_enter(&rfs_async_write_lock);
1357         if (rfs_async_write_head == nlp)
1358             rfs_async_write_head = nlp->next;
1359         else {
1360             lp = rfs_async_write_head;
1361             while (lp->next != nlp)
1362                 lp = lp->next;
1363             lp->next = nlp->next;
1364         }
1365         for (rp = nlp->list; rp != NULL; rp = rp->list) {
1366             if (rp->ns->ns_status == RFSWRITE_INITVAL) {
1367                 rp->ns->ns_status = puterrno(error);
1368                 rp->thread->t_flag |= T_WOULDBLOCK;
1369             }
1370         }
1371         cv_broadcast(&nlp->cv);
1372         mutex_exit(&rfs_async_write_lock);

1374     }
1375     return;

1377     /*
1378     * Disconnect this cluster from the list of clusters.
1379     * The cluster that is being dealt with must be fixed
1380     * in size after this point, so there is no reason
1381     * to leave it on the list so that new requests can
1382     * find it.
1383     *
1384     * The algorithm is that the first write request will
1385     * create a cluster, convert the file handle to a
1386     * vnode pointer, and then lock the file for writing.
1387     * This request is not likely to be clustered with
1388     * any others. However, the next request will create
1389     * a new cluster and be blocked in VOP_RWLOCK while
1390     * the first request is being processed. This delay
1391     * will allow more requests to be clustered in this
1392     * second cluster.
1393     */
1394     mutex_enter(&rfs_async_write_lock);
1395     if (rfs_async_write_head == nlp)
1396         rfs_async_write_head = nlp->next;
1397     else {
1398         lp = rfs_async_write_head;
1399         while (lp->next != nlp)
1400             lp = lp->next;
1401         lp->next = nlp->next;
1402     }
1403     mutex_exit(&rfs_async_write_lock);

1405     /*

```

```

1406     * Step through the list of requests in this cluster.
1407     * We need to check permissions to make sure that all
1408     * of the requests have sufficient permission to write
1409     * the file. A cluster can be composed of requests
1410     * from different clients and different users on each
1411     * client.
1412     *
1413     * As a side effect, we also calculate the size of the
1414     * byte range that this cluster encompasses.
1415     */
1416     rp = nlp->list;
1417     off = rp->wa->wa_offset;
1418     len = (uint_t)0;
1419     do {
1420         if (rdonly(rp->ro, vp)) {
1421             rp->ns->ns_status = NFSERR_ROFS;
1422             t_flag = curthread->t_flag & T_WOULDBLOCK;
1423             rp->thread->t_flag |= t_flag;
1424             continue;
1425         }
1427         va.va_mask = AT_UID|AT_MODE;
1429         error = VOP_GETATTR(vp, &va, 0, rp->cr, &ct);
1431         if (!error) {
1432             if (crgetuid(rp->cr) != va.va_uid) {
1433                 /*
1434                  * This is a kludge to allow writes of files
1435                  * created with read only permission. The
1436                  * owner of the file is always allowed to
1437                  * write it.
1438                  */
1439                 error = VOP_ACCESS(vp, VWRITE, 0, rp->cr, &ct);
1441                 if (!error && MANDLOCK(vp, va.va_mode))
1442                     error = EACCES;
1443             }
1445             /*
1446             * Check for a conflict with a nbmand-locked region.
1447             */
1448             if (in_crit && nbl_conflict(vp, NBL_WRITE, rp->wa->wa_offset,
1449                 rp->wa->wa_count, 0, NULL)) {
1450                 error = EACCES;
1451             }
1453             if (error) {
1454                 rp->ns->ns_status = puterrno(error);
1455                 t_flag = curthread->t_flag & T_WOULDBLOCK;
1456                 rp->thread->t_flag |= t_flag;
1457                 continue;
1458             }
1459             if (len < rp->wa->wa_offset + rp->wa->wa_count - off)
1460                 len = rp->wa->wa_offset + rp->wa->wa_count - off;
1461         } while ((rp = rp->list) != NULL);
1463     /*
1464     * Step through the cluster attempting to gather as many
1465     * requests which are contiguous as possible. These
1466     * contiguous requests are handled via one call to VOP_WRITE
1467     * instead of different calls to VOP_WRITE. We also keep
1468     * track of the fact that any data was written.
1469     */
1470     rp = nlp->list;
1471     data_written = 0;

```

```

1472     do {
1473         /*
1474         * Skip any requests which are already marked as having an
1475         * error.
1476         */
1477         if (rp->ns->ns_status != RFSWRITE_INITVAL) {
1478             rp = rp->list;
1479             continue;
1480         }
1482         /*
1483         * Count the number of iovec's which are required
1484         * to handle this set of requests. One iovec is
1485         * needed for each data buffer, whether addressed
1486         * by wa_data or by the b_rptr pointers in the
1487         * mblk chains.
1488         */
1489         iovcnt = 0;
1490         lrp = rp;
1491         for (;;) {
1492             if (lrp->wa->wa_data || lrp->wa->wa_rlist)
1493                 iovcnt++;
1494             else {
1495                 m = lrp->wa->wa_mblk;
1496                 while (m != NULL) {
1497                     iovcnt++;
1498                     m = m->b_cont;
1499                 }
1501             if (lrp->list == NULL ||
1502                 lrp->list->ns->ns_status != RFSWRITE_INITVAL ||
1503                 lrp->wa->wa_offset + lrp->wa->wa_count !=
1504                 lrp->list->wa->wa_offset) {
1505                 lrp = lrp->list;
1506                 break;
1507             }
1508             lrp = lrp->list;
1509         }
1511         if (iovcnt <= MAXCLIOVECS) {
1512             #ifdef DEBUG
1513                 rfs_write_hits++;
1514             #endif
1515             niovp = iov;
1516         } else {
1517             #ifdef DEBUG
1518                 rfs_write_misses++;
1519             #endif
1520             niovp = kmem_alloc(sizeof (*niovp) * iovcnt, KM_SLEEP);
1521         }
1522         /*
1523         * Put together the scatter/gather iovecs.
1524         */
1525         iovp = niovp;
1526         trp = rp;
1527         count = 0;
1528         do {
1529             if (trp->wa->wa_data || trp->wa->wa_rlist) {
1530                 if (trp->wa->wa_rlist) {
1531                     iovp->iov_base =
1532                         (char *)((trp->wa->wa_rlist)->
1533                             u.c_daddr3);
1534                     iovp->iov_len = trp->wa->wa_count;
1535                 } else {
1536                     iovp->iov_base = trp->wa->wa_data;
1537                     iovp->iov_len = trp->wa->wa_count;

```

```

1538     }
1539     iovp++;
1540 } else {
1541     m = trp->wa->wa_mblk;
1542     rcount = trp->wa->wa_count;
1543     while (m != NULL) {
1544         iovp->iov_base = (caddr_t)m->b_rptr;
1545         iovp->iov_len = (m->b_wptr - m->b_rptr);
1546         rcount -= iovp->iov_len;
1547         if (rcount < 0)
1548             iovp->iov_len += rcount;
1549         iovp++;
1550         if (rcount <= 0)
1551             break;
1552         m = m->b_cont;
1553     }
1554 }
1555 count += trp->wa->wa_count;
1556 trp = trp->list;
1557 } while (trp != lrp);

1559 uio.uio_iov = niovp;
1560 uio.uio_iovcnt = iovcnt;
1561 uio.uio_segflg = UIO_SYSSPACE;
1562 uio.uio_extflg = UIO_COPY_DEFAULT;
1563 uio.uio_loffset = (offset_t)rp->wa->wa_offset;
1564 uio.uio_resid = count;
1565 /*
1566  * The limit is checked on the client. We
1567  * should allow any size writes here.
1568  */
1569 uio.uio_llimit = curproc->p_fsz_ctl;
1570 rlimit = uio.uio_llimit - rp->wa->wa_offset;
1571 if (rlimit < (rlim64_t)uio.uio_resid)
1572     uio.uio_resid = (uint_t)rlimit;

1574 /*
1575  * For now we assume no append mode.
1576  */

1578 /*
1579  * We're changing creds because VM may fault
1580  * and we need the cred of the current
1581  * thread to be used if quota * checking is
1582  * enabled.
1583  */
1584 savecred = curthread->t_cred;
1585 curthread->t_cred = cr;
1586 error = VOP_WRITE(vp, &uio, 0, rp->cr, &ct);
1587 curthread->t_cred = savecred;

1589 /* check if a monitor detected a delegation conflict */
1590 if (error == EAGAIN && (ct.cc_flags & CC_WOULDBLOCK))
1591     /* mark as wouldblock so response is dropped */
1592     curthread->t_flag |= T_WOULDBLOCK;

1594 if (niovp != iov)
1595     kmem_free(niovp, sizeof (*niovp) * iovcnt);

1597 if (!error) {
1598     data_written = 1;
1599     /*
1600      * Get attributes again so we send the latest mod
1601      * time to the client side for his cache.
1602      */
1603     va.va_mask = AT_ALL; /* now we want everything */

```

```

1605         error = VOP_GETATTR(vp, &va, 0, rp->cr, &ct);
1607         if (!error)
1608             acl_perm(vp, exi, &va, rp->cr);
1609     }
1611     /*
1612     * Fill in the status responses for each request
1613     * which was just handled. Also, copy the latest
1614     * attributes in to the attribute responses if
1615     * appropriate.
1616     */
1617     t_flag = curthread->t_flag & T_WOULDBLOCK;
1618     do {
1619         rp->thread->t_flag |= t_flag;
1620         /* check for overflows */
1621         if (!error) {
1622             error = vattr_to_natr(&va, &rp->ns->ns_attr);
1623         }
1624         rp->ns->ns_status = puterrno(error);
1625         rp = rp->list;
1626     } while (rp != lrp);
1627 } while (rp != NULL);

1629 /*
1630  * If any data was written at all, then we need to flush
1631  * the data and metadata to stable storage.
1632  */
1633 if (data_written) {
1634     error = VOP_PUTPAGE(vp, (u_offset_t)off, len, 0, cr, &ct);
1636     if (!error) {
1637         error = VOP_FSYNC(vp, FNODSYNC, cr, &ct);
1638     }
1639 }

1641 VOP_RWUNLOCK(vp, V_WRITELOCK_TRUE, &ct);

1643 if (in_crit)
1644     nbl_end_crit(vp);
1645 VN_RELE(vp);

1647 t_flag = curthread->t_flag & T_WOULDBLOCK;
1648 mutex_enter(&rfs_async_write_lock);
1649 for (rp = nlp->list; rp != NULL; rp = rp->list) {
1650     if (rp->ns->ns_status == RFSWRITE_INITVAL) {
1651         rp->ns->ns_status = puterrno(error);
1652         rp->thread->t_flag |= t_flag;
1653     }
1654 }
1655 cv_broadcast(&nlp->cv);
1656 mutex_exit(&rfs_async_write_lock);

1658 }
_____unchanged_portion_omitted_____

```

```

*****
73921 Thu Feb 18 08:40:39 2016
new/usr/src/uts/common/os/clock.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

371 /*
372  * test hook for tod broken detection in tod_validate
373  */
374 int tod_unit_test = 0;
375 time_t tod_test_injector;

377 #define CLOCK_ADJ_HIST_SIZE    4

379 static int    adj_hist_entry;

381 int64_t clock_adj_hist[CLOCK_ADJ_HIST_SIZE];

383 static void calcloadavg(int, uint64_t *);
384 static int genloadavg(struct loadavg_s *);
385 static void loadavg_update();

387 void (*cmm_clock_callout)() = NULL;
388 void (*cpucaps_clock_callout)() = NULL;

390 extern clock_t clock_tick_proc_max;

392 static int64_t deadman_counter = 0;

394 static void
395 clock(void)
396 {
397     kthread_t    *t;
398     uint_t nrunnable;
399     uint_t w_io;
400     cpu_t    *cp;
401     cpupart_t *cpupart;
402     extern void set_freemem();
403     void (*funcp)();
404     int32_t ltemp;
405     int64_t lltemp;
406     int s;
407     int do_lgrp_load;
408     int i;
409     clock_t now = LBOLT_NO_ACCOUNT; /* current tick */

411     if (panicstr)
412         return;

414     /*
415      * Make sure that 'freemem' do not drift too far from the truth
416      */
417     set_freemem();

420     /*
421      * Before the section which is repeated is executed, we do
422      * the time delta processing which occurs every clock tick
423      *
424      * There is additional processing which happens every time
425      * the nanosecond counter rolls over which is described
426      * below - see the section which begins with : if (one_sec)
427      *
428      * This section marks the beginning of the precision-kernel
429      * code fragment.

```

```

430     *
431     * First, compute the phase adjustment. If the low-order bits
432     * (time_phase) of the update overflow, bump the higher order
433     * bits (time_update).
434     */
435     time_phase += time_adj;
436     if (time_phase <= -FINEUSEC) {
437         ltemp = -time_phase / SCALE_PHASE;
438         time_phase += ltemp * SCALE_PHASE;
439         s = hr_clock_lock();
440         timedelta -= ltemp * (NANOSEC/MICROSEC);
441         hr_clock_unlock(s);
442     } else if (time_phase >= FINEUSEC) {
443         ltemp = time_phase / SCALE_PHASE;
444         time_phase -= ltemp * SCALE_PHASE;
445         s = hr_clock_lock();
446         timedelta += ltemp * (NANOSEC/MICROSEC);
447         hr_clock_unlock(s);
448     }

450     /*
451     * End of precision-kernel code fragment which is processed
452     * every timer interrupt.
453     *
454     * Continue with the interrupt processing as scheduled.
455     */
456     /*
457     * Count the number of runnable threads and the number waiting
458     * for some form of I/O to complete -- gets added to
459     * sysinfo.waiting. To know the state of the system, must add
460     * wait counts from all CPUs. Also add up the per-partition
461     * statistics.
462     */
463     w_io = 0;
464     nrunnable = 0;

466     /*
467     * keep track of when to update lgrp/part loads
468     */

470     do_lgrp_load = 0;
471     if (lgrp_ticks++ >= hz / 10) {
472         lgrp_ticks = 0;
473         do_lgrp_load = 1;
474     }

476     if (one_sec) {
477         loadavg_update();
478         deadman_counter++;
479     }

481     /*
482     * First count the threads waiting on kpreempt queues in each
483     * CPU partition.
484     */

486     cpupart = cp_list_head;
487     do {
488         uint_t cpupart_nrunnable = cpupart->cp_kp_queue.disp_nrunnable;

490         cpupart->cp_updates++;
491         nrunnable += cpupart_nrunnable;
492         cpupart->cp_nrunnable_cum += cpupart_nrunnable;
493         if (one_sec) {
494             cpupart->cp_nrunning = 0;
495             cpupart->cp_nrunnable = cpupart_nrunnable;

```

```

496     }
497 } while ((cpupart = cpupart->cp_next) != cp_list_head);

500 /* Now count the per-CPU statistics. */
501 cp = cpu_list;
502 do {
503     uint_t cpu_nrunnable = cp->cpu_disp->disp_nrunnable;

505     nrunnable += cpu_nrunnable;
506     cpupart = cp->cpu_part;
507     cpupart->cp_nrunnable_cum += cpu_nrunnable;
508     if (one_sec) {
509         cpupart->cp_nrunnable += cpu_nrunnable;
510         /*
511          * Update user, system, and idle cpu times.
512          */
513         cpupart->cp_nrunning++;
514         /*
515          * w_io is used to update sysinfo.waiting during
516          * one_second processing below. Only gather w_io
517          * information when we walk the list of cpus if we're
518          * going to perform one_second processing.
519          */
520         w_io += CPU_STATS(cp, sys.iowait);
521     }

523     if (one_sec && (cp->cpu_flags & CPU_EXISTS)) {
524         int i, load, change;
525         hrtime_t intracct, intrused;
526         const hrtime_t maxnsec = 1000000000;
527         const int precision = 100;

529         /*
530          * Estimate interrupt load on this cpu each second.
531          * Computes cpu_intrload as %utilization (0-99).
532          */

534         /* add up interrupt time from all micro states */
535         for (intracct = 0, i = 0; i < NCMSTATES; i++)
536             intracct += cp->cpu_intracct[i];
537         scalehrtime(&intracct);

539         /* compute nsec used in the past second */
540         intrused = intracct - cp->cpu_intrlast;
541         cp->cpu_intrlast = intracct;

543         /* limit the value for safety (and the first pass) */
544         if (intrused >= maxnsec)
545             intrused = maxnsec - 1;

547         /* calculate %time in interrupt */
548         load = (precision * intrused) / maxnsec;
549         ASSERT(load >= 0 && load < precision);
550         change = cp->cpu_intrload - load;

552         /* jump to new max, or decay the old max */
553         if (change < 0)
554             cp->cpu_intrload = load;
555         else if (change > 0)
556             cp->cpu_intrload -= (change + 3) / 4;

558         DTRACE_PROBE3(cpu_intrload,
559                     cpu_t *, cp,
560                     hrtime_t, intracct,
561                     hrtime_t, intrused);

```

```

562     }

564     if (do_lgrp_load &&
565         (cp->cpu_flags & CPU_EXISTS)) {
566         /*
567          * When updating the lgroup's load average,
568          * account for the thread running on the CPU.
569          * If the CPU is the current one, then we need
570          * to account for the underlying thread which
571          * got the clock interrupt not the thread that is
572          * handling the interrupt and calculating the load
573          * average
574          */
575         t = cp->cpu_thread;
576         if (CPU == cp)
577             t = t->t_intr;

579         /*
580          * Account for the load average for this thread if
581          * it isn't the idle thread or it is on the interrupt
582          * stack and not the current CPU handling the clock
583          * interrupt
584          */
585         if ((t && t != cp->cpu_idle_thread) || (CPU != cp &&
586             CPU_ON_INTR(cp))) {
587             if (t->t_lpl == cp->cpu_lpl) {
588                 /* local thread */
589                 cpu_nrunnable++;
590             } else {
591                 /*
592                  * This is a remote thread, charge it
593                  * against its home lgroup. Note that
594                  * we notice that a thread is remote
595                  * only if it's currently executing.
596                  * This is a reasonable approximation,
597                  * since queued remote threads are rare.
598                  * Note also that if we didn't charge
599                  * it to its home lgroup, remote
600                  * execution would often make a system
601                  * appear balanced even though it was
602                  * not, and thread placement/migration
603                  * would often not be done correctly.
604                  */
605                 lgrp_loadavg(t->t_lpl,
606                             LGRP_LOADAVG_IN_THREAD_MAX, 0);
607             }
608         }
609         lgrp_loadavg(cp->cpu_lpl,
610                     cpu_nrunnable * LGRP_LOADAVG_IN_THREAD_MAX, 1);
611     }
612 } while ((cp = cp->cpu_next) != cpu_list);

614 clock_tick_schedule(one_sec);

616 /*
617  * Check for a callout that needs be called from the clock
618  * thread to support the membership protocol in a clustered
619  * system. Copy the function pointer so that we can reset
620  * this to NULL if needed.
621  */
622 if ((funcp = cmm_clock_callout) != NULL)
623     (*funcp)();

625 if ((funcp = cpucaps_clock_callout) != NULL)
626     (*funcp)();

```

```

628  /*
629  * Wakeup the cageout thread waiters once per second.
630  */
631  if (one_sec)
632      kstage_tick();
634  if (one_sec) {
636      int drift, absdrift;
637      timestruc_t tod;
638      int s;
640      /*
641      * Beginning of precision-kernel code fragment executed
642      * every second.
643      *
644      * On rollover of the second the phase adjustment to be
645      * used for the next second is calculated. Also, the
646      * maximum error is increased by the tolerance. If the
647      * PPS frequency discipline code is present, the phase is
648      * increased to compensate for the CPU clock oscillator
649      * frequency error.
650      *
651      * On a 32-bit machine and given parameters in the timex.h
652      * header file, the maximum phase adjustment is +-512 ms
653      * and maximum frequency offset is (a tad less than)
654      * +-512 ppm. On a 64-bit machine, you shouldn't need to ask.
655      */
656      time_maxerror += time_tolerance / SCALE_USEC;
658      /*
659      * Leap second processing. If in leap-insert state at
660      * the end of the day, the system clock is set back one
661      * second; if in leap-delete state, the system clock is
662      * set ahead one second. The microtime() routine or
663      * external clock driver will insure that reported time
664      * is always monotonic. The ugly divides should be
665      * replaced.
666      */
667      switch (time_state) {
669          case TIME_OK:
670              if (time_status & STA_INS)
671                  time_state = TIME_INS;
672              else if (time_status & STA_DEL)
673                  time_state = TIME_DEL;
674              break;
676          case TIME_INS:
677              if (hrestime.tv_sec % 86400 == 0) {
678                  s = hr_clock_lock();
679                  hrestime.tv_sec--;
680                  hr_clock_unlock(s);
681                  time_state = TIME_OOP;
682              }
683              break;
685          case TIME_DEL:
686              if ((hrestime.tv_sec + 1) % 86400 == 0) {
687                  s = hr_clock_lock();
688                  hrestime.tv_sec++;
689                  hr_clock_unlock(s);
690                  time_state = TIME_WAIT;
691              }
692              break;

```

```

694          case TIME_OOP:
695              time_state = TIME_WAIT;
696              break;
698          case TIME_WAIT:
699              if (!(time_status & (STA_INS | STA_DEL)))
700                  time_state = TIME_OK;
701          default:
702              break;
703      }
705      /*
706      * Compute the phase adjustment for the next second. In
707      * PLL mode, the offset is reduced by a fixed factor
708      * times the time constant. In FLL mode the offset is
709      * used directly. In either mode, the maximum phase
710      * adjustment for each second is clamped so as to spread
711      * the adjustment over not more than the number of
712      * seconds between updates.
713      */
714      if (time_offset == 0)
715          time_adj = 0;
716      else if (time_offset < 0) {
717          lltemp = -time_offset;
718          if (!(time_status & STA_FLL)) {
719              if ((1 << time_constant) >= SCALE_KG)
720                  lltemp *= (1 << time_constant) /
721                          SCALE_KG;
722              else
723                  lltemp = (lltemp / SCALE_KG) >>
724                          time_constant;
725          }
726          if (lltemp > (MAXPHASE / MINSEC) * SCALE_UPDATE)
727              lltemp = (MAXPHASE / MINSEC) * SCALE_UPDATE;
728          time_offset += lltemp;
729          time_adj = -(lltemp * SCALE_PHASE) / hz / SCALE_UPDATE;
730      } else {
731          lltemp = time_offset;
732          if (!(time_status & STA_FLL)) {
733              if ((1 << time_constant) >= SCALE_KG)
734                  lltemp *= (1 << time_constant) /
735                          SCALE_KG;
736              else
737                  lltemp = (lltemp / SCALE_KG) >>
738                          time_constant;
739          }
740          if (lltemp > (MAXPHASE / MINSEC) * SCALE_UPDATE)
741              lltemp = (MAXPHASE / MINSEC) * SCALE_UPDATE;
742          time_offset -= lltemp;
743          time_adj = (lltemp * SCALE_PHASE) / hz / SCALE_UPDATE;
744      }
746      /*
747      * Compute the frequency estimate and additional phase
748      * adjustment due to frequency error for the next
749      * second. When the PPS signal is engaged, gnaw on the
750      * watchdog counter and update the frequency computed by
751      * the pll and the PPS signal.
752      */
753      pps_valid++;
754      if (pps_valid == PPS_VALID) {
755          pps_jitter = MAXTIME;
756          pps_stabil = MAXFREQ;
757          time_status &= ~(STA_PPSSIGNAL | STA_PPSJITTER |
758                          STA_PPSWANDER | STA_PPSERROR);
759      }

```



```

760         lltemp = time_freq + pps_freq;
762         if (lltemp)
763             time_adj += (lltemp * SCALE_PHASE) / (SCALE_USEC * hz);
765     /*
766     * End of precision kernel-code fragment
767     *
768     * The section below should be modified if we are planning
769     * to use NTP for synchronization.
770     *
771     * Note: the clock synchronization code now assumes
772     * the following:
773     * - if dosynctodr is 1, then compute the drift between
774     *   the tod chip and software time and adjust one or
775     *   the other depending on the circumstances
776     *
777     * - if dosynctodr is 0, then the tod chip is independent
778     *   of the software clock and should not be adjusted,
779     *   but allowed to free run.  this allows NTP to sync.
780     *   hrestime without any interference from the tod chip.
781     */
783     tod_validate_deferred = B_FALSE;
784     mutex_enter(&tod_lock);
785     tod = tod_get();
786     drift = tod.tv_sec - hrestime.tv_sec;
787     absdrift = (drift >= 0) ? drift : -drift;
788     if (tod_needsync || absdrift > 1) {
789         int s;
790         if (absdrift > 2) {
791             if (!tod_broken && tod_faulted == TOD_NOFAULT) {
792                 s = hr_clock_lock();
793                 hrestime = tod;
794                 membar_enter(); /* hrestime visible */
795                 timedelta = 0;
796                 timechanged++;
797                 tod_needsync = 0;
798                 hr_clock_unlock(s);
799                 callout_hrestime();
801             }
802         } else {
803             if (tod_needsync || !dosynctodr) {
804                 gethrestime(&tod);
805                 tod_set(tod);
806                 s = hr_clock_lock();
807                 if (timedelta == 0)
808                     tod_needsync = 0;
809                 hr_clock_unlock(s);
810             } else {
811                 /*
812                 * If the drift is 2 seconds on the
813                 * money, then the TOD is adjusting
814                 * the clock; record that.
815                 */
816                 clock_adj_hist[adj_hist_entry++ %
817                     CLOCK_ADJ_HIST_SIZE] = now;
818                 s = hr_clock_lock();
819                 timedelta = (int64_t)drift*NANOSEC;
820                 hr_clock_unlock(s);
821             }
822         }
823     }
824     one_sec = 0;
825     time = gethrestime_sec(); /* for crusty old kmem readers */

```

```

826         mutex_exit(&tod_lock);
828     /*
829     * Some drivers still depend on this... XXX
830     */
831     cv_broadcast(&lbolt_cv);
833     vminfo.freemem += freemem;
834     {
835         pgcnt_t maxswap, resv, free;
836         pgcnt_t avail =
837             MAX((spgcnt_t)(availrmem - swapfs_minfree), 0);
839         maxswap = k_anoninfo.ani_mem_resv +
840             k_anoninfo.ani_max + avail;
841         /* Update ani_free */
842         set_anoninfo();
843         free = k_anoninfo.ani_free + avail;
844         resv = k_anoninfo.ani_phys_resv +
845             k_anoninfo.ani_mem_resv;
847         vminfo.swap_resv += resv;
848         /* number of reserved and allocated pages */
849 #ifdef  DEBUG
850         if (maxswap < free)
851             cmn_err(CE_WARN, "clock: maxswap < free");
852         if (maxswap < resv)
853             cmn_err(CE_WARN, "clock: maxswap < resv");
854 #endif
855         vminfo.swap_alloc += maxswap - free;
856         vminfo.swap_avail += maxswap - resv;
857         vminfo.swap_free += free;
858     }
859     vminfo.updates++;
860     if (nrunnable) {
861         sysinfo.runque += nrunnable;
862         sysinfo.runocc++;
863     }
864     if (nswapped) {
865         sysinfo.swpque += nswapped;
866         sysinfo.swpocc++;
867     }
868     sysinfo.waiting += w_io;
869     sysinfo.updates++;
871     /*
872     * Wake up fsflush to write out DELWRI
873     * buffers, dirty pages and other cached
874     * administrative data, e.g. inodes.
875     */
876     if (--fsflushcnt <= 0) {
877         fsflushcnt = tune.t_fsflushr;
878         cv_signal(&fsflush_cv);
879     }
881     vmmeter();
882     calcloadavg(genloadavg(&loadavg), hp_avenrun);
883     for (i = 0; i < 3; i++)
884         /*
885         * At the moment avenrun[] can only hold 31
886         * bits of load average as it is a signed
887         * int in the API. We need to ensure that
888         * hp_avenrun[i] >> (16 - FSHIFT) will not be
889         * too large. If it is, we put the largest value
890         * that we can use into avenrun[i]. This is
891         * kludgy, but about all we can do until we

```

```

892         * avenrun[] is declared as an array of uint64[]
893         */
894         if (hp_avenrun[i] < ((uint64_t)1<<(31+16-FSHIFT)))
895             avenrun[i] = (int32_t)(hp_avenrun[i] >>
896                 (16 - FSHIFT));
897         else
898             avenrun[i] = 0x7fffffff;

900     cpupart = cp_list_head;
901     do {
902         calcloadavg(genloadavg(&cpupart->cp_loadavg),
903             cpupart->cp_hp_avenrun);
904     } while ((cpupart = cpupart->cp_next) != cp_list_head);

906     /*
907     * Wake up the swapper thread if necessary.
908     */
909     if (runin ||
910         (runout && (avefree < desfree || wake_sched_sec))) {
911         t = &t0;
912         thread_lock(t);
913         if (t->t_state == TS_STOPPED) {
914             runin = runout = 0;
915             wake_sched_sec = 0;
916             t->t_whystop = 0;
917             t->t_whatstop = 0;
918             t->t_schedflag &= ~TS_ALLSTART;
919             THREAD_TRANSITION(t);
920             setfrontdq(t);
921         }
922         thread_unlock(t);
923     }
924 }

926 /*
927 * Wake up the swapper if any high priority swapped-out threads
928 * became runnable during the last tick.
929 */
930 if (wake_sched) {
931     t = &t0;
932     thread_lock(t);
933     if (t->t_state == TS_STOPPED) {
934         runin = runout = 0;
935         wake_sched = 0;
936         t->t_whystop = 0;
937         t->t_whatstop = 0;
938         t->t_schedflag &= ~TS_ALLSTART;
939         THREAD_TRANSITION(t);
940         setfrontdq(t);
941     }
942     thread_unlock(t);
943 }
944 }

```

unchanged portion omitted

```

*****
21374 Thu Feb 18 08:40:40 2016
new/usr/src/uts/common/os/condvar.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

182 #define cv_block_sig(t, cvp) \
183     { (t)->t_flag |= T_WAKEABLE; cv_block(cvp); }

185 /*
186  * Block on the indicated condition variable and release the
187  * associated kmutex while blocked.
188  */
189 void
190 cv_wait(kcondvar_t *cvp, kmutex_t *mp)
191 {
192     if (panicstr)
193         return;
194     ASSERT(!quiesce_active);

196     ASSERT(curthread->t_schedflag & TS_DONT_SWAP);
196     thread_lock(curthread); /* lock the thread */
197     cv_block((condvar_impl_t *)cvp);
198     thread_unlock_nopreempt(curthread); /* unlock the waiters field */
199     mutex_exit(mp);
200     swtch();
201     mutex_enter(mp);
202 }

_____unchanged_portion_omitted_____

303 int
304 cv_wait_sig(kcondvar_t *cvp, kmutex_t *mp)
305 {
306     kthread_t *t = curthread;
307     proc_t *p = ttoproc(t);
308     klwp_t *lwp = ttolwp(t);
309     int cancel_pending;
310     int rval = 1;
311     int signalled = 0;

313     if (panicstr)
314         return (rval);
315     ASSERT(!quiesce_active);

317     /*
318     * Threads in system processes don't process signals. This is
319     * true both for standard threads of system processes and for
320     * interrupt threads which have borrowed their pinned thread's LWP.
321     */
322     if (lwp == NULL || (p->p_flag & SSYS)) {
323         cv_wait(cvp, mp);
324         return (rval);
325     }
326     ASSERT(t->t_intr == NULL);

329     ASSERT(curthread->t_schedflag & TS_DONT_SWAP);
328     cancel_pending = schedctl_cancel_pending();
329     lwp->lwp_asleep = 1;
330     lwp->lwp_sysabort = 0;
331     thread_lock(t);
332     cv_block_sig(t, (condvar_impl_t *)cvp);
333     thread_unlock_nopreempt(t);
334     mutex_exit(mp);
335     if (ISSIG(t, JUSTLOOKING) || MUSTRETURN(p, t) || cancel_pending)
336         setrun(t);

```

```

337     /* ASSERT(no locks are held) */
338     swtch();
339     signalled = (t->t_schedflag & TS_SIGNALLED);
340     t->t_flag &= ~T_WAKEABLE;
341     mutex_enter(mp);
342     if (ISSIG_PENDING(t, lwp, p)) {
343         mutex_exit(mp);
344         if (issig(FORREAL))
345             rval = 0;
346         mutex_enter(mp);
347     }
348     if (lwp->lwp_sysabort || MUSTRETURN(p, t))
349         rval = 0;
350     if (rval != 0 && cancel_pending) {
351         schedctl_cancel_eintr();
352         rval = 0;
353     }
354     lwp->lwp_asleep = 0;
355     lwp->lwp_sysabort = 0;
356     if (rval == 0 && signalled) /* avoid consuming the cv_signal() */
357         cv_signal(cvp);
358     return (rval);
359 }

_____unchanged_portion_omitted_____

517 /*
518  * Like cv_wait_sig_swap but allows the caller to indicate (with a
519  * non-NULL sigret) that they will take care of signalling the cv
520  * after wakeup, if necessary. This is a vile hack that should only
521  * be used when no other option is available; almost all callers
522  * should just use cv_wait_sig_swap (which takes care of the cv_signal
523  * stuff automatically) instead.
524  */
525 int
526 cv_wait_sig_swap_core(kcondvar_t *cvp, kmutex_t *mp, int *sigret)
527 {
528     kthread_t *t = curthread;
529     proc_t *p = ttoproc(t);
530     klwp_t *lwp = ttolwp(t);
531     int cancel_pending;
532     int rval = 1;
533     int signalled = 0;

535     if (panicstr)
536         return (rval);

538     /*
539     * Threads in system processes don't process signals. This is
540     * true both for standard threads of system processes and for
541     * interrupt threads which have borrowed their pinned thread's LWP.
542     */
543     if (lwp == NULL || (p->p_flag & SSYS)) {
544         cv_wait(cvp, mp);
545         return (rval);
546     }
547     ASSERT(t->t_intr == NULL);

549     cancel_pending = schedctl_cancel_pending();
550     lwp->lwp_asleep = 1;
551     lwp->lwp_sysabort = 0;
552     thread_lock(t);
553     t->t_kpri_req = 0; /* don't need kernel priority */
554     cv_block_sig(t, (condvar_impl_t *)cvp);
555     /* I can be swapped now */
556     curthread->t_schedflag &= ~TS_DONT_SWAP;
557     thread_unlock_nopreempt(t);

```

```
556     mutex_exit(mp);
557     if (ISSIG(t, JUSTLOOKING) || MUSTRETURN(p, t) || cancel_pending)
558         setrun(t);
559     /* ASSERT(no locks are held) */
560     swtch();
561     signalled = (t->t_schedflag & TS_SIGNALLED);
562     t->t_flag &= ~T_WAKEABLE;
563     /* TS_DONT_SWAP set by disp() */
564     ASSERT(curthread->t_schedflag & TS_DONT_SWAP);
565     mutex_enter(mp);
566     if (ISSIG_PENDING(t, lwp, p)) {
567         mutex_exit(mp);
568         if (issig(FORREAL))
569             rval = 0;
570         mutex_enter(mp);
571     }
572     if (lwp->lwp_sysabort || MUSTRETURN(p, t))
573         rval = 0;
574     if (rval != 0 && cancel_pending) {
575         schedctl_cancel_eintr();
576         rval = 0;
577     }
578     lwp->lwp_asleep = 0;
579     lwp->lwp_sysabort = 0;
580     if (rval == 0) {
581         if (sigret != NULL)
582             *sigret = signalled; /* just tell the caller */
583         else if (signalled)
584             cv_signal(cvp); /* avoid consuming the cv_signal() */
585     }
586     return (rval);
587 }
```

\_\_\_\_\_ unchanged\_portion\_omitted

\*\*\*\*\*  
 93874 Thu Feb 18 08:40:40 2016

new/usr/src/uts/common/os/cpu.c

6583 remove whole-process swapping

\*\*\*\*\*

unchanged portion omitted

```

297 static struct cpu_vm_stats_ks_data {
298     kstat_named_t  pgrec;
299     kstat_named_t  pgfrec;
300     kstat_named_t  pgin;
301     kstat_named_t  pgpgin;
302     kstat_named_t  pgout;
303     kstat_named_t  pgpgout;
304     kstat_named_t  swapin;
305     kstat_named_t  pgswapin;
306     kstat_named_t  swapout;
307     kstat_named_t  pgswapout;
308     kstat_named_t  zfod;
309     kstat_named_t  dfree;
310     kstat_named_t  scan;
311     kstat_named_t  rev;
312     kstat_named_t  hat_fault;
313     kstat_named_t  as_fault;
314     kstat_named_t  maj_fault;
315     kstat_named_t  cow_fault;
316     kstat_named_t  prot_fault;
317     kstat_named_t  softlock;
318     kstat_named_t  kernel_asflt;
319     kstat_named_t  pgrun;
320     kstat_named_t  execpgin;
321     kstat_named_t  execpgout;
322     kstat_named_t  execfree;
323     kstat_named_t  anonpgin;
324     kstat_named_t  anonpgout;
325     kstat_named_t  anonfree;
326     kstat_named_t  fspgin;
327     kstat_named_t  fspgout;
328     kstat_named_t  fsfree;
329 } cpu_vm_stats_ks_data_template = {
330     {"pgrec",          KSTAT_DATA_UINT64 },
331     {"pgfrec",        KSTAT_DATA_UINT64 },
332     {"pgin",          KSTAT_DATA_UINT64 },
333     {"pgpgin",        KSTAT_DATA_UINT64 },
334     {"pgout",         KSTAT_DATA_UINT64 },
335     {"pgpgout",       KSTAT_DATA_UINT64 },
336     {"swapin",        KSTAT_DATA_UINT64 },
337     {"pgswapin",      KSTAT_DATA_UINT64 },
338     {"swapout",       KSTAT_DATA_UINT64 },
339     {"pgswapout",     KSTAT_DATA_UINT64 },
340     {"zfod",          KSTAT_DATA_UINT64 },
341     {"dfree",         KSTAT_DATA_UINT64 },
342     {"scan",          KSTAT_DATA_UINT64 },
343     {"rev",           KSTAT_DATA_UINT64 },
344     {"hat_fault",     KSTAT_DATA_UINT64 },
345     {"as_fault",      KSTAT_DATA_UINT64 },
346     {"maj_fault",     KSTAT_DATA_UINT64 },
347     {"cow_fault",     KSTAT_DATA_UINT64 },
348     {"prot_fault",    KSTAT_DATA_UINT64 },
349     {"softlock",      KSTAT_DATA_UINT64 },
350     {"kernel_asflt", KSTAT_DATA_UINT64 },
351     {"pgrun",         KSTAT_DATA_UINT64 },
352     {"execpgin",      KSTAT_DATA_UINT64 },
353     {"execpgout",     KSTAT_DATA_UINT64 },
354     {"execfree",      KSTAT_DATA_UINT64 },
355     {"anonpgin",      KSTAT_DATA_UINT64 },

```

```

348     { "anonpgout",      KSTAT_DATA_UINT64 },
349     { "anonfree",       KSTAT_DATA_UINT64 },
350     { "fspgin",         KSTAT_DATA_UINT64 },
351     { "fspgout",        KSTAT_DATA_UINT64 },
352     { "fsfree",         KSTAT_DATA_UINT64 },
353 };

```

unchanged portion omitted

```

2515 /*
2516  * Bind a thread to a CPU as requested.
2517  */
2518 int
2519 cpu_bind_thread(kthread_id_t tp, processorid_t bind, processorid_t *obind,
2520               int *error)
2521 {
2522     processorid_t  binding;
2523     cpu_t          *cp = NULL;
2524
2525     ASSERT(MUTEX_HELD(&cpu_lock));
2526     ASSERT(MUTEX_HELD(&ttoproc(tp)->p_lock));
2527
2528     thread_lock(tp);
2529
2530     /*
2531      * Record old binding, but change the obind, which was initialized
2532      * to PBIND_NONE, only if this thread has a binding. This avoids
2533      * reporting PBIND_NONE for a process when some LWPs are bound.
2534      */
2535     binding = tp->t_bind_cpu;
2536     if (binding != PBIND_NONE)
2537         *obind = binding; /* record old binding */
2538
2539     switch (bind) {
2540     case PBIND_QUERY:
2541         /* Just return the old binding */
2542         thread_unlock(tp);
2543         return (0);
2544
2545     case PBIND_QUERY_TYPE:
2546         /* Return the binding type */
2547         *obind = TB_CPU_IS_SOFT(tp) ? PBIND_SOFT : PBIND_HARD;
2548         thread_unlock(tp);
2549         return (0);
2550
2551     case PBIND_SOFT:
2552         /*
2553          * Set soft binding for this thread and return the actual
2554          * binding
2555          */
2556         TB_CPU_SOFT_SET(tp);
2557         thread_unlock(tp);
2558         return (0);
2559
2560     case PBIND_HARD:
2561         /*
2562          * Set hard binding for this thread and return the actual
2563          * binding
2564          */
2565         TB_CPU_HARD_SET(tp);
2566         thread_unlock(tp);
2567         return (0);
2568
2569     default:
2570         break;
2571     }

```

```

2573 /*
2574  * If this thread/LWP cannot be bound because of permission
2575  * problems, just note that and return success so that the
2576  * other threads/LWPs will be bound. This is the way
2577  * processor_bind() is defined to work.
2578  *
2579  * Binding will get EPERM if the thread is of system class
2580  * or hasprocperm() fails.
2581  */
2582 if (tp->t_cid == 0 || !hasprocperm(tp->t_cred, CRED())) {
2583     *error = EPERM;
2584     thread_unlock(tp);
2585     return (0);
2586 }
2588 binding = bind;
2589 if (binding != PBIND_NONE) {
2590     cp = cpu_get((processorid_t)binding);
2591     /*
2592      * Make sure binding is valid and is in right partition.
2593      */
2594     if (cp == NULL || tp->t_cpupart != cp->cpu_part) {
2595         *error = EINVAL;
2596         thread_unlock(tp);
2597         return (0);
2598     }
2599 }
2600 tp->t_bind_cpu = binding; /* set new binding */
2602 /*
2603  * If there is no system-set reason for affinity, set
2604  * the t_bound_cpu field to reflect the binding.
2605  */
2606 if (tp->t_affinitycnt == 0) {
2607     if (binding == PBIND_NONE) {
2608         /*
2609          * We may need to adjust disp_max_unbound_pri
2610          * since we're becoming unbound.
2611          */
2612         disp_adjust_unbound_pri(tp);
2614         tp->t_bound_cpu = NULL; /* set new binding */
2616         /*
2617          * Move thread to lgroup with strongest affinity
2618          * after unbinding
2619          */
2620         if (tp->t_lgrp_affinity)
2621             lgrp_move_thread(tp,
2622                 lgrp_choose(tp, tp->t_cpupart), 1);
2624         if (tp->t_state == TS_ONPROC &&
2625             tp->t_cpu->cpu_part != tp->t_cpupart)
2626             cpu_surrender(tp);
2627     } else {
2628         lpl_t *lpl;
2630         tp->t_bound_cpu = cp;
2631         ASSERT(cp->cpu_lpl != NULL);
2633         /*
2634          * Set home to lgroup with most affinity containing CPU
2635          * that thread is being bound or minimum bounding
2636          * lgroup if no affinities set
2637          */
2638         if (tp->t_lgrp_affinity)

```

```

2639         lpl = lgrp_affinity_best(tp, tp->t_cpupart,
2640             LGRP_NONE, B_FALSE);
2641     else
2642         lpl = cp->cpu_lpl;
2644     if (tp->t_lpl != lpl) {
2645         /* can't grab cpu_lock */
2646         lgrp_move_thread(tp, lpl, 1);
2647     }
2649     /*
2650     * Make the thread switch to the bound CPU.
2651     * If the thread is runnable, we need to
2652     * requeue it even if t_cpu is already set
2653     * to the right CPU, since it may be on a
2654     * kpreempt queue and need to move to a local
2655     * queue. We could check t_disp_queue to
2656     * avoid unnecessary overhead if it's already
2657     * on the right queue, but since this isn't
2658     * a performance-critical operation it doesn't
2659     * seem worth the extra code and complexity.
2660     *
2661     * If the thread is weakbound to the cpu then it will
2662     * resist the new binding request until the weak
2663     * binding drops. The cpu_surrender or requeueing
2664     * below could be skipped in such cases (since it
2665     * will have no effect), but that would require
2666     * thread_allowmigrate to acquire thread_lock so
2667     * we'll take the very occasional hit here instead.
2668     */
2669     if (tp->t_state == TS_ONPROC) {
2670         cpu_surrender(tp);
2671     } else if (tp->t_state == TS_RUN) {
2672         cpu_t *ocp = tp->t_cpu;
2674         (void) dispdeq(tp);
2675         setbackdq(tp);
2676         /*
2677          * On the bound CPU's disp queue now.
2678          * Either on the bound CPU's disp queue now,
2679          * or swapped out or on the swap queue.
2680          */
2681         ASSERT(tp->t_disp_queue == cp->cpu_disp ||
2682             tp->t_weakbound_cpu == ocp);
2683         tp->t_weakbound_cpu == ocp ||
2684         (tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ))
2685         != TS_LOAD);
2686     }
2688     /*
2689     * Our binding has changed; set TP_CHANGEBIND.
2690     */
2691     tp->t_proc_flag |= TP_CHANGEBIND;
2692     aston(tp);
2694     thread_unlock(tp);
2696     return (0);
2697 }
2699 unchanged_portion_omitted
3262 static int
3263 cpu_vm_stats_ks_update(kstat_t *ksp, int rw)
3264 {

```

```

3265     cpu_t *cp = (cpu_t *)ksp->ks_private;
3266     struct cpu_vm_stats_ks_data *cvskd;
3267     cpu_vm_stats_t *cvs;

3269     if (rw == KSTAT_WRITE)
3270         return (EACCES);

3272     cvs = &cp->cpu_stats.vm;
3273     cvskd = ksp->ks_data;

3275     bcopy(&cpu_vm_stats_ks_data_template, ksp->ks_data,
3276           sizeof(cpu_vm_stats_ks_data_template));
3277     cvskd->pgrec.value.ui64 = cvs->pgrec;
3278     cvskd->pgfrec.value.ui64 = cvs->pgfrec;
3279     cvskd->pggin.value.ui64 = cvs->pggin;
3280     cvskd->pgpggin.value.ui64 = cvs->pgpggin;
3281     cvskd->pggout.value.ui64 = cvs->pggout;
3282     cvskd->pgpgout.value.ui64 = cvs->pgpgout;
3294     cvskd->swapin.value.ui64 = cvs->swapin;
3295     cvskd->pgswapin.value.ui64 = cvs->pgswapin;
3296     cvskd->swapout.value.ui64 = cvs->swapout;
3297     cvskd->pgswapout.value.ui64 = cvs->pgswapout;
3283     cvskd->zfod.value.ui64 = cvs->zfod;
3284     cvskd->dfree.value.ui64 = cvs->dfree;
3285     cvskd->scan.value.ui64 = cvs->scan;
3286     cvskd->rev.value.ui64 = cvs->rev;
3287     cvskd->hat_fault.value.ui64 = cvs->hat_fault;
3288     cvskd->as_fault.value.ui64 = cvs->as_fault;
3289     cvskd->maj_fault.value.ui64 = cvs->maj_fault;
3290     cvskd->cow_fault.value.ui64 = cvs->cow_fault;
3291     cvskd->prot_fault.value.ui64 = cvs->prot_fault;
3292     cvskd->softlock.value.ui64 = cvs->softlock;
3293     cvskd->kernel_asflt.value.ui64 = cvs->kernel_asflt;
3294     cvskd->pgrrun.value.ui64 = cvs->pgrrun;
3295     cvskd->execpgin.value.ui64 = cvs->execpgin;
3296     cvskd->execpgout.value.ui64 = cvs->execpgout;
3297     cvskd->execfree.value.ui64 = cvs->execfree;
3298     cvskd->anonpgin.value.ui64 = cvs->anonpgin;
3299     cvskd->anonpgout.value.ui64 = cvs->anonpgout;
3300     cvskd->anonfree.value.ui64 = cvs->anonfree;
3301     cvskd->fspgin.value.ui64 = cvs->fspgin;
3302     cvskd->fspgout.value.ui64 = cvs->fspgout;
3303     cvskd->fsfree.value.ui64 = cvs->fsfree;

3305     return (0);
3306 }

3308 static int
3309 cpu_stat_ks_update(kstat_t *ksp, int rw)
3310 {
3311     cpu_stat_t *cso;
3312     cpu_t *cp;
3313     int i;
3314     hrtime_t msnsecs[NCMSTATES];

3316     cso = (cpu_stat_t *)ksp->ks_data;
3317     cp = (cpu_t *)ksp->ks_private;

3319     if (rw == KSTAT_WRITE)
3320         return (EACCES);

3322     /*
3323     * Read CPU mstate, but compare with the last values we
3324     * received to make sure that the returned kstats never
3325     * decrease.
3326     */

```

```

3328     get_cpu_mstate(cp, msnsecs);
3329     msnsecs[CMS_IDLE] = NSEC_TO_TICK(msnsecs[CMS_IDLE]);
3330     msnsecs[CMS_USER] = NSEC_TO_TICK(msnsecs[CMS_USER]);
3331     msnsecs[CMS_SYSTEM] = NSEC_TO_TICK(msnsecs[CMS_SYSTEM]);
3332     if (cso->cpu_sysinfo.cpu[CPU_IDLE] < msnsecs[CMS_IDLE])
3333         cso->cpu_sysinfo.cpu[CPU_IDLE] = msnsecs[CMS_IDLE];
3334     if (cso->cpu_sysinfo.cpu[CPU_USER] < msnsecs[CMS_USER])
3335         cso->cpu_sysinfo.cpu[CPU_USER] = msnsecs[CMS_USER];
3336     if (cso->cpu_sysinfo.cpu[CPU_KERNEL] < msnsecs[CMS_SYSTEM])
3337         cso->cpu_sysinfo.cpu[CPU_KERNEL] = msnsecs[CMS_SYSTEM];
3338     cso->cpu_sysinfo.cpu[CPU_WAIT] = 0;
3339     cso->cpu_sysinfo.wait[W_IO] = 0;
3340     cso->cpu_sysinfo.wait[W_SWAP] = 0;
3341     cso->cpu_sysinfo.wait[W_PIO] = 0;
3342     cso->cpu_sysinfo.bread = CPU_STATS(cp, sys.bread);
3343     cso->cpu_sysinfo.bwrite = CPU_STATS(cp, sys.bwrite);
3344     cso->cpu_sysinfo.lread = CPU_STATS(cp, sys.lread);
3345     cso->cpu_sysinfo.lwrite = CPU_STATS(cp, sys.lwrite);
3346     cso->cpu_sysinfo.phread = CPU_STATS(cp, sys.phread);
3347     cso->cpu_sysinfo.phwrite = CPU_STATS(cp, sys.phwrite);
3348     cso->cpu_sysinfo.pswitch = CPU_STATS(cp, sys.pswitch);
3349     cso->cpu_sysinfo.trap = CPU_STATS(cp, sys.trap);
3350     cso->cpu_sysinfo.intr = 0;
3351     for (i = 0; i < PIL_MAX; i++)
3352         cso->cpu_sysinfo.intr += CPU_STATS(cp, sys.intr[i]);
3353     cso->cpu_sysinfo.syscall = CPU_STATS(cp, sys.syscall);
3354     cso->cpu_sysinfo.sysread = CPU_STATS(cp, sys.sysread);
3355     cso->cpu_sysinfo.syswrite = CPU_STATS(cp, sys.syswrite);
3356     cso->cpu_sysinfo.sysfork = CPU_STATS(cp, sys.sysfork);
3357     cso->cpu_sysinfo.sysvfork = CPU_STATS(cp, sys.sysvfork);
3358     cso->cpu_sysinfo.sysexec = CPU_STATS(cp, sys.sysexec);
3359     cso->cpu_sysinfo.readch = CPU_STATS(cp, sys.readch);
3360     cso->cpu_sysinfo.writetech = CPU_STATS(cp, sys.writetech);
3361     cso->cpu_sysinfo.rcvint = CPU_STATS(cp, sys.rcvint);
3362     cso->cpu_sysinfo.xmtint = CPU_STATS(cp, sys.xmtint);
3363     cso->cpu_sysinfo.mdmint = CPU_STATS(cp, sys.mdmint);
3364     cso->cpu_sysinfo.rawch = CPU_STATS(cp, sys.rawch);
3365     cso->cpu_sysinfo.canch = CPU_STATS(cp, sys.canch);
3366     cso->cpu_sysinfo.outch = CPU_STATS(cp, sys.outch);
3367     cso->cpu_sysinfo.msg = CPU_STATS(cp, sys.msg);
3368     cso->cpu_sysinfo.sema = CPU_STATS(cp, sys.sema);
3369     cso->cpu_sysinfo.namei = CPU_STATS(cp, sys.namei);
3370     cso->cpu_sysinfo.ufsiget = CPU_STATS(cp, sys.ufsiget);
3371     cso->cpu_sysinfo.ufsdirblk = CPU_STATS(cp, sys.ufsdirblk);
3372     cso->cpu_sysinfo.ufsipage = CPU_STATS(cp, sys.ufsipage);
3373     cso->cpu_sysinfo.ufsinopage = CPU_STATS(cp, sys.ufsinopage);
3374     cso->cpu_sysinfo.inodeovf = 0;
3375     cso->cpu_sysinfo.fileovf = 0;
3376     cso->cpu_sysinfo.procovf = CPU_STATS(cp, sys.procovf);
3377     cso->cpu_sysinfo.intrthread = 0;
3378     for (i = 0; i < LOCK_LEVEL - 1; i++)
3379         cso->cpu_sysinfo.intrthread += CPU_STATS(cp, sys.intr[i]);
3380     cso->cpu_sysinfo.intrblk = CPU_STATS(cp, sys.intrblk);
3381     cso->cpu_sysinfo.idlethread = CPU_STATS(cp, sys.idlethread);
3382     cso->cpu_sysinfo.inv_swtdch = CPU_STATS(cp, sys.inv_swtdch);
3383     cso->cpu_sysinfo.nthreads = CPU_STATS(cp, sys.nthreads);
3384     cso->cpu_sysinfo.cpumigrate = CPU_STATS(cp, sys.cpumigrate);
3385     cso->cpu_sysinfo.xcalls = CPU_STATS(cp, sys.xcalls);
3386     cso->cpu_sysinfo.mutex_adenters = CPU_STATS(cp, sys.mutex_adenters);
3387     cso->cpu_sysinfo.rw_rdfails = CPU_STATS(cp, sys.rw_rdfails);
3388     cso->cpu_sysinfo.rw_wrfails = CPU_STATS(cp, sys.rw_wrfails);
3389     cso->cpu_sysinfo.modload = CPU_STATS(cp, sys.modload);
3390     cso->cpu_sysinfo.modunload = CPU_STATS(cp, sys.modunload);
3391     cso->cpu_sysinfo.bawrite = CPU_STATS(cp, sys.bawrite);
3392     cso->cpu_sysinfo.rw_enters = 0;

```

```
3393     cso->cpu_sysinfo.win_uo_cnt     = 0;
3394     cso->cpu_sysinfo.win_uu_cnt     = 0;
3395     cso->cpu_sysinfo.win_so_cnt     = 0;
3396     cso->cpu_sysinfo.win_su_cnt     = 0;
3397     cso->cpu_sysinfo.win_suo_cnt    = 0;

3399     cso->cpu_syswait.iowait         = CPU_STATS(cp, sys.iowait);
3400     cso->cpu_syswait.swap           = 0;
3401     cso->cpu_syswait.physio         = 0;

3403     cso->cpu_vminfo.pgrec           = CPU_STATS(cp, vm.pgrec);
3404     cso->cpu_vminfo.pgfreq          = CPU_STATS(cp, vm.pgfreq);
3405     cso->cpu_vminfo.pggin           = CPU_STATS(cp, vm.pggin);
3406     cso->cpu_vminfo.pgggin          = CPU_STATS(cp, vm.pgggin);
3407     cso->cpu_vminfo.pgout           = CPU_STATS(cp, vm.pgout);
3408     cso->cpu_vminfo.pggout          = CPU_STATS(cp, vm.pggout);
3424     cso->cpu_vminfo.swapin          = CPU_STATS(cp, vm.swapin);
3425     cso->cpu_vminfo.pgswapin        = CPU_STATS(cp, vm.pgswapin);
3426     cso->cpu_vminfo.swapout         = CPU_STATS(cp, vm.swapout);
3427     cso->cpu_vminfo.pgswapout       = CPU_STATS(cp, vm.pgswapout);
3409     cso->cpu_vminfo.zfod            = CPU_STATS(cp, vm.zfod);
3410     cso->cpu_vminfo.dfree           = CPU_STATS(cp, vm.dfree);
3411     cso->cpu_vminfo.scan            = CPU_STATS(cp, vm.scan);
3412     cso->cpu_vminfo.rev             = CPU_STATS(cp, vm.rev);
3413     cso->cpu_vminfo.hat_fault        = CPU_STATS(cp, vm.hat_fault);
3414     cso->cpu_vminfo.as_fault        = CPU_STATS(cp, vm.as_fault);
3415     cso->cpu_vminfo.maj_fault       = CPU_STATS(cp, vm.maj_fault);
3416     cso->cpu_vminfo.cow_fault       = CPU_STATS(cp, vm.cow_fault);
3417     cso->cpu_vminfo.prot_fault      = CPU_STATS(cp, vm.prot_fault);
3418     cso->cpu_vminfo.softlock        = CPU_STATS(cp, vm.softlock);
3419     cso->cpu_vminfo.kernel_asflt    = CPU_STATS(cp, vm.kernel_asflt);
3420     cso->cpu_vminfo.pgrrun          = CPU_STATS(cp, vm.pgrrun);
3421     cso->cpu_vminfo.execpgin        = CPU_STATS(cp, vm.execpgin);
3422     cso->cpu_vminfo.execpgout       = CPU_STATS(cp, vm.execpgout);
3423     cso->cpu_vminfo.execfree        = CPU_STATS(cp, vm.execfree);
3424     cso->cpu_vminfo.anonpgin        = CPU_STATS(cp, vm.anonpgin);
3425     cso->cpu_vminfo.anonpgout       = CPU_STATS(cp, vm.anonpgout);
3426     cso->cpu_vminfo.anonfree        = CPU_STATS(cp, vm.anonfree);
3427     cso->cpu_vminfo.fspgin          = CPU_STATS(cp, vm.fspgin);
3428     cso->cpu_vminfo.fspgout         = CPU_STATS(cp, vm.fspgout);
3429     cso->cpu_vminfo.fsfree          = CPU_STATS(cp, vm.fsfree);

3431     return (0);
3432 }
```

unchanged\_portion\_omitted



```

*****
15482 Thu Feb 18 08:40:40 2016
new/usr/src/uts/common/os/panic.c
6583 remove whole-process swapping
*****
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) 1999, 2010, Oracle and/or its affiliates. All rights reserved.
23 */
24
25 /*
26 * Copyright (c) 2011, Joyent, Inc. All rights reserved.
27 */
28
29 /*
30 * When the operating system detects that it is in an invalid state, a panic
31 * is initiated in order to minimize potential damage to user data and to
32 * facilitate debugging. There are three major tasks to be performed in
33 * a system panic: recording information about the panic in memory (and thus
34 * making it part of the crash dump), synchronizing the file systems to
35 * preserve user file data, and generating the crash dump. We define the
36 * system to be in one of four states with respect to the panic code:
37 *
38 * CALM - the state of the system prior to any thread initiating a panic
39 *
40 * QUIESCE - the state of the system when the first thread to initiate
41 * a system panic records information about the cause of the panic
42 * and renders the system quiescent by stopping other processors
43 *
44 * SYNC - the state of the system when we synchronize the file systems
45 * DUMP - the state when we generate the crash dump.
46 *
47 * The transitions between these states are irreversible: once we begin
48 * panicking, we only make one attempt to perform the actions associated with
49 * each state.
50 *
51 * The panic code itself must be re-entrant because actions taken during any
52 * state may lead to another system panic. Additionally, any Solaris
53 * thread may initiate a panic at any time, and so we must have synchronization
54 * between threads which attempt to initiate a state transition simultaneously.
55 * The panic code makes use of a special locking primitive, a trigger, to
56 * perform this synchronization. A trigger is simply a word which is set
57 * atomically and can only be set once. We declare three triggers, one for
58 * each transition between the four states. When a thread enters the panic
59 * code it attempts to set each trigger; if it fails it moves on to the
60 * next trigger. A special case is the first trigger: if two threads race
61 * to perform the transition to QUIESCE, the losing thread may execute before

```

```

62 * the winner has a chance to stop its CPU. To solve this problem, we have
63 * the loser look ahead to see if any other triggers are set; if not, it
64 * presumes a panic is underway and simply spins. Unfortunately, since we
65 * are panicking, it is not possible to know this with absolute certainty.
66 *
67 * There are two common reasons for re-entering the panic code once a panic
68 * has been initiated: (1) after we debug_enter() at the end of QUIESCE,
69 * the operator may type "sync" instead of "go", and the PROM's sync callback
70 * routine will invoke panic(); (2) if the clock routine decides that sync
71 * or dump is not making progress, it will invoke panic() to force a timeout.
72 * The design assumes that a third possibility, another thread causing an
73 * unrelated panic while sync or dump is still underway, is extremely unlikely.
74 * If this situation occurs, we may end up triggering dump while sync is
75 * still in progress. This third case is considered extremely unlikely because
76 * all other CPUs are stopped and low-level interrupts have been blocked.
77 *
78 * The panic code is entered via a call directly to the vpanic() function,
79 * or its varargs wrappers panic() and cmn_err(9F). The vpanic routine
80 * is implemented in assembly language to record the current machine
81 * registers, attempt to set the trigger for the QUIESCE state, and
82 * if successful, switch stacks on to the panic_stack before calling into
83 * the common panicsys() routine. The first thread to initiate a panic
84 * is allowed to make use of the reserved panic_stack so that executing
85 * the panic code itself does not overwrite valuable data on that thread's
86 * stack *ahead* of the current stack pointer. This data will be preserved
87 * in the crash dump and may prove invaluable in determining what this
88 * thread has previously been doing. The first thread, saved in panic_thread,
89 * is also responsible for stopping the other CPUs as quickly as possible,
90 * and then setting the various panic_* variables. Most important among
91 * these is panicstr, which allows threads to subsequently bypass held
92 * locks so that we can proceed without ever blocking. We must stop the
93 * other CPUs *prior* to setting panicstr in case threads running there are
94 * currently spinning to acquire a lock; we want that state to be preserved.
95 * Every thread which initiates a panic has its T_PANIC flag set so we can
96 * identify all such threads in the crash dump.
97 *
98 * The panic_thread is also allowed to make use of the special memory buffer
99 * panicbuf, which on machines with appropriate hardware is preserved across
100 * reboots. We allow the panic_thread to store its register set and panic
101 * message in this buffer, so even if we fail to obtain a crash dump we will
102 * be able to examine the machine after reboot and determine some of the
103 * state at the time of the panic. If we do get a dump, the panic buffer
104 * data is structured so that a debugger can easily consume the information
105 * therein (see <sys/panic.h>).
106 *
107 * Each platform or architecture is required to implement the functions
108 * panic_savetrap() to record trap-specific information to panicbuf,
109 * panic_saverregs() to record a register set to panicbuf, panic_stopcpu()
110 * to halt all CPUs but the panicking CPU, panic_quiesce_hw() to perform
111 * miscellaneous platform-specific tasks *after* panicstr is set,
112 * panic_showtrap() to print trap-specific information to the console,
113 * and panic_dump_hw() to perform platform tasks prior to calling dumpsys().
114 *
115 * A Note on Word Formation, courtesy of the Oxford Guide to English Usage:
116 *
117 * Words ending in -c interpose k before suffixes which otherwise would
118 * indicate a soft c, and thus the verb and adjective forms of 'panic' are
119 * spelled "panicked", "panicking", and "panicky" respectively. Use of
120 * the ill-conceived "panicing" and "panic'd" is discouraged.
121 */
122
123 #include <sys/types.h>
124 #include <sys/varargs.h>
125 #include <sys/sysmacros.h>
126 #include <sys/cmn_err.h>
127 #include <sys/cpuvar.h>

```

```

128 #include <sys/thread.h>
129 #include <sys/t_lock.h>
130 #include <sys/cred.h>
131 #include <sys/systm.h>
132 #include <sys/archsystem.h>
133 #include <sys/uadmin.h>
134 #include <sys/callb.h>
135 #include <sys/vfs.h>
136 #include <sys/log.h>
137 #include <sys/disp.h>
138 #include <sys/param.h>
139 #include <sys/dumphdr.h>
140 #include <sys/ftrace.h>
141 #include <sys/reboot.h>
142 #include <sys/debug.h>
143 #include <sys/stack.h>
144 #include <sys/spl.h>
145 #include <sys/errorq.h>
146 #include <sys/panic.h>
147 #include <sys/fm/util.h>
148 #include <sys/clock_impl.h>

150 /*
151  * Panic variables which are set once during the QUIESCE state by the
152  * first thread to initiate a panic. These are examined by post-mortem
153  * debugging tools; the inconsistent use of 'panic' versus 'panic_' in
154  * the variable naming is historical and allows legacy tools to work.
155  */
156 #pragma align STACK_ALIGN(panic_stack)
157 char panic_stack[PANICSTKSIZE]; /* reserved stack for panic_thread */
158 kthread_t *panic_thread; /* first thread to call panicsys() */
159 cpu_t panic_cpu; /* cpu from first call to panicsys() */
160 label_t panic_regs; /* setjmp label from panic_thread */
161 label_t panic_pcb; /* t_pcb at time of panic */
162 struct regs *panic_reg; /* regs struct from first panicsys() */
163 char *volatile panicstr; /* format string to first panicsys() */
164 va_list panicargs; /* arguments to first panicsys() */
165 clock_t panic_lbolt; /* lbolt at time of panic */
166 int64_t panic_lbolt64; /* lbolt64 at time of panic */
167 hrtime_t panic_hrttime; /* hrttime at time of panic */
168 timespec_t panic_hrestime; /* hrestime at time of panic */
169 int panic_ipl; /* ipl on panic_cpu at time of panic */
170 ushort_t panic_schedflag; /* t_schedflag for panic_thread */
171 cpu_t *panic_bound_cpu; /* t_bound_cpu for panic_thread */
172 char panic_preempt; /* t_preempt for panic_thread */

174 /*
175  * Panic variables which can be set via /etc/system or patched while
176  * the system is in operation. Again, the stupid names are historic.
177  */
178 char *panic_bootstr = NULL; /* mddbboot string to use after panic */
179 int panic_bootfcn = AD_BOOT; /* mddbboot function to use after panic */
180 int halt_on_panic = 0; /* halt after dump instead of reboot? */
181 int nopanicdebug = 0; /* reboot instead of call debugger? */
182 int in_sync = 0; /* skip vfs_syncall() and just dump? */

184 /*
185  * The do_polled_io flag is set by the panic code to inform the SCSI subsystem
186  * to use polled mode instead of interrupt-driven i/o.
187  */
188 int do_polled_io = 0;

190 /*
191  * The panic_forced flag is set by the uadmin A_DUMP code to inform the
192  * panic subsystem that it should not attempt an initial debug_enter.
193  */

```

```

194 int panic_forced = 0;

196 /*
197  * Triggers for panic state transitions:
198  */
199 int panic_quiesce; /* trigger for CALM -> QUIESCE */
200 int panic_sync; /* trigger for QUIESCE -> SYNC */
201 int panic_dump; /* trigger for SYNC -> DUMP */

203 /*
204  * Variable signifying quiesce(9E) is in progress.
205  */
206 volatile int quiesce_active = 0;

208 void
209 panicsys(const char *format, va_list alist, struct regs *rp, int on_panic_stack)
210 {
211     int s = spl8();
212     kthread_t *t = curthread;
213     cpu_t *cp = CPU;

215     caddr_t intr_stack = NULL;
216     uint_t intr_actv;

218     ushort_t schedflag = t->t_schedflag;
219     cpu_t *bound_cpu = t->t_bound_cpu;
220     char preempt = t->t_preempt;
221     label_t pcb = t->t_pcb;

223     (void) setjmp(&t->t_pcb);
224     t->t_flag |= T_PANIC;

226     t->t_schedflag /= TS_DONT_SWAP;
226     t->t_bound_cpu = cp;
227     t->t_preempt++;

229     panic_enter_hw(s);

231     /*
232     * If we're on the interrupt stack and an interrupt thread is available
233     * in this CPU's pool, preserve the interrupt stack by detaching an
234     * interrupt thread and making its stack the intr_stack.
235     */
236     if (CPU_ON_INTR(cp) && cp->cpu_intr_thread != NULL) {
237         kthread_t *it = cp->cpu_intr_thread;

239         intr_stack = cp->cpu_intr_stack;
240         intr_actv = cp->cpu_intr_actv;

242         cp->cpu_intr_stack = thread_stk_init(it->t_stk);
243         cp->cpu_intr_thread = it->t_link;

245         /*
246         * Clear only the high level bits of cpu_intr_actv.
247         * We want to indicate that high-level interrupts are
248         * not active without destroying the low-level interrupt
249         * information stored there.
250         */
251         cp->cpu_intr_actv &= ((1 << (LOCK_LEVEL + 1)) - 1);
252     }

254     /*
255     * Record one-time panic information and quiesce the other CPUs.
256     * Then print out the panic message and stack trace.
257     */
258     if (on_panic_stack) {

```

```

259         panic_data_t *pdp = (panic_data_t *)panicbuf;

261         pdp->pd_version = PANICBUFVERS;
262         pdp->pd_msgoff = sizeof (panic_data_t) - sizeof (panic_nv_t);

264         (void) strncpy(pdp->pd_uid, dump_get_uid(),
265             sizeof (pdp->pd_uid));

267         if (t->t_panic_trap != NULL)
268             panic_savetrap(pdp, t->t_panic_trap);
269         else
270             panic_saveregs(pdp, rp);

272         (void) vsnprintf(&panicbuf[pdp->pd_msgoff],
273             PANICBUFSIZE - pdp->pd_msgoff, format, alist);

275         /*
276          * Call into the platform code to stop the other CPUs.
277          * We currently have all interrupts blocked, and expect that
278          * the platform code will lower ipl only as far as needed to
279          * perform cross-calls, and will acquire as *few* locks as is
280          * possible -- panicstr is not set so we can still deadlock.
281          */
282         panic_stopcpus(cp, t, s);

284         panicstr = (char *)format;
285         va_copy(panicargs, alist);
286         panic_lbolt = LBOLT_NO_ACCOUNT;
287         panic_lbolt64 = LBOLT_NO_ACCOUNT64;
288         panic_hrestime = hrestime;
289         panic_hrtime = gethrtime_waitfree();
290         panic_thread = t;
291         panic_regs = t->t_pcb;
292         panic_reg = rp;
293         panic_cpu = *cp;
294         panic_ipl = spltoipl(s);
295         panic_schedflag = schedflag;
296         panic_bound_cpu = bound_cpu;
297         panic_preempt = preempt;
298         panic_pcb = pcb;

300         if (intr_stack != NULL) {
301             panic_cpu.cpu_intr_stack = intr_stack;
302             panic_cpu.cpu_intr_actv = intr_actv;
303         }

305         /*
306          * Lower ipl to 10 to keep clock() from running, but allow
307          * keyboard interrupts to enter the debugger. These callbacks
308          * are executed with panicstr set so they can bypass locks.
309          */
310         splx(ipltospl(CLOCK_LEVEL));
311         panic_quiesce_hw(pdp);
312         (void) FTRACE_STOP();
313         (void) callb_execute_class(CB_CL_PANIC, NULL);

315         if (log_intrq != NULL)
316             log_flushq(log_intrq);

318         /*
319          * If log_consq has been initialized and syslogd has started,
320          * print any messages in log_consq that haven't been consumed.
321          */
322         if (log_consq != NULL && log_consq != log_backlogq)
323             log_printq(log_consq);

```

```

325         fm_banner();

327 #if defined(__x86)
328         /*
329          * A hypervisor panic originates outside of Solaris, so we
330          * don't want to prepend the panic message with misleading
331          * pointers from within Solaris.
332          */
333         if (!IN_XPV_PANIC())
334 #endif
335             printf("\n\rpanic[cpu%d]/thread=%p: ", cp->cpu_id,
336                 (void *)t);
337         vprintf(format, alist);
338         printf("\n\n");

340         if (t->t_panic_trap != NULL) {
341             panic_showtrap(t->t_panic_trap);
342             printf("\n");
343         }

345         traceregs(rp);
346         printf("\n");

348         if (((boothowto & RB_DEBUG) || obpdebug) &&
349             !npanicdebug && !panic_forced) {
350             if (dumpvp != NULL) {
351                 debug_enter("panic: entering debugger "
352                     "(continue to save dump)");
353             } else {
354                 debug_enter("panic: entering debugger "
355                     "(no dump device, continue to reboot)");
356             }
357         }

359     } else if (panic_dump != 0 || panic_sync != 0 || panicstr != NULL) {
360         printf("\n\rpanic[cpu%d]/thread=%p: ", cp->cpu_id, (void *)t);
361         vprintf(format, alist);
362         printf("\n");
363     } else
364         goto spin;

366     /*
367      * Prior to performing sync or dump, we make sure that do_polled_io is
368      * set, but we'll leave ipl at 10; deadman(), a CY_HIGH_LEVEL cyclic,
369      * will re-enter panic if we are not making progress with sync or dump.
370      */

372     /*
373      * Sync the filesystems. Reset t_cred if not set because much of
374      * the filesystem code depends on CRED() being valid.
375      */
376     if (!in_sync && panic_trigger(&panic_sync)) {
377         if (t->t_cred == NULL)
378             t->t_cred = kcred;
379         splx(ipltospl(CLOCK_LEVEL));
380         do_polled_io = 1;
381         vfs_syncall();
382     }

384     /*
385      * Take the crash dump. If the dump trigger is already set, try to
386      * enter the debugger again before rebooting the system.
387      */
388     if (panic_trigger(&panic_dump)) {
389         panic_dump_hw(s);
390         splx(ipltospl(CLOCK_LEVEL));

```

```
391         errorq_panic();
392         do_polled_io = 1;
393         dumpsys();
394     } else if (((boothowto & RB_DEBUG) || obpdebug) && !npanicdebug) {
395         debug_enter("panic: entering debugger (continue to reboot)");
396     } else
397         printf("dump aborted: please record the above information!\n");

399     if (halt_on_panic)
400         mboot(A_REBOOT, AD_HALT, NULL, B_FALSE);
401     else
402         mboot(A_REBOOT, panic_bootfcn, panic_bootstr, B_FALSE);
403 spin:
404     /*
405     * Restore ipl to at most CLOCK_LEVEL so we don't end up spinning
406     * and unable to jump into the debugger.
407     */
408     splx(MIN(s, ipltospl(CLOCK_LEVEL)));
409     for (;;)
410         ;
411 }
_____unchanged_portion_omitted_____
```

```

*****
2754 Thu Feb 18 08:40:40 2016
new/usr/src/uts/common/os/sched.c
6583 remove whole-process swapping
*****
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
27 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
28 /*      All Rights Reserved      */
29
30 #include <sys/param.h>
31 #include <sys/types.h>
32 #include <sys/sysmacros.h>
33 #include <sys/system.h>
34 #include <sys/proc.h>
35 #include <sys/cpuvar.h>
36 #include <sys/var.h>
37 #include <sys/tuneable.h>
38 #include <sys/cmn_err.h>
39 #include <sys/buf.h>
40 #include <sys/disp.h>
41 #include <sys/vmsystem.h>
42 #include <sys/vmparam.h>
43 #include <sys/class.h>
44 #include <sys/vtrace.h>
45 #include <sys/modctl.h>
46 #include <sys/debug.h>
47 #include <sys/tnf_probe.h>
48 #include <sys/procfs.h>
49
50 #include <vm/seg.h>
51 #include <vm/seg_kp.h>
52 #include <vm/as.h>
53 #include <vm/rm.h>
54 #include <vm/seg_kmem.h>
55 #include <sys/callb.h>
56
57 /*
58  * The swapper sleeps on runout when there is no one to swap in.
59  * It sleeps on runin when it could not find space to swap someone
60  * in or after swapping someone in.
61  */

```

```

62 char      runout;
63 char      runin;
64 char      wake_sched; /* flag tells clock to wake swapper on next tick */
65 char      wake_sched_sec; /* flag tells clock to wake swapper after a second */
66
67 /*
68  * The swapper swaps processes to reduce memory demand and runs
69  * when avefree < desfree. The swapper resorts to SOFTSWAP when
70  * avefree < desfree which results in swapping out all processes
71  * sleeping for more than maxslp seconds. HARDSWAP occurs when the
72  * system is on the verge of thrashing and this results in swapping
73  * out runnable threads or threads sleeping for less than maxslp secs.
74  *
75  * The swapper runs through all the active processes in the system
76  * and invokes the scheduling class specific swapin/swapout routine
77  * for every thread in the process to obtain an effective priority
78  * for the process. A priority of -1 implies that the thread isn't
79  * swappable. This effective priority is used to find the most
80  * eligible process to swapout or swapin.
81  *
82  * NOTE: Threads which have been swapped are not linked on any
83  *       queue and their dispatcher lock points at the "swapped_lock".
84  *
85  * Processes containing threads with the TS_DONT_SWAP flag set cannot be
86  * swapped out immediately by the swapper. This is due to the fact that
87  * such threads may be holding locks which may be needed by the swapper
88  * to push its pages out. The TS_SWAPENQ flag is set on such threads
89  * to prevent them running in user mode. When such threads reach a
90  * safe point (i.e., are not holding any locks - CL_TRAPRET), they
91  * queue themselves onto the swap queue which is processed by the
92  * swapper. This results in reducing memory demand when the system
93  * is desperate for memory as the thread can't run in user mode.
94  *
95  * The swap queue consists of threads, linked via t_link, which are
96  * haven't been swapped, are runnable but not on the run queue. The
97  * swap queue is protected by the "swapped_lock". The dispatcher
98  * lock (t_lockp) of all threads on the swap queue points at the
99  * "swapped_lock". Thus, the entire queue and/or threads on the
100 * queue can be locked by acquiring "swapped_lock".
101 */
102 static kthread_t *tswap_queue;
103 extern disp_lock_t swapped_lock; /* protects swap queue and threads on it */
104
105 int      maxslp = 0;
106 pgcnt_t avefree; /* 5 sec moving average of free memory */
107 pgcnt_t avefree30; /* 30 sec moving average of free memory */
108
109
110 /*
111  * Minimum size used to decide if sufficient memory is available
112  * before a process is swapped in. This is necessary since in most
113  * cases the actual size of a process (p_swrss) being swapped in
114  * is usually 2 pages (kernel stack pages). This is due to the fact
115  * almost all user pages of a process are stolen by pageout before
116  * the swapper decides to swapout it out.
117  */
118 int      min_procsiz = 12;
119
120 static int      swapin(proc_t *);
121 static int      swapout(proc_t *, uint_t *, int);
122 static void      process_swap_queue();
123
124 #ifdef __sparc
125 extern void lwp_swapin(kthread_t *);
126 #endif /* __sparc */
127
128 /*

```

```

128 * Counters to keep track of the number of swapins or swapouts.
129 */
130 uint_t tot_swapped_in, tot_swapped_out;
131 uint_t softswap, hardswap, swapqswap;

133 /*
134 * Macro to determine if a process is eligible to be swapped.
135 */
136 #define not_swappable(p) \
137     (((p)->p_flag & SSYS) || (p)->p_stat == SIDL || \
138      (p)->p_stat == SZOMB || (p)->p_as == NULL || \
139      (p)->p_as == &kas)

141 /*
142 * Memory scheduler.
143 */
144 void
145 sched()
146 {
147     kthread_id_t    t;
148     pri_t           proc_pri;
149     pri_t           thread_pri;
150     pri_t           swapin_pri;
151     int             desperate;
152     pgcnt_t         needs;
153     int             divisor;
154     proc_t          *prp;
155     proc_t          *swapout_prp;
156     proc_t          *swapin_prp;
157     spgcnt_t        avail;
158     int             chosen_pri;
159     time_t          swapout_time;
160     time_t          swapin_proc_time;
161     callb_cpr_t     cprinfo;
162     kmutex_t        swap_cpr_lock;

163     mutex_init(&swap_cpr_lock, NULL, MUTEX_DEFAULT, NULL);
164     CALLB_CPR_INIT(&cprinfo, &swap_cpr_lock, callb_generic_cpr, "sched");
165     if (maxslp == 0)
166         maxslp = MAXSLP;
167 loop:
168     needs = 0;
169     desperate = 0;
170
171     for (;;) {
172         swapin_pri = v.v_nglobpris;
173         swapin_prp = NULL;
174         chosen_pri = -1;

175         process_swap_queue();

176     /*
177      * Set desperate if
178      * 1. At least 2 runnable processes (on average).
179      * 2. Short (5 sec) and longer (30 sec) average is less
180      *    than minfree and desfree respectively.
181      * 3. Pagein + pageout rate is excessive.
182      */
183     if (avenrun[0] >= 2 * FSCALE &&
184         (MAX(avefree, avefree30) < desfree) &&
185         (pginrate + pgoutrate > maxpgio || avefree < minfree)) {
186         TRACE_4(TR_FAC_SCHED, TR_DESPERATE,
187             "desp:avefree: %d, avefree30: %d, freemem: %d"
188             " pginrate: %d\n", avefree, avefree30, freemem, pginrate);
189         desperate = 1;
190         goto unload;
191     }

```

```

193     }
194
195     /*
196      * Search list of processes to swapin and swapout deadwood.
197      */
198     swapin_proc_time = 0;
199 top:
200     mutex_enter(&pidlock);
201     for (prp = practive; prp != NULL; prp = prp->p_next) {
202         if (not_swappable(prp))
203             continue;

204     /*
205      * Look at processes with at least one swapped lwp.
206      */
207     if (prp->p_swapcnt) {
208         time_t proc_time;

209         /*
210          * Higher priority processes are good candidates
211          * to swapin.
212          */
213         mutex_enter(&prp->p_lock);
214         proc_pri = -1;
215         t = prp->p_tlist;
216         proc_time = 0;
217         do {
218             if (t->t_schedflag & TS_LOAD)
219                 continue;

220             thread_lock(t);
221             thread_pri = CL_SWAPIN(t, 0);
222             thread_unlock(t);

223             if (t->t_stime - proc_time > 0)
224                 proc_time = t->t_stime;
225             if (thread_pri > proc_pri)
226                 proc_pri = thread_pri;
227         } while ((t = t->t_forw) != prp->p_tlist);
228         mutex_exit(&prp->p_lock);

229         if (proc_pri == -1)
230             continue;

231         TRACE_3(TR_FAC_SCHED, TR_CHOOSE_SWAPIN,
232             "prp %p epr %d proc_time %d",
233             prp, proc_pri, proc_time);

234         /*
235          * Swapin processes with a high effective priority.
236          */
237         if (swapin_prp == NULL || proc_pri > chosen_pri) {
238             swapin_prp = prp;
239             chosen_pri = proc_pri;
240             swapin_pri = proc_pri;
241             swapin_proc_time = proc_time;
242         }
243     } else {
244         /*
245          * No need to soft swap if we have sufficient
246          * memory.
247          */
248         if (avefree > desfree ||
249             avefree < desfree && freemem > desfree)
250             continue;

```

```

259     /*
260      * Skip processes that are exiting
261      * or whose address spaces are locked.
262      */
263     mutex_enter(&prp->p_lock);
264     if ((prp->p_flag & SEXITING) ||
265         (prp->p_as != NULL && AS_ISPGLCK(prp->p_as))) {
266         mutex_exit(&prp->p_lock);
267         continue;
268     }
269
270     /*
271      * Softswapping to kick out deadwood.
272      */
273     proc_pri = -1;
274     t = prp->p_tlist;
275     do {
276         if ((t->t_schedflag & (TS_SWAPENQ |
277             TS_ON_SWAPQ | TS_LOAD)) != TS_LOAD)
278             continue;
279
280         thread_lock(t);
281         thread_pri = CL_SWAPOUT(t, SOFTSWAP);
282         thread_unlock(t);
283         if (thread_pri > proc_pri)
284             proc_pri = thread_pri;
285     } while ((t = t->t_forw) != prp->p_tlist);
286
287     if (proc_pri != -1) {
288         uint_t swrss;
289
290         mutex_exit(&pidlock);
291
292         TRACE_1(TR_FAC_SCHED, TR_SOFTSWAP,
293             "softswap:prp %p", prp);
294
295         (void) swapout(prp, &swrss, SOFTSWAP);
296         softswap++;
297         prp->p_swrss += swrss;
298         mutex_exit(&prp->p_lock);
299         goto top;
300     }
301     mutex_exit(&prp->p_lock);
302 }
303
304 if (swapin_prp != NULL)
305     mutex_enter(&swapin_prp->p_lock);
306 mutex_exit(&pidlock);
307
308 if (swapin_prp == NULL) {
309     TRACE_3(TR_FAC_SCHED, TR_RUNOUT,
310         "schedrunout:runout nswapped: %d, avefree: %ld freemem: %ld",
311         nswapped, avefree, freemem);
312
313     t = curthread;
314     thread_lock(t);
315     runout++;
316     t->t_schedflag |= (TS_ALLSTART & ~TS_CSTART);
317     t->t_whystop = PR_SUSPENDED;
318     t->t_whatstop = SUSPEND_NORMAL;
319     (void) new_mstate(t, LMS_SLEEP);
320     mutex_enter(&swap_cpr_lock);
321     CALLB_CPR_SAFE_BEGIN(&cprinfo);
322     mutex_exit(&swap_cpr_lock);
323     thread_stop(t); /* change state and drop lock */
324     swtch();

```

```

325         mutex_enter(&swap_cpr_lock);
326         CALLB_CPR_SAFE_END(&cprinfo, &swap_cpr_lock);
327         mutex_exit(&swap_cpr_lock);
328         goto loop;
329     }
330
331     /*
332      * Decide how deserving this process is to be brought in.
333      * Needs is an estimate of how much core the process will
334      * need. If the process has been out for a while, then we
335      * will bring it in with 1/2 the core needed, otherwise
336      * we are conservative.
337      */
338     divisor = 1;
339     swapout_time = (ddi_get_lbolt() - swapin_proc_time) / hz;
340     if (swapout_time > maxslp / 2)
341         divisor = 2;
342
343     needs = MIN(swapin_prp->p_swrss, lotsfree);
344     needs = MAX(needs, min_procsize);
345     needs = needs / divisor;
346
347     /*
348      * Use freemem, since we want processes to be swapped
349      * in quickly.
350      */
351     avail = freemem - deficit;
352     if (avail > (spgcnt_t)needs) {
353         deficit += needs;
354
355         TRACE_2(TR_FAC_SCHED, TR_SWAPIN_VALUES,
356             "swapin_values: prp %p needs %lu", swapin_prp, needs);
357
358         if (swapin(swapin_prp)) {
359             mutex_exit(&swapin_prp->p_lock);
360             goto loop;
361         }
362         deficit -= MIN(needs, deficit);
363         mutex_exit(&swapin_prp->p_lock);
364     } else {
365         mutex_exit(&swapin_prp->p_lock);
366         /*
367          * If deficit is high, too many processes have been
368          * swapped in so wait a sec before attempting to
369          * swapin more.
370          */
371         if (freemem > needs) {
372             TRACE_2(TR_FAC_SCHED, TR_HIGH_DEFICIT,
373                 "deficit: prp %p needs %lu", swapin_prp, needs);
374             goto block;
375         }
376     }
377
378     TRACE_2(TR_FAC_SCHED, TR_UNLOAD,
379         "unload: prp %p needs %lu", swapin_prp, needs);
380
381 unload:
382     /*
383      * Unload all unloadable modules, free all other memory
384      * resources we can find, then look for a thread to
385      * hardswap.
386      * resources we can find, then look for a thread to hardswap.
387      */
388     modreap();
389     segkp_cache_free();

```

```

389     swapout_prp = NULL;
390     mutex_enter(&pidlock);
391     for (prp = practive; prp != NULL; prp = prp->p_next) {

393         /*
394          * No need to soft swap if we have sufficient
395          * memory.
396          */
397         if (not_swappable(prp))
398             continue;

400         if (avefree > minfree ||
401             avefree < minfree && freemem > desfree) {
402             swapout_prp = NULL;
403             break;
404         }
405     }

406     /*
407     * Skip processes that are exiting
408     * or whose address spaces are locked.
409     */
410     mutex_enter(&prp->p_lock);
411     if ((prp->p_flag & SEXITING) ||
412         (prp->p_as != NULL && AS_ISPLCK(prp->p_as))) {
413         mutex_exit(&prp->p_lock);
414         continue;
415     }

417     proc_pri = -1;
418     t = prp->p_tlist;
419     do {
420         if ((t->t_schedflag & (TS_SWAPENQ |
421             TS_ON_SWAPQ | TS_LOAD)) != TS_LOAD)
422             continue;

424         thread_lock(t);
425         thread_pri = CL_SWAPOUT(t, HARDSWAP);
426         thread_unlock(t);
427         if (thread_pri > proc_pri)
428             proc_pri = thread_pri;
429     } while ((t = t->t_forw) != prp->p_tlist);

431     mutex_exit(&prp->p_lock);
432     if (proc_pri == -1)
433         continue;

435     /*
436     * Swapout processes sleeping with a lower priority
437     * than the one currently being swapped in, if any.
438     */
439     if (swapin_prp == NULL || swapin_pri > proc_pri) {
440         TRACE_2(TR_FAC_SCHED, TR_CHOOSE_SWAPOUT,
441             "hardswap: prp %p needs %lu", prp, needs);

443         if (swapout_prp == NULL || proc_pri < chosen_pri) {
444             swapout_prp = prp;
445             chosen_pri = proc_pri;
446         }
447     }
448 }

450 /*
451 * Acquire the "p_lock" before dropping "pidlock"
452 * to prevent the proc structure from being freed
453 * if the process exits before swapout completes.
454 */

```

```

455     if (swapout_prp != NULL)
456         mutex_enter(&swapout_prp->p_lock);
457     mutex_exit(&pidlock);

459     if ((prp = swapout_prp) != NULL) {
460         uint_t swrss = 0;
461         int swapped;

463         swapped = swapout(prp, &swrss, HARDSWAP);
464         if (swapped) {
465             /*
466              * If desperate, we want to give the space obtained
467              * by swapping this process out to processes in core,
468              * so we give them a chance by increasing deficit.
469              */
470             prp->p_swrss += swrss;
471             if (desperate)
472                 deficit += MIN(prp->p_swrss, lotsfree);
473             hardswap++;
474         }
475         mutex_exit(&swapout_prp->p_lock);

477         if (swapped)
478             goto loop;
479     }

481     /*
482     * Delay for 1 second and look again later.
483     */
484     TRACE_3(TR_FAC_SCHED, TR_RUNIN,
485         "schedrunin:runin nswapped: %d, avefree: %ld freemem: %ld",
486         nswapped, avefree, freemem);

488 block:
489     t = curthread;
490     thread_lock(t);
491     runin++;
492     t->t_schedflag |= (TS_ALLSTART & ~TS_CSTART);
493     t->t_whystop = PR_SUSPENDED;
494     t->t_whatstop = SUSPEND_NORMAL;
495     (void) new_mstate(t, LMS_SLEEP);
496     mutex_enter(&swap_cpr_lock);
497     CALLB_CPR_SAFE_BEGIN(&cprinfo);
498     mutex_exit(&swap_cpr_lock);
499     thread_stop(t); /* change to stop state and drop lock */
500     switch();
501     mutex_enter(&swap_cpr_lock);
502     CALLB_CPR_SAFE_END(&cprinfo, &swap_cpr_lock);
503     mutex_exit(&swap_cpr_lock);
504     goto loop;
505 }

507 /*
508 * Remove the specified thread from the swap queue.
509 */
510 static void
511 swapdeg(kthread_id_t tp)
512 {
513     kthread_id_t *tpp;

515     ASSERT(THREAD_LOCK_HELD(tp));
516     ASSERT(tp->t_schedflag & TS_ON_SWAPQ);

518     tpp = &tswap_queue;
519     for (;;) {
520         ASSERT(*tpp != NULL);

```



```

521         if (*tpp == tp)
522             break;
523         tpp = &(*tpp)->t_link;
524     }
525     *tpp = tp->t_link;
526     tp->t_schedflag &= ~TS_ON_SWAPQ;
527 }

529 /*
530 * Swap in lwps. Returns nonzero on success (i.e., if at least one lwp is
531 * swapped in) and 0 on failure.
532 */
533 static int
534 swapin(proc_t *pp)
535 {
536     kthread_id_t tp;
537     int err;
538     int num_swapped_in = 0;
539     struct cpu *cpup = CPU;
540     pri_t thread_pri;

542     ASSERT(MUTEX_HELD(&pp->p_lock));
543     ASSERT(pp->p_swpcnt);

545 top:
546     tp = pp->p_tlist;
547     do {
548         /*
549          * Only swapin eligible lwps (specified by the scheduling
550          * class) which are unloaded and ready to run.
551          */
552         thread_lock(tp);
553         thread_pri = CL_SWAPIN(tp, 0);
554         if (thread_pri != -1 && tp->t_state == TS_RUN &&
555             (tp->t_schedflag & TS_LOAD) == 0) {
556             size_t stack_size;
557             pgcnt_t stack_pages;

559             ASSERT((tp->t_schedflag & TS_ON_SWAPQ) == 0);

561             thread_unlock(tp);
562             /*
563              * Now drop the p_lock since the stack needs
564              * to be brought in.
565              */
566             mutex_exit(&pp->p_lock);

568             stack_size = swappsize(tp->t_swap);
569             stack_pages = btopr(stack_size);
570             /* Kernel probe */
571             TNF_PROBE_4(swapin_lwp, "vm swap swapin", /* CSTYLED */
572                 tnf_pid, pid, pp->p_pid,
573                 tnf_lwpid, lwpid, tp->t_tid,
574                 tnf_kthread_id, tid, tp,
575                 tnf_ulong, page_count, stack_pages);

577             rw_enter(&kas.a_lock, RW_READER);
578             err = segkp_fault(segkp->s_as->a_hat, segkp,
579                 tp->t_swap, stack_size, F_SOFTLOCK, S_OTHER);
580             rw_exit(&kas.a_lock);

582             /*
583              * Re-acquire the p_lock.
584              */
585             mutex_enter(&pp->p_lock);
586             if (err) {

```

```

587         num_swapped_in = 0;
588         break;
589     } else {
590 #ifdef __sparc
591         lwp_swapin(tp);
592 #endif /* __sparc */
593         CPU_STATS_ADDQ(cpup, vm, swapin, 1);
594         CPU_STATS_ADDQ(cpup, vm, pgswwapin,
595             stack_pages);

597         pp->p_swpcnt--;
598         pp->p_swrss -= stack_pages;

600         thread_lock(tp);
601         tp->t_schedflag |= TS_LOAD;
602         dq_sruninc(tp);

604         /* set swapin time */
605         tp->t_stime = ddi_get_lbolt();
606         thread_unlock(tp);

608         nswapped--;
609         tot_swapped_in++;
610         num_swapped_in++;

612         TRACE_2(TR_FAC_SCHED, TR_SWAPIN,
613             "swapin: pp %p stack_pages %lu",
614             pp, stack_pages);
615         goto top;
616     }
617 }
618 thread_unlock(tp);
619 } while ((tp = tp->t_forw) != pp->p_tlist);
620 return (num_swapped_in);
621 }

623 /*
624 * Swap out lwps. Returns nonzero on success (i.e., if at least one lwp is
625 * swapped out) and 0 on failure.
626 */
627 static int
628 swapout(proc_t *pp, uint_t *swrss, int swapflags)
629 {
630     kthread_id_t tp;
631     pgcnt_t ws_pages = 0;
632     int err;
633     int swapped_lwps = 0;
634     struct as *as = pp->p_as;
635     struct cpu *cpup = CPU;
636     pri_t thread_pri;

638     ASSERT(MUTEX_HELD(&pp->p_lock));

640     if (pp->p_flag & SEXITING)
641         return (0);

643 top:
644     tp = pp->p_tlist;
645     do {
646         klwp_t *lwp = ttolwp(tp);

648         /*
649          * Swapout eligible lwps (specified by the scheduling
650          * class) which don't have TS_DONT_SWAP set. Set the
651          * "intent to swap" flag (TS_SWAPENQ) on threads
652          * which have TS_DONT_SWAP set so that they can be

```

```

653     * swapped if and when they reach a safe point.
654     */
655     thread_lock(tp);
656     thread_pri = CL_SWAPOUT(tp, swapflags);
657     if (thread_pri != -1) {
658         if (tp->t_schedflag & TS_DONT_SWAP) {
659             tp->t_schedflag |= TS_SWAPENQ;
660             tp->t_trapret = 1;
661             aston(tp);
662         } else {
663             pgcnt_t stack_pages;
664             size_t stack_size;
665
666             ASSERT((tp->t_schedflag &
667                 (TS_DONT_SWAP | TS_LOAD)) == TS_LOAD);
668
669             if (lock_try(&tp->t_lock)) {
670                 /*
671                  * Remove thread from the swap_queue.
672                  */
673                 if (tp->t_schedflag & TS_ON_SWAPQ) {
674                     ASSERT(!(tp->t_schedflag &
675                         TS_SWAPENQ));
676                     swapdeq(tp);
677                 } else if (tp->t_state == TS_RUN)
678                     dq_srundec(tp);
679
680                 tp->t_schedflag &=
681                     ~(TS_LOAD | TS_SWAPENQ);
682                 lock_clear(&tp->t_lock);
683
684                 /*
685                  * Set swapout time if the thread isn't
686                  * sleeping.
687                  */
688                 if (tp->t_state != TS_SLEEP)
689                     tp->t_stime = ddi_get_lbolt();
690                 thread_unlock(tp);
691
692                 nswapped++;
693                 tot_swapped_out++;
694
695                 lwp->lwp_ru.nswap++;
696
697                 /*
698                  * Now drop the p_lock since the
699                  * stack needs to be pushed out.
700                  */
701                 mutex_exit(&pp->p_lock);
702
703                 stack_size = swappsize(tp->t_swap);
704                 stack_pages = btopr(stack_size);
705                 ws_pages += stack_pages;
706                 /* Kernel probe */
707                 TNF_PROBE_4(swapout_lwp,
708                     "vm swap swapout",
709                     /* CSTYLEL */
710                     tnf_pid, pid, pp->p_pid,
711                     tnf_lwpid, lwpid, tp->t_tid,
712                     tnf_kthread_id, tid, tp,
713                     tnf_ulong, page_count,
714                     stack_pages);
715
716                 rw_enter(&kas.a_lock, RW_READER);
717                 err = segkp_fault(segkp->s_as->a_hat,
718                     segkp, tp->t_swap, stack_size,

```

```

719                 F_SOFTUNLOCK, S_WRITE);
720                 rw_exit(&kas.a_lock);
721
722                 if (err) {
723                     cmn_err(CE_PANIC,
724                         "swapout: segkp_fault "
725                         "failed err: %d", err);
726                 }
727                 CPU_STATS_ADDQ(cpup,
728                     vm, pgswapout, stack_pages);
729
730                 mutex_enter(&pp->p_lock);
731                 pp->p_swapcnt++;
732                 swapped_lwps++;
733                 goto top;
734             }
735         }
736     }
737     thread_unlock(tp);
738 } while ((tp = tp->t_forw) != pp->p_tlist);
739
740 /*
741  * Unload address space when all lwps are swapped out.
742  */
743 if (pp->p_swapcnt == pp->p_lwpcnt) {
744     size_t as_size = 0;
745
746     /*
747      * Avoid invoking as_swapout() if the process has
748      * no MMU resources since pageout will eventually
749      * steal pages belonging to this address space. This
750      * saves CPU cycles as the number of pages that are
751      * potentially freed or pushed out by the segment
752      * swapout operation is very small.
753      */
754     if (rm_asrss(pp->p_as) != 0)
755         as_size = as_swapout(as);
756
757     CPU_STATS_ADDQ(cpup, vm, pgswapout, btop(as_size));
758     CPU_STATS_ADDQ(cpup, vm, swapout, 1);
759     ws_pages += btop(as_size);
760
761     TRACE_2(TR_FAC_SCHED, TR_SWAPOUT,
762         "swapout: pp %p pages_pushed %lu", pp, ws_pages);
763     /* Kernel probe */
764     TNF_PROBE_2(swapout_process, "vm swap swapout", /* CSTYLEL */,
765         tnf_pid, pid, pp->p_pid,
766         tnf_ulong, page_count, ws_pages);
767 }
768 *swrss = ws_pages;
769 return (swapped_lwps);
770 }
771
772 void
773 swapout_lwp(klwp_t *lwp)
774 {
775     kthread_id_t tp = curthread;
776
777     ASSERT(curthread == lwptot(lwp));
778
779     /*
780      * Don't insert the thread onto the swap queue if
781      * sufficient memory is available.
782      */
783     if (avefree > desfree || avefree < desfree && freemem > desfree) {
784         thread_lock(tp);

```

```

785         tp->t_schedflag &= ~TS_SWAPENQ;
786         thread_unlock(tp);
787         return;
788     }

790     /*
791     * Lock the thread, then move it to the swapped queue from the
792     * onproc queue and set its state to be TS_RUN.
793     */
794     thread_lock(tp);
795     ASSERT(tp->t_state == TS_ONPROC);
796     if (tp->t_schedflag & TS_SWAPENQ) {
797         tp->t_schedflag &= ~TS_SWAPENQ;

799         /*
800         * Set the state of this thread to be runnable
801         * and move it from the onproc queue to the swap queue.
802         */
803         disp_swapped_enq(tp);

805         /*
806         * Insert the thread onto the swap queue.
807         */
808         tp->t_link = tswap_queue;
809         tswap_queue = tp;
810         tp->t_schedflag |= TS_ON_SWAPQ;

812         thread_unlock_nopreempt(tp);

814         TRACE_1(TR_FAC_SCHED, TR_SWAPOUT_LWP, "swapout_lwp:%x", lwp);

816         swtch());
817     } else {
818         thread_unlock(tp);
819     }
820 }

822 /*
823 * Swap all threads on the swap queue.
824 */
825 static void
826 process_swap_queue(void)
827 {
828     kthread_id_t tp;
829     uint_t ws_pages;
830     proc_t *pp;
831     struct cpu *cpup = CPU;
832     klwp_t *lwp;
833     int err;

835     if (tswap_queue == NULL)
836         return;

838     /*
839     * Acquire the "swapped_lock" which locks the swap queue,
840     * and unload the stacks of all threads on it.
841     */
842     disp_lock_enter(&swapped_lock);
843     while ((tp = tswap_queue) != NULL) {
844         pgcnt_t stack_pages;
845         size_t stack_size;

847         tswap_queue = tp->t_link;
848         tp->t_link = NULL;

850         /*

```

```

851         * Drop the "dispatcher lock" before acquiring "t_lock"
852         * to avoid spinning on it since the thread at the front
853         * of the swap queue could be pinned before giving up
854         * its "t_lock" in resume.
855         */
856         disp_lock_exit(&swapped_lock);
857         lock_set(&tp->t_lock);

859         /*
860         * Now, re-acquire the "swapped_lock". Acquiring this lock
861         * results in locking the thread since its dispatcher lock
862         * (t_lockp) is the "swapped_lock".
863         */
864         disp_lock_enter(&swapped_lock);
865         ASSERT(tp->t_state == TS_RUN);
866         ASSERT(tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ));

868         tp->t_schedflag &= ~(TS_LOAD | TS_ON_SWAPQ);
869         tp->t_stime = ddi_get_lbolt(); /* swapout time */
870         disp_lock_exit(&swapped_lock);
871         lock_clear(&tp->t_lock);

873         lwp = ttolwp(tp);
874         lwp->lwp_ru.nswap++;

876         pp = ttoproc(tp);
877         stack_size = swappsize(tp->t_swap);
878         stack_pages = btopr(stack_size);

880         /* Kernel probe */
881         TNF_PROBE_4(swapout_lwp, "vm swap swapout", /* CSTYLED */,
882                 tnf_pid, pid, pp->p_pid,
883                 tnf_lwpid, lwpid, tp->t_tid,
884                 tnf_kthread_id, tid, tp,
885                 tnf_ulong, page_count, stack_pages);

887         rw_enter(&kas.a_lock, RW_READER);
888         err = segkp_fault(segkp->s_as->a_hat, segkp, tp->t_swap,
889                 stack_size, F_SOFTUNLOCK, S_WRITE);
890         rw_exit(&kas.a_lock);

892         if (err) {
893             cmn_err(CE_PANIC,
894                 "process_swap_list: segkp_fault failed err: %d", err);
895         }
896         CPU_STATS_ADDQ(cpup, vm, pgswapout, stack_pages);

898         nswapped++;
899         tot_swapped_out++;
900         swapqswap++;

902         /*
903         * Don't need p_lock since the swapper is the only
904         * thread which increments/decrements p_swapcnt and p_swrss.
905         */
906         ws_pages = stack_pages;
907         pp->p_swapcnt++;

909         TRACE_1(TR_FAC_SCHED, TR_SWAPQ_LWP, "swaplist: pp %p", pp);

911         /*
912         * Unload address space when all lwps are swapped out.
913         */
914         if (pp->p_swapcnt == pp->p_lwpcnt) {
915             size_t as_size = 0;

```

```
917         if (rm_asrss(pp->p_as) != 0)
918             as_size = as_swapout(pp->p_as);
919
920         CPU_STATS_ADDQ(cpup, vm, pgswapout,
921             btop(as_size));
922         CPU_STATS_ADDQ(cpup, vm, swapout, 1);
923
924         ws_pages += btop(as_size);
925
926         TRACE_2(TR_FAC_SCHED, TR_SWAPOQ_PROC,
927             "swaplist_proc: pp %p pages_pushed: %lu",
928             pp, ws_pages);
929         /* Kernel probe */
930         TNF_PROBE_2(swapout_process, "vm swap swapout",
931             /* CSTYLELED */,
932             tnf_pid, pid, pp->p_pid,
933             tnf_ulong, page_count, ws_pages);
934     }
935     pp->p_swrss += ws_pages;
936     disp_lock_enter(&swapped_lock);
100 }
938 disp_lock_exit(&swapped_lock);
101 }
```

unchanged\_portion\_omitted

```

*****
39428 Thu Feb 18 08:40:40 2016
new/usr/src/uts/common/os/timers.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

622 /*
623  * Real time profiling interval timer expired:
624  * Increment microstate counters for each lwp in the process
625  * and ensure that running lwps are kicked into the kernel.
626  * If time is not set up to reload, then just return.
627  * Else compute next time timer should go off which is > current time,
628  * as above.
629  */
630 static void
631 realprofexpire(void *arg)
632 {
633     struct proc *p = arg;
634     kthread_t *t;

636     mutex_enter(&p->p_lock);
637     if (p->p_rprof_cyclic == CYCLIC_NONE ||
638         (t = p->p_tlist) == NULL) {
639         mutex_exit(&p->p_lock);
640         return;
641     }
642     do {
643         int mstate;

645         /*
646          * Attempt to allocate the SIGPROF buffer, but don't sleep.
647          */
648         if (t->t_rprof == NULL)
649             t->t_rprof = kmem_zalloc(sizeof (struct rprof),
650                                     KM_NOSLEEP);
651         if (t->t_rprof == NULL)
652             continue;

654         thread_lock(t);
655         switch (t->t_state) {
656         case TS_SLEEP:
657             /*
658              * Don't touch the lwp is it is swapped out.
659              */
660             if (!(t->t_schedflag & TS_LOAD)) {
661                 mstate = LMS_SLEEP;
662                 break;
663             }
657             switch (mstate = ttolwp(t)->lwp_mstate.ms_prev) {
658             case LMS_TFAULT:
659             case LMS_DFAULT:
660             case LMS_KFAULT:
661             case LMS_USER_LOCK:
662                 break;
663             default:
664                 mstate = LMS_SLEEP;
665                 break;
666             }
667             break;
668         case TS_RUN:
669         case TS_WAIT:
670             mstate = LMS_WAIT_CPU;
671             break;
672         case TS_ONPROC:
673             switch (mstate = t->t_mstate) {

```

```

674             case LMS_USER:
675             case LMS_SYSTEM:
676             case LMS_TRAP:
677                 break;
678             default:
679                 mstate = LMS_SYSTEM;
680                 break;
681             }
682             break;
683         default:
684             mstate = t->t_mstate;
685             break;
686         }
687         t->t_rprof->rp_anystate = 1;
688         t->t_rprof->rp_state[mstate]++;
689         aston(t);
690         /*
691          * force the thread into the kernel
692          * if it is not already there.
693          */
694         if (t->t_state == TS_ONPROC && t->t_cpu != CPU)
695             poke_cpu(t->t_cpu->cpu_id);
696         thread_unlock(t);
697     } while ((t = t->t_forw) != p->p_tlist);

699     mutex_exit(&p->p_lock);
700 }
_____unchanged_portion_omitted_____

```

\*\*\*\*\*

10025 Thu Feb 18 08:40:40 2016

new/usr/src/uts/common/os/waitq.c

6583 remove whole-process swapping

\*\*\*\*\*

unchanged\_portion\_omitted

```
197 /*
198  * Put specified thread to specified wait queue without dropping thread's lock.
199  * Returns 1 if thread was successfully placed on project's wait queue, or
200  * 0 if wait queue is blocked.
201  */
202 int
203 waitq_enqueue(waitq_t *wq, kthread_t *t)
204 {
205     ASSERT(THREAD_LOCK_HELD(t));
206     ASSERT(t->t_sleepq == NULL);
207     ASSERT(t->t_waitq == NULL);
208     ASSERT(t->t_link == NULL);
209
210     disp_lock_enter_high(&wq->wq_lock);
211
212     /*
213      * Can't enqueue anything on a blocked wait queue
214      */
215     if (wq->wq_blocked) {
216         disp_lock_exit_high(&wq->wq_lock);
217         return (0);
218     }
219
220     /*
221      * Mark the time when thread is placed on wait queue. The microstate
222      * accounting code uses this timestamp to determine wait times.
223      */
224     t->t_waitrq = gethrtime_unscaled();
225
226     /*
227      * Mark thread as not swappable. If necessary, it will get
228      * swapped out when it returns to the userland.
229      */
230     t->t_schedflag |= TS_DONT_SWAP;
231     DTRACE_SCHED1(cpucaps__sleep, kthread_t *, t);
232     waitq_link(wq, t);
233
234     THREAD_WAIT(t, &wq->wq_lock);
235     return (1);
236 }
237
238 unchanged_portion_omitted
```

```

*****
7493 Thu Feb 18 08:40:41 2016
new/usr/src/uts/common/sys/class.h
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

74 typedef struct thread_ops {
75     int      (*cl_enterclass)(kthread_t *, id_t, void *, cred_t *, void *);
76     void     (*cl_exitclass)(void *);
77     int      (*cl_canexit)(kthread_t *, cred_t *);
78     int      (*cl_fork)(kthread_t *, kthread_t *, void *);
79     void     (*cl_forkret)(kthread_t *, kthread_t *);
80     void     (*cl_parmsget)(kthread_t *, void *);
81     int      (*cl_parmsset)(kthread_t *, void *, id_t, cred_t *);
82     void     (*cl_stop)(kthread_t *, int, int);
83     void     (*cl_exit)(kthread_t *);
84     void     (*cl_active)(kthread_t *);
85     void     (*cl_inactive)(kthread_t *);
86     pri_t   (*cl_swapin)(kthread_t *, int);
87     pri_t   (*cl_swapout)(kthread_t *, int);
88     void     (*cl_trapret)(kthread_t *);
89     void     (*cl_preempt)(kthread_t *);
90     void     (*cl_setrun)(kthread_t *);
91     void     (*cl_sleep)(kthread_t *);
92     void     (*cl_tick)(kthread_t *);
93     void     (*cl_wakeup)(kthread_t *);
94     int      (*cl_donice)(kthread_t *, cred_t *, int, int *);
95     pri_t   (*cl_globpri)(kthread_t *);
96     void     (*cl_set_process_group)(pid_t, pid_t, pid_t);
97     void     (*cl_yield)(kthread_t *);
98     int      (*cl_doprio)(kthread_t *, cred_t *, int, int *);
99 } thread_ops_t;
_____unchanged_portion_omitted_____

113 #define STATIC_SCHED      (krwlock_t *)0xffffffff
114 #define LOADABLE_SCHED(s)  ((s)->cl_lock != STATIC_SCHED)
115 #define SCHED_INSTALLED(s) ((s)->cl_funcs != NULL)
116 #define ALLOCATED_SCHED(s) ((s)->cl_lock != NULL)

118 #if defined(_KERNEL) || defined(_FAKE_KERNEL)

120 #define CLASS_KERNEL(cid)  ((cid) == syscid || (cid) == sysdccid)

122 extern int      nclass; /* number of configured scheduling classes */
123 extern char    *defaultclass; /* default class for newproc'd processes */
124 extern struct sclass sclass[]; /* the class table */
125 extern kmutex_t class_lock; /* lock protecting class table */
126 extern int      loaded_classes; /* number of classes loaded */

128 extern pri_t   minclsypri;
129 extern id_t    syscid; /* system scheduling class ID */
130 extern id_t    sysdccid; /* system duty-cycle scheduling class ID */
131 extern id_t    defaultcid; /* "default" class id; see dispadmin(1M) */

133 extern int      alloc_cid(char *, id_t *);
134 extern int      scheduler_load(char *, sclass_t *);
135 extern int      getcid(char *, id_t *);
136 extern int      getcidbyname(char *, id_t *);
137 extern int      parmsin(pcparms_t *, pc_vaparms_t *);
138 extern int      parmsout(pcparms_t *, pc_vaparms_t *);
139 extern int      parmsset(pcparms_t *, kthread_t *);
140 extern void     parmsget(kthread_t *, pcparms_t *);
141 extern int      vaparmsout(char *, pcparms_t *, pc_vaparms_t *, uio_seg_t);

143 #endif

```

```

145 #define CL_ADMIN(clp, uaddr, reqpcredp) \
146     (*(clp)->cl_funcs->sclass.cl_admin)(uaddr, reqpcredp)

148 #define CL_ENTERCLASS(t, cid, clparmsp, credp, bufp) \
149     (sclass[cid].cl_funcs->thread.cl_enterclass)(t, cid, \
150     (void *)clparmsp, credp, bufp)

152 #define CL_EXITCLASS(cid, clprocp) \
153     (sclass[cid].cl_funcs->thread.cl_exitclass)((void *)clprocp)

155 #define CL_CANEXIT(t, cr)      (*(t)->t_clfuncs->cl_canexit)(t, cr)

157 #define CL_FORK(tp, ct, bufp)  (*(tp)->t_clfuncs->cl_fork)(tp, ct, bufp)

159 #define CL_FORKKRET(t, ct)    (*(t)->t_clfuncs->cl_forkret)(t, ct)

161 #define CL_GETCLINFO(clp, clinfop) \
162     (*(clp)->cl_funcs->sclass.cl_getclinfo)((void *)clinfop)

164 #define CL_GETCLPRI(clp, clprip) \
165     (*(clp)->cl_funcs->sclass.cl_getclpri)(clprip)

167 #define CL_PARMSGET(t, clparmsp) \
168     (*(t)->t_clfuncs->cl_parmsget)(t, (void *)clparmsp)

170 #define CL_PARMSIN(clp, clparmsp) \
171     (clp)->cl_funcs->sclass.cl_parmsin((void *)clparmsp)

173 #define CL_PARMSOUT(clp, clparmsp, vaparmsp) \
174     (clp)->cl_funcs->sclass.cl_parmsout((void *)clparmsp, vaparmsp)

176 #define CL_VAPARMSIN(clp, clparmsp, vaparmsp) \
177     (clp)->cl_funcs->sclass.cl_vaparmsin((void *)clparmsp, vaparmsp)

179 #define CL_VAPARMSOUT(clp, clparmsp, vaparmsp) \
180     (clp)->cl_funcs->sclass.cl_vaparmsout((void *)clparmsp, vaparmsp)

182 #define CL_PARMSSET(t, clparmsp, cid, curpcredp) \
183     (*(t)->t_clfuncs->cl_parmsset)(t, (void *)clparmsp, cid, curpcredp)

185 #define CL_PREEMPT(tp)        (*(tp)->t_clfuncs->cl_preempt)(tp)

187 #define CL_SETRUN(tp)         (*(tp)->t_clfuncs->cl_setrun)(tp)

189 #define CL_SLEEP(tp)          (*(tp)->t_clfuncs->cl_sleep)(tp)

191 #define CL_STOP(t, why, what) (*(t)->t_clfuncs->cl_stop)(t, why, what)

193 #define CL_EXIT(t)            (*(t)->t_clfuncs->cl_exit)(t)

195 #define CL_ACTIVE(t)          (*(t)->t_clfuncs->cl_active)(t)

197 #define CL_INACTIVE(t)        (*(t)->t_clfuncs->cl_inactive)(t)

201 #define CL_SWAPIN(t, flags)   (*(t)->t_clfuncs->cl_swapin)(t, flags)

203 #define CL_SWAPOUT(t, flags)  (*(t)->t_clfuncs->cl_swapout)(t, flags)

199 #define CL_TICK(t)            (*(t)->t_clfuncs->cl_tick)(t)

201 #define CL_TRAPRET(t)         (*(t)->t_clfuncs->cl_trapret)(t)

203 #define CL_WAKEUP(t)          (*(t)->t_clfuncs->cl_wakeup)(t)

205 #define CL_DONICE(t, cr, inc, ret) \

```

```
206      (*(t)->t_clfuncs->cl_donice)(t, cr, inc, ret)
208 #define CL_DOPRIO(t, cr, inc, ret) \
209      (*(t)->t_clfuncs->cl_doprio)(t, cr, inc, ret)
211 #define CL_GLOBPRI(t)          (*(t)->t_clfuncs->cl_globpri)(t)
213 #define CL_SET_PROCESS_GROUP(t, s, b, f) \
214      (*(t)->t_clfuncs->cl_set_process_group)(s, b, f)
216 #define CL_YIELD(tp)          (*(tp)->t_clfuncs->cl_yield)(tp)
218 #define CL_ALLOC(pp, cid, flag) \
219      (sclass[cid].cl_funcs->sclass.cl_alloc) (pp, flag)
221 #define CL_FREE(cid, bufp)     (sclass[cid].cl_funcs->sclass.cl_free) (bufp)
223 #ifdef __cplusplus
224 }
_____ unchanged portion omitted
```



```

*****
5872 Thu Feb 18 08:40:41 2016
new/usr/src/uts/common/sys/disp.h
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

82 #if defined(_KERNEL) || defined(_FAKE_KERNEL)

84 #define MAXCLSYSPRI    99
85 #define MINCLSYSPRI   60

88 /*
89  * Global scheduling variables.
90  *   - See sys/cpuvar.h for CPU-local variables.
91  */
92 extern int      nswapped;      /* number of swapped threads */
93                /* nswapped protected by swap_lock */

95 extern pri_t   minclsyspri;    /* minimum level of any system class */
96 extern pri_t   maxclsyspri;    /* maximum level of any system class */
97 extern pri_t   intr_pri;      /* interrupt thread priority base level */

99 #endif /* _KERNEL || _FAKE_KERNEL */
100 #if defined(_KERNEL)

102 /*
103  * Minimum amount of time that a thread can remain runnable before it can
104  * be stolen by another CPU (in nanoseconds).
105  */
106 extern hrtime_t nosteal_nsec;

108 /*
109  * Kernel preemption occurs if a higher-priority thread is runnable with
110  * a priority at or above kpreemptpri.
111  */
112  * So that other processors can watch for such threads, a separate
113  * dispatch queue with unbound work above kpreemptpri is maintained.
114  * This is part of the CPU partition structure (cpupart_t).
115  */
116 extern pri_t   kpreemptpri;    /* level above which preemption takes place */

118 extern void    disp_kp_alloc(disp_t *, pri_t); /* allocate kp queue */
119 extern void    disp_kp_free(disp_t *);        /* free kp queue */

121 /*
122  * Macro for use by scheduling classes to decide whether the thread is about
123  * to be scheduled or not. This returns the maximum run priority.
124  */
125 #define DISP_MAXRUNPRI(t)      ((t)->t_disp_queue->disp_maxrunpri)

127 /*
128  * Platform callbacks for various dispatcher operations
129  */
130  * idle_cpu() is invoked when a cpu goes idle, and has nothing to do.
131  * disp_eng_thread() is invoked when a thread is placed on a run queue.
132  */
133 extern void    (*idle_cpu)();
134 extern void    (*disp_eng_thread)(struct cpu *, int);

137 extern int    dispdeq(kthread_t *);
138 extern void    dispinit(void);
139 extern void    disp_add(sclass_t *);
140 extern int    intr_active(struct cpu *, int);

```

```

141 extern int    servicing_interrupt(void);
142 extern void    preempt(void);
143 extern void    setbackdq(kthread_t *);
144 extern void    setfrontdq(kthread_t *);
145 extern void    swtch(void);
146 extern void    swtch_to(kthread_t *);
147 extern void    swtch_from_zombie(void);
148                __NORETURN;
149 extern void    dq_srunic(kthread_t *);
150 extern void    dq_srundec(kthread_t *);
151 extern void    cpu_rechoose(kthread_t *);
152 extern void    cpu_surrender(kthread_t *);
153                kpreempt(int);
154 extern struct cpu *disp_lowpri_cpu(struct cpu *, struct lgrp_ld *, pri_t,
155                struct cpu *);
156 extern int    disp_bound_threads(struct cpu *, int);
157 extern int    disp_bound_anythreads(struct cpu *, int);
158 extern int    disp_bound_partition(struct cpu *, int);
159 extern void    disp_cpu_init(struct cpu *);
160 extern void    disp_cpu_fini(struct cpu *);
161 extern void    disp_cpu_inactive(struct cpu *);
162 extern void    disp_adjust_unbound_pri(kthread_t *);
163 extern void    resume(kthread_t *);
164                resume_from_intr(kthread_t *);
165                resume_from_zombie(kthread_t *);
166                __NORETURN;
167                disp_swapped_eng(kthread_t *);
168                disp_anywork(void);

168 #define KPREEMPT_SYNC      (-1)
169 #define kpreempt_disable() \
170     { \
171         curthread->t_preempt++; \
172         ASSERT(curthread->t_preempt >= 1); \
173     } \
174 #define kpreempt_enable() \
175     { \
176         ASSERT(curthread->t_preempt >= 1); \
177         if (--curthread->t_preempt == 0 && \
178             CPU->cpu_kprunrun) \
179             kpreempt(KPREEMPT_SYNC); \
180     }

182 #endif /* _KERNEL */

184 #ifdef __cplusplus
185 }
_____unchanged_portion_omitted_____

```

```

*****
29368 Thu Feb 18 08:40:41 2016
new/usr/src/uts/common/sys/proc.h
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

125 struct pool;
126 struct task;
127 struct zone;
128 struct brand;
129 struct corectl_path;
130 struct corectl_content;

132 /*
133  * One structure allocated per active process. Per-process data (user.h) is
134  * also inside the proc structure.
135  * One structure allocated per active process. It contains all
136  * data needed about the process while the process may be swapped
137  * out. Other per-process data (user.h) is also inside the proc structure.
138  * Lightweight-process data (lwp.h) and the kernel stack may be swapped out.
139  */
140 typedef struct proc {
141     /*
142      * Fields requiring no explicit locking
143      */
144     struct vnode *p_exec;          /* pointer to a.out vnode */
145     struct as *p_as;              /* process address space pointer */
146     struct plock *p_lockp;        /* ptr to proc struct's mutex lock */
147     kmutex_t p_crlock;           /* lock for p_cred */
148     struct cred *p_cred;          /* process credentials */
149     /*
150      * Fields protected by pidlock
151      */
152     int p_swapcnt;               /* number of swapped out lwps */
153     char p_stat;                 /* status of process */
154     char p_wcode;                /* current wait code */
155     ushort_t p_pidflag;          /* flags protected only by pidlock */
156     int p_wdata;                 /* current wait return value */
157     pid_t p_ppid;                /* process id of parent */
158     struct proc *p_link;         /* forward link */
159     struct proc *p_parent;       /* ptr to parent process */
160     struct proc *p_child;        /* ptr to first child process */
161     struct proc *p_sibling;      /* ptr to next sibling proc on chain */
162     struct proc *p_psibling;     /* ptr to prev sibling proc on chain */
163     struct proc *p_sibling_ns;   /* prt to siblings with new state */
164     struct proc *p_child_ns;     /* prt to children with new state */
165     struct proc *p_next;         /* active chain link next */
166     struct proc *p_prev;         /* active chain link prev */
167     struct proc *p_nextofkin;    /* gets accounting info at exit */
168     struct proc *p_orphan;
169     struct proc *p_nextorph;
170     struct proc *p_pglink;       /* process group hash chain link next */
171     struct proc *p_ppglink;      /* process group hash chain link prev */
172     struct sess *p_sessp;        /* session information */
173     struct pid *p_pidp;         /* process ID info */
174     struct pid *p_pgidp;        /* process group ID info */
175     /*
176      * Fields protected by p_lock
177      */
178     kcondvar_t p_cv;             /* proc struct's condition variable */
179     kcondvar_t p_flag_cv;
180     kcondvar_t p_lwpexit;        /* waiting for some lwp to exit */
181     kcondvar_t p_holdlwps;       /* process is waiting for its lwps */
182     /*
183      * to be held.
184      */
185     uint_t p_proc_flag;         /* /proc-related flags */

```

```

179     uint_t p_flag;              /* protected while set. */
180     /*
181      * flags defined below
182      */
183     clock_t p_utime;            /* user time, this process */
184     clock_t p_stime;            /* system time, this process */
185     clock_t p_cutime;           /* sum of children's user time */
186     clock_t p_cstime;           /* sum of children's system time */
187     avl_tree_t *p_segacct;      /* System V shared segment list */
188     avl_tree_t *p_semacct;      /* System V semaphore undo list */
189     caddr_t p_bssbase;          /* base addr of last bss below heap */
190     caddr_t p_brkbase;          /* base addr of heap */
191     size_t p_brksize;           /* heap size in bytes */
192     uint_t p_brkpageszc;        /* preferred heap max page size code */
193     /*
194      * Per process signal stuff.
195      */
196     k_sigset_t p_sig;           /* signals pending to this process */
197     k_sigset_t p_extsig;        /* signals sent from another contract */
198     k_sigset_t p_ignore;        /* ignore when generated */
199     k_sigset_t p_siginfo;       /* gets signal info with signal */
200     void *p_sigfd;              /* signalfd support state */
201     struct sigqueue *p_sigqueue; /* queued siginfo structures */
202     struct sigqhdr *p_sigqhdr;  /* hdr to sigqueue structure pool */
203     struct sigqhdr *p_sighdr;   /* hdr to signotify structure pool */
204     uchar_t p_stopsig;          /* jobcontrol stop signal */
205     /*
206      * Special per-process flag when set will fix misaligned memory
207      * references.
208      */
209     char p_fixalignment;
210     /*
211      * Per process lwp and kernel thread stuff
212      */
213     id_t p_lwpid;               /* most recently allocated lwpid */
214     int p_lwpcnt;               /* number of lwps in this process */
215     int p_lwprcnt;              /* number of not stopped lwps */
216     int p_lwpdaemon;            /* number of TP_DAEMON lwps */
217     int p_lwpwait;              /* number of lwps in lwp_wait() */
218     int p_lwpdwait;             /* number of daemons in lwp_wait() */
219     int p_zombcnt;              /* number of zombie lwps */
220     kthread_t *p_tlist;         /* circular list of threads */
221     lwpdir_t *p_lwpdir;         /* thread (lwp) directory */
222     lwpdir_t *p_lwplib;         /* p_lwpdir free list */
223     tidhash_t *p_tidhash;       /* tid (lwpid) lookup hash table */
224     uint_t p_lwpdir_sz;         /* number of p_lwpdir[] entries */
225     uint_t p_tidhash_sz;        /* number of p_tidhash[] entries */
226     ret_tidhash_t *p_ret_tidhash; /* retired tidhash hash tables */
227     uint64_t p_lgrpset;         /* unprotected hint of set of lgrps */
228     /*
229      * on which process has threads
230      */
231     volatile lgrp_id_t p_tl_lgrp; /* main's thread lgroup id */
232     volatile lgrp_id_t p_tr_lgrp; /* text replica's lgroup id */
233 #if defined(LP64)
234     uintptr_t p_lgrpres2;        /* reserved for lgrp migration */
235 #endif
236     /*
237      * /proc (process filesystem) debugger interface stuff.
238      */
239     k_sigset_t p_sigmask;        /* mask of traced signals (/proc) */
240     k_fltset_t p_fltmask;        /* mask of traced faults (/proc) */
241     struct vnode *p_trace;       /* pointer to primary /proc vnode */
242     struct vnode *p_plist;       /* list of /proc vnodes for process */
243     kthread_t *p_agenttp;        /* thread ptr for /proc agent lwp */
244     avl_tree_t p_warea;          /* list of watched areas */
245     avl_tree_t p_wpage;         /* remembered watched pages (vfork) */
246     watched_page_t *p_wprot;     /* pages that need to have prot set */

```

```

245 int p_mapcnt; /* number of active pr_mappage()s */
246 kmutex_t p_maplock; /* lock for pr_mappage() */
247 struct proc *p_rlink; /* linked list for server */
248 kcondvar_t p_srwchan_cv;
249 size_t p_stksize; /* process stack size in bytes */
250 uint_t p_stkpagesz; /* preferred stack max page size code */

252 /*
253 * Microstate accounting, resource usage, and real-time profiling
254 */
255 hrtime_t p_mstart; /* hi-res process start time */
256 hrtime_t p_mterm; /* hi-res process termination time */
257 hrtime_t p_mlreal; /* elapsed time sum over defunct lwps */
258 hrtime_t p_acct[NMSTATES]; /* microstate sum over defunct lwps */
259 hrtime_t p_cacct[NMSTATES]; /* microstate sum over child procs */
260 struct lrusage p_ru; /* lrusage sum over defunct lwps */
261 struct lrusage p_cru; /* lrusage sum over child procs */
262 struct itimerval p_rprof_timer; /* ITIMER_REALPROF interval timer */
263 uintptr_t p_rprof_cyclic; /* ITIMER_REALPROF cyclic */
264 uint_t p_defunct; /* number of defunct lwps */
265 /*
266 * profiling. A lock is used in the event of multiple lwp's
267 * using the same profiling base/size.
268 */
269 kmutex_t p_plock; /* protects user profile arguments */
270 struct prof p_prof; /* profile arguments */

272 /*
273 * Doors.
274 */
275 door_pool_t p_server_threads; /* common thread pool */
276 struct door_node *p_door_list; /* active doors */
277 struct door_node *p_unref_list;
278 kcondvar_t p_unref_cv;
279 char p_unref_thread; /* unref thread created */

281 /*
282 * Kernel probes
283 */
284 uchar_t p_tnf_flags;

286 /*
287 * Solaris Audit
288 */
289 struct p_audit_data *p_audit_data; /* per process audit structure */

291 pctxop_t *p_pctx;

293 #if defined(__x86)
294 /*
295 * LDT support.
296 */
297 kmutex_t p_ldtlock; /* protects the following fields */
298 user_desc_t *p_ldt; /* Pointer to private LDT */
299 system_desc_t p_ldt_desc; /* segment descriptor for private LDT */
300 ushort_t p_ldtlimit; /* highest selector used */
301 #endif
302 size_t p_swrss; /* resident set size before last swap */
303 struct aio *p_aio; /* pointer to async I/O struct */
304 struct itimer *p_itimer; /* interval timers */
305 timeout_id_t p_alarmid; /* alarm's timeout id */
306 caddr_t p_usrstack; /* top of the process stack */
307 uint_t p_stkprot; /* stack memory protection */
308 uint_t p_datprot; /* data memory protection */
309 model_t p_model; /* data model determined at exec time */
310 struct lwpchan_data *p_lcp; /* lwpchan cache */

```

```

311 kmutex_t p_lcp_lock; /* protects assignments to p_lcp */
312 utrap_handler_t *p_utraps; /* pointer to user trap handlers */
313 struct corectl_path *p_corefile; /* pattern for core file */
314 struct task *p_task; /* our containing task */
315 struct proc *p_taskprev; /* ptr to previous process in task */
316 struct proc *p_tasknext; /* ptr to next process in task */
317 kmutex_t p_sc_lock; /* protects p_pagep */
318 struct sc_page_ctl *p_pagep; /* list of process's shared pages */
319 struct rctl_set *p_rctls; /* resource controls for this process */
320 rlim64_t p_stk_ctl; /* currently enforced stack size */
321 rlim64_t p_fsz_ctl; /* currently enforced file size */
322 rlim64_t p_vmem_ctl; /* currently enforced addr-space size */
323 rlim64_t p_fno_ctl; /* currently enforced file-desc limit */
324 pid_t p_ancpid; /* ancestor pid, used by exacct */
325 struct itimerval p_realitimer; /* real interval timer */
326 timeout_id_t p_itimerid; /* real interval timer's timeout id */
327 struct corectl_content *p_content; /* content of core file */

329 avl_tree_t p_ct_held; /* held contracts */
330 struct ct_equeue **p_ct_equeue; /* process-type event queues */

332 struct cont_process *p_ct_process; /* process contract */
333 list_node_t p_ct_member; /* process contract membership */
334 sigqueue_t *p_killsp; /* sigqueue pointer for SIGKILL */

336 int p_dtrace_probes; /* are there probes for this proc? */
337 uint64_t p_dtrace_count; /* number of DTrace tracepoints */
338 /* (protected by P_PR_LOCK) */
339 void *p_dtrace_helpers; /* DTrace helpers, if any */
340 struct pool *p_pool; /* pointer to containing pool */
341 kcondvar_t p_poolcv; /* synchronization with pools */
342 uint_t p_poolcnt; /* # threads inside pool barrier */
343 uint_t p_poolflag; /* pool-related flags (see below) */
344 uintptr_t p_portcnt; /* event ports counter */
345 struct zone *p_zone; /* zone in which process lives */
346 struct vnode *p_execdir; /* directory that p_exec came from */
347 struct brand *p_brand; /* process's brand */
348 void *p_brand_data; /* per-process brand state */

350 /* additional lock to protect p_sessp (but not its contents) */
351 kmutex_t p_splck;
352 rctl_qty_t p_locked_mem; /* locked memory charged to proc */
353 /* protected by p_lock */
354 rctl_qty_t p_crypto_mem; /* /dev/crypto memory charged to proc */
355 /* protected by p_lock */
356 clock_t p_pttime; /* buffered task time */

358 /*
359 * The user structure
360 */
361 struct user p_user; /* (see sys/user.h) */
362 } proc_t;
    unchanged portion omitted

```

\*\*\*\*\*

11325 Thu Feb 18 08:40:41 2016

new/usr/src/uts/common/sys/sysinfo.h

6583 remove whole-process swapping

\*\*\*\*\*

unchanged portion omitted

```
248 typedef struct cpu_vm_stats {
249     uint64_t pgrec;                /* page reclaims (includes pageout) */
250     uint64_t pgfrec;              /* page reclaims from free list */
251     uint64_t pgin;                /* pageins */
252     uint64_t pgpgin;              /* pages paged in */
253     uint64_t pgout;               /* pageouts */
254     uint64_t pgpgout;             /* pages paged out */
255     uint64_t swapin;              /* swapins */
256     uint64_t pgswpin;             /* pages swapped in */
257     uint64_t swapout;             /* swapouts */
258     uint64_t pgswpout;            /* pages swapped out */
259     uint64_t zfod;                /* pages zero filled on demand */
260     uint64_t dfree;               /* pages freed by daemon or auto */
261     uint64_t scan;                /* pages examined by pageout daemon */
262     uint64_t rev;                 /* revolutions of page daemon hand */
263     uint64_t hat_fault;           /* minor page faults via hat_fault() */
264     uint64_t as_fault;            /* minor page faults via as_fault() */
265     uint64_t maj_fault;           /* major page faults */
266     uint64_t cow_fault;           /* copy-on-write faults */
267     uint64_t prot_fault;          /* protection faults */
268     uint64_t softlock;            /* faults due to software locking req */
269     uint64_t kernel_asflt;        /* as_fault()s in kernel addr space */
270     uint64_t pgrun;               /* times pager scheduled */
271     uint64_t execpgin;            /* executable pages paged in */
272     uint64_t execpgout;           /* executable pages paged out */
273     uint64_t execfree;            /* executable pages freed */
274     uint64_t anonpgin;            /* anon pages paged in */
275     uint64_t anonpgout;           /* anon pages paged out */
276     uint64_t anonfree;           /* anon pages freed */
277     uint64_t fspgin;              /* fs pages paged in */
278     uint64_t fspgout;            /* fs pages paged out */
279     uint64_t fsfree;              /* fs pages free */
280 } cpu_vm_stats_t;
```

unchanged portion omitted

```

*****
15545 Thu Feb 18 08:40:41 2016
new/usr/src/uts/common/sys/system.h
6583 remove whole-process swapping
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
22 /*      All Rights Reserved      */

25 /*
26  * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
27  * Use is subject to license terms.
28  */

30 #ifndef _SYS_SYSTEM_H
31 #define _SYS_SYSTEM_H

33 #include <sys/types.h>
34 #include <sys/t_lock.h>
35 #include <sys/proc.h>
36 #include <sys/dditypes.h>

38 #ifdef __cplusplus
39 extern "C" {
40 #endif

42 /*
43  * The pc_t is the type of the kernel's program counter.  In general, a
44  * pc_t is a uintptr_t -- except for a sparcv9 kernel, in which case all
45  * instruction text is below 4G, and a pc_t is thus a uint32_t.
46  */
47 #ifdef __sparcv9
48 typedef uint32_t pc_t;
49 #else
50 typedef uintptr_t pc_t;
51 #endif

53 /*
54  * Random set of variables used by more than one routine.
55  */

57 #if defined(_KERNEL) || defined(_FAKE_KERNEL)
58 #include <sys/types32.h>
59 #include <sys/varargs.h>
60 #include <sys/uadmin.h>

```

```

62 extern int hz; /* system clock rate */
63 extern struct vnode *rootdir; /* pointer to vnode of root directory */
64 extern struct vnode *devicesdir; /* pointer to /devices vnode */
65 extern int interrupts_unleashed; /* set after the spl0() in main() */

67 extern char runin; /* scheduling flag */
68 extern char runout; /* scheduling flag */
69 extern char wake_sched; /* causes clock to wake swapper on next tick */
70 extern char wake_sched_sec; /* causes clock to wake swapper after a sec */

67 extern pgcnt_t maxmem; /* max available memory (pages) */
68 extern pgcnt_t physmem; /* physical memory (pages) on this CPU */
69 extern pfn_t physmax; /* highest numbered physical page present */
70 extern pgcnt_t physinstalled; /* physical pages including PROM/boot use */

72 extern caddr_t s_text; /* start of kernel text segment */
73 extern caddr_t e_text; /* end of kernel text segment */
74 extern caddr_t s_data; /* start of kernel text segment */
75 extern caddr_t e_data; /* end of kernel text segment */

77 extern pgcnt_t availrmem; /* Available resident (not swapable) */
78 /* memory in pages. */
79 extern pgcnt_t availrmem_initial; /* initial value of availrmem */
80 extern pgcnt_t segspt_minfree; /* low water mark for availrmem in seg_spt */
81 extern pgcnt_t freemem; /* Current free memory. */

83 extern dev_t rootdev; /* device of the root */
84 extern struct vnode *rootvp; /* vnode of root device */
85 extern boolean_t root_is_svm; /* root is a mirrored device flag */
86 extern boolean_t root_is_ramdisk; /* root is boot_archive ramdisk */
87 extern uint32_t ramdisk_size; /* (KB) set only for sparc netboots */
88 extern char *volatile_panicstr; /* panic string pointer */
89 extern va_list panicargs; /* panic arguments */
90 extern volatile int quiesce_active; /* quiesce(9E) is in progress */

92 extern int rstchown; /* 1 ==> restrictive chown(2) semantics */
93 extern int klustsize;

95 extern int abort_enable; /* Platform input-device abort policy */

97 extern int audit_active; /* Solaris Auditing module state */

99 extern int avenrun[]; /* array of load averages */

101 extern char *isa_list; /* For sysinfo's isalist option */

103 extern int noexec_user_stack; /* patchable via /etc/system */
104 extern int noexec_user_stack_log; /* patchable via /etc/system */

106 /*
107  * Use NFS client operations in the global zone only. Under contract with
108  * admin/install; do not change without coordinating with that consolidation.
109  */
110 extern int nfs_global_client_only;

112 extern void report_stack_exec(proc_t *, caddr_t);

114 extern void startup(void);
115 extern void clkstart(void);
116 extern void post_startup(void);
117 extern void kern_setup1(void);
118 extern void ka_init(void);
119 extern void nodename_set(void);

121 /*
122  * for tod fault detection

```

new/usr/src/uts/common/sys/system.h

3

```
123 */
124 enum tod_fault_type {
125     TOD_REVERSED = 0,
126     TOD_STALLED,
127     TOD_JUMPED,
128     TOD_RATECHANGED,
129     TOD_RDONLY,
130     TOD_NOFAULT
131 };
_____unchanged_portion_omitted_____
```

```

*****
26129 Thu Feb 18 08:40:41 2016
new/usr/src/uts/common/sys/thread.h
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

 96 typedef struct _kthread *kthread_id_t;

 98 struct turnstile;
 99 struct panic_trap_info;
100 struct upimutex;
101 struct kproject;
102 struct on_trap_data;
103 struct waitq;
104 struct _kcpc_ctx;
105 struct _kcpc_set;

107 /* Definition for kernel thread identifier type */
108 typedef uint64_t kt_did_t;

110 typedef struct _kthread {
111     struct _kthread *t_link; /* dispq, sleepq, and free queue link */

113     caddr_t t_stk; /* base of stack (kernel sp value to use) */
114     void (*t_startpc)(void); /* PC where thread started */
115     struct cpu *t_bound_cpu; /* cpu bound to, or NULL if not bound */
116     short t_affinitycnt; /* nesting level of kernel affinity-setting */
117     short t_bind_cpu; /* user-specified CPU binding (-1 if none) */
118     ushort_t t_flag; /* modified only by current thread */
119     ushort_t t_proc_flag; /* modified holding tproc(t)->p_lock */
120     ushort_t t_schedflag; /* modified holding thread_lock(t) */
121     volatile char t_preempt; /* don't preempt thread if set */
122     volatile char t_preempt_lk;
123     uint_t t_state; /* thread state (protected by thread_lock) */
124     pri_t t_pri; /* assigned thread priority */
125     pri_t t_epri; /* inherited thread priority */
126     pri_t t_cpri; /* thread scheduling class priority */
127     char t_writer; /* sleeping in lwp_rwlock_lock(RW_WRITE_LOCK) */
128     uchar_t t_bindflag; /* CPU and pset binding type */
129     label_t t_pcb; /* pcb, save area when switching */
130     lwpchan_t t_lwpchan; /* reason for blocking */
131 #define t_wchan0 t_lwpchan.lc_wchan0
132 #define t_wchan t_lwpchan.lc_wchan
133     struct _sobj_ops *t_sobj_ops;
134     id_t t_cid; /* scheduling class id */
135     struct thread_ops *t_clfuncs; /* scheduling class ops vector */
136     void *t_cldata; /* per scheduling class specific data */
137     ctxop_t *t_ctx; /* thread context */
138     uintptr_t t_lofault; /* ret pc for failed page faults */
139     label_t *t_onfault; /* on_fault() setjmp buf */
140     struct on_trap_data *t_ontrap; /* on_trap() protection data */
141     caddr_t t_swap; /* the bottom of the stack, if from segkp */
142     lock_t t_lock; /* used to resume() a thread */
143     uint8_t t_lockstat; /* set while thread is in lockstat code */
144     uint8_t t_pil; /* interrupt thread PIL */
145     disp_lock_t t_pi_lock; /* lock protecting t_prioinv list */
146     char t_nomigrate; /* do not migrate if set */
147     struct cpu *t_cpu; /* CPU that thread last ran on */
148     struct cpu *t_weakbound_cpu; /* cpu weakly bound to */
149     struct lgrp_ld *t_lpl; /* load average for home lgroup */
150     void *t_lgrp_reserv[2]; /* reserved for future */
151     struct _kthread *t_intr; /* interrupted (pinned) thread */
152     uint64_t t_intr_start; /* timestamp when time slice began */
153     kt_did_t t_did; /* thread id for kernel debuggers */
154     caddr_t t_tnf_tpd; /* Trace facility data pointer */

```

```

155     struct _kcpc_ctx *t_cpc_ctx; /* performance counter context */
156     struct _kcpc_set *t_cpc_set; /* set this thread has bound */

158 /*
159  * non swappable part of the lwp state.
160  */
161     id_t t_tid; /* lwp's id */
162     id_t t_waitfor; /* target lwp id in lwp_wait() */
163     struct sigqueue *t_sigqueue; /* queue of siginfo structs */
164     k_sigset_t t_sig; /* signals pending to this process */
165     k_sigset_t t_extsig; /* signals sent from another contract */
166     k_sigset_t t_hold; /* hold signal bit mask */
167     k_sigset_t t_sigwait; /* sigtimedwait/sigfd accepting these */
168     struct _kthread *t_forw; /* process's forward thread link */
169     struct _kthread *t_back; /* process's backward thread link */
170     struct _kthread *t_thlink; /* tid (lwpid) lookup hash link */
171     k_lwp_t *t_lwp; /* thread's lwp pointer */
172     struct proc *t_proc; /* proc pointer */
173     struct t_audit_data *t_audit_data; /* per thread audit data */
174     struct _kthread *t_next; /* doubly linked list of all threads */
175     struct _kthread *t_prev;
176     ushort_t t_whystop; /* reason for stopping */
177     ushort_t t_whatstop; /* more detailed reason */
178     int t_dslot; /* index in proc's thread directory */
179     struct pollstate *t_pollstate; /* state used during poll(2) */
180     struct pollcache *t_pollcache; /* to pass a pcache ptr by /dev/poll */
181     struct cred *t_cred; /* pointer to current cred */
182     time_t t_start; /* start time, seconds since epoch */
183     clock_t t_lbolt; /* lbolt at last clock_tick() */
184     hrtime_t t_stoptime; /* timestamp at stop() */
185     uint_t t_pctcpu; /* %cpu at last clock_tick(), binary */
186     /* point at right of high-order bit */
187     /* system call number */
188     short t_sysnum;
189     kcondvar_t t_delay_cv;
190     kmutex_t t_delay_lock;

191 /*
192  * Pointer to the dispatcher lock protecting t_state and state-related
193  * flags. This pointer can change during waits on the lock, so
194  * it should be grabbed only by thread_lock().
195  */
196     disp_lock_t *t_lock; /* pointer to the dispatcher lock */
197     ushort_t t_oldspl; /* spl level before dispatcher locked */
198     volatile char t_pre_sys; /* pre-syscall work needed */
199     lock_t t_lock_flush; /* for lock_mutex_flush() impl */
200     struct _disp *t_disp_queue; /* run queue for chosen CPU */
201     clock_t t_disp_time; /* last time this thread was running */
202     uint_t t_kpri_req; /* kernel priority required */

203 /*
204  * Post-syscall / post-trap flags.
205  * No lock is required to set these.
206  * These must be cleared only by the thread itself.
207  */
208     * t_astflag indicates that some post-trap processing is required,
209     * possibly a signal or a preemption. The thread will not
210     * return to user with this set.
211     * t_post_sys indicates that some unusual post-system call
212     * handling is required, such as an error or tracing.
213     * t_sig_check indicates that some condition in ISSIG() must be
214     * checked, but doesn't prevent returning to user.
215     * t_post_sys_ast is a way of checking whether any of these three
216     * flags are set.
217     */
218     union __tu {
219         struct __ts {

```

```

218         volatile char    _t_astflag;    /* AST requested */
219         volatile char    _t_sig_check;  /* ISSIG required */
220         volatile char    _t_post_sys;   /* post_syscall req */
221         volatile char    _t_trapret;    /* call CL_TRAPRET */
222     } _ts;
223     volatile int        _t_post_sys_ast; /* OR of these flags */
224 } _tu;
225 #define t_astflag        _tu._ts._t_astflag
226 #define t_sig_check     _tu._ts._t_sig_check
227 #define t_post_sys      _tu._ts._t_post_sys
228 #define t_trapret       _tu._ts._t_trapret
229 #define t_post_sys_ast  _tu._t_post_sys_ast

231 /*
232  * Real time microstate profiling.
233  */
234 /* possible 4-byte filler */
235 hrtime_t t_waitrq;    /* timestamp for run queue wait time */
236 int t_mstate;        /* current microstate */
237 struct rprof {
238     int    rp_anystate;    /* set if any state non-zero */
239     uint_t rp_state[NMSTATES]; /* mstate profiling counts */
240 } *t_rprof;

242 /*
243  * There is a turnstile inserted into the list below for
244  * every priority inverted synchronization object that
245  * this thread holds.
246  */

248 struct turnstile *t_prioinv;

250 /*
251  * Pointer to the turnstile attached to the synchronization
252  * object where this thread is blocked.
253  */

255 struct turnstile *t_ts;

257 /*
258  * kernel thread specific data
259  * Borrowed from userland implementation of POSIX tsd
260  */
261 struct tsd_thread {
262     struct tsd_thread *ts_next;    /* threads with TSD */
263     struct tsd_thread *ts_prev;    /* threads with TSD */
264     uint_t            ts_nkeys;    /* entries in value array */
265     void              **ts_value; /* array of value/key */
266 } *t_tsd;

271 clock_t    t_stime;    /* time stamp used by the swapper */
272 struct door_data *t_door; /* door invocation data */
273 kmutex_t    t_t_plockp; /* pointer to process's p_lock */

271 struct sc_shared *t_schedctl; /* scheduler activations shared data */
272 uintptr_t        t_sc_uaddr;  /* user-level address of shared data */

274 struct cpupart    *t_cpupart; /* partition containing thread */
275 int                t_bind_pset; /* processor set binding */

277 struct copyops    *t_copyops; /* copy in/out ops vector */

279 caddr_t          t_stkbase; /* base of the the stack */
280 struct page       *t_red_pp; /* if non-NULL, redzone is mapped */

282 afd_t            t_activefd; /* active file descriptor table */

```

```

284 struct _kthread *t_priforw; /* sleepq per-priority sublist */
285 struct _kthread *t_priback;

287 struct sleepq *t_sleepq; /* sleep queue thread is waiting on */
288 struct panic_trap_info *t_panic_trap; /* saved data from fatal trap */
289 int            *t_lgrp_affinity; /* lgroup affinity */
290 struct upimutex *t_upimutex; /* list of upimutexes owned by thread */
291 uint32_t        t_nupinest; /* number of nested held upi mutexes */
292 struct kproject *t_proj; /* project containing this thread */
293 uint8_t         t_unpark; /* modified holding t_delay_lock */
294 uint8_t         t_release; /* lwp_release() waked up the thread */
295 uint8_t         t_hatdepth; /* depth of recursive hat_memloads */
296 uint8_t         t_xpvcntr; /* see xen_block_migrate() */
297 kcondvar_t     t_joincv; /* cv used to wait for thread exit */
298 void           *t_taskq; /* for threads belonging to taskq */
299 hrtime_t        t_anttime; /* most recent time anticipatory load */
300 /* was added to an lgroup's load */
301 /* on this thread's behalf */
302 char            *t_pdmsg; /* privilege debugging message */

304 uint_t          t_predcache; /* DTrace predicate cache */
305 hrtime_t        t_dtrace_vtime; /* DTrace virtual time */
306 hrtime_t        t_dtrace_start; /* DTrace slice start time */

308 uint8_t         t_dtrace_stop; /* indicates a DTrace-desired stop */
309 uint8_t         t_dtrace_sig; /* signal sent via DTrace's raise() */

311 union __tdu {
312     struct __tds {
313         uint8_t _t_dtrace_on; /* hit a fasttrap tracepoint */
314         uint8_t _t_dtrace_step; /* about to return to kernel */
315         uint8_t _t_dtrace_ret; /* handling a return probe */
316         uint8_t _t_dtrace_ast; /* saved ast flag */
317 #ifdef __amd64
318         uint8_t _t_dtrace_reg; /* modified register */
319 #endif
320     } _tds;
321     ulong_t _t_dtrace_ft; /* bitwise or of these flags */
322 } _tdu;
323 #define t_dtrace_ft    _tdu._tds._t_dtrace_ft
324 #define t_dtrace_on    _tdu._tds._t_dtrace_on
325 #define t_dtrace_step  _tdu._tds._t_dtrace_step
326 #define t_dtrace_ret   _tdu._tds._t_dtrace_ret
327 #define t_dtrace_ast   _tdu._tds._t_dtrace_ast
328 #ifdef __amd64
329 #define t_dtrace_reg   _tdu._tds._t_dtrace_reg
330 #endif

332 uintptr_t        t_dtrace_pc; /* DTrace saved pc from fasttrap */
333 uintptr_t        t_dtrace_npc; /* DTrace next pc from fasttrap */
334 uintptr_t        t_dtrace_scrpc; /* DTrace per-thread scratch location */
335 uintptr_t        t_dtrace_astpc; /* DTrace return sequence location */
336 #ifdef __amd64
337 uint64_t         t_dtrace_regv; /* DTrace saved reg from fasttrap */
338 #endif
339 hrtime_t         t_hrtime; /* high-res last time on cpu */
340 kmutex_t         t_ctx_lock; /* protects t_ctx in removectx() */
341 struct waitq     *t_waitq; /* wait queue */
342 kmutex_t         t_wait_mutex; /* used in CV wait functions */
343 } kthread_t;

345 /*
346  * Thread flag (t_flag) definitions.
347  * These flags must be changed only for the current thread,
348  * and not during preemption code, since the code being

```



```

349 *      preempted could be modifying the flags.
350 *
351 *      For the most part these flags do not need locking.
352 *      The following flags will only be changed while the thread_lock is held,
353 *      to give assurance that they are consistent with t_state:
354 *      T_WAKEABLE
355 */
356 #define T_INTR_THREAD 0x0001 /* thread is an interrupt thread */
357 #define T_WAKEABLE 0x0002 /* thread is blocked, signals enabled */
358 #define T_TOMASK 0x0004 /* use lwp_sigoldmask on return from signal */
359 #define T_TALLOCTK 0x0008 /* thread structure allocated from stk */
360 #define T_FORKALL 0x0010 /* thread was cloned by forkall() */
361 #define T_WOULDBLOCK 0x0020 /* for lockfs */
362 #define T_DONTBLOCK 0x0040 /* for lockfs */
363 #define T_DONTPEND 0x0080 /* for lockfs */
364 #define T_SYS_PROF 0x0100 /* profiling on for duration of system call */
365 #define T_WAITCVSEM 0x0200 /* waiting for a lwp_cv or lwp_sema on sleepq */
366 #define T_WATCHPT 0x0400 /* thread undergoing a watchpoint emulation */
367 #define T_PANIC 0x0800 /* thread initiated a system panic */
368 #define T_LWP_REUSE 0x1000 /* stack and LWP can be reused */
369 #define T_CAPTURING 0x2000 /* thread is in page capture logic */
370 #define T_VFPARENT 0x4000 /* thread is vfork parent, must call vfwait */
371 #define T_DONTDTRACE 0x8000 /* disable DTrace probes */

373 /*
374 * Flags in t_proc_flag.
375 * These flags must be modified only when holding the p_lock
376 * for the associated process.
377 */
378 #define TP_DAEMON 0x0001 /* this is an LWP_DAEMON lwp */
379 #define TP_HOLDLWP 0x0002 /* hold thread's lwp */
380 #define TP_TWAIT 0x0004 /* wait to be freed by lwp_wait() */
381 #define TP_LWPEXIT 0x0008 /* lwp has exited */
382 #define TP_PRSTOP 0x0010 /* thread is being stopped via /proc */
383 #define TP_CHKPT 0x0020 /* thread is being stopped via CPR checkpoint */
384 #define TP_EXITLWP 0x0040 /* terminate this lwp */
385 #define TP_PRVSTOP 0x0080 /* thread is virtually stopped via /proc */
386 #define TP_MSACCT 0x0100 /* collect micro-state accounting information */
387 #define TP_STOPPING 0x0200 /* thread is executing stop() */
388 #define TP_WATCHPT 0x0400 /* process has watchpoints in effect */
389 #define TP_PAUSE 0x0800 /* process is being stopped via pauselwps() */
390 #define TP_CHANGEBIND 0x1000 /* thread has a new cpu/cpart binding */
391 #define TP_ZTHREAD 0x2000 /* this is a kernel thread for a zone */
392 #define TP_WATCHSTOP 0x4000 /* thread is stopping via holdwatch() */

394 /*
395 * Thread scheduler flag (t_schedflag) definitions.
396 * The thread must be locked via thread_lock() or equiv. to change these.
397 */
402 #define TS_LOAD 0x0001 /* thread is in memory */
403 #define TS_DONT_SWAP 0x0002 /* thread/lwp should not be swapped */
404 #define TS_SWAPEQ 0x0004 /* swap thread when it reaches a safe point */
405 #define TS_ON_SWAPO 0x0008 /* thread is on the swap queue */
398 #define TS_SIGNALLED 0x0010 /* thread was awakened by cv_signal() */
399 #define TS_PROJWAITQ 0x0020 /* thread is on its project's waitq */
400 #define TS_ZONEWAITQ 0x0040 /* thread is on its zone's waitq */
401 #define TS_CSTART 0x0100 /* setrun() by continualwps() */
402 #define TS_UNPAUSE 0x0200 /* setrun() by unpauselwps() */
403 #define TS_XSTART 0x0400 /* setrun() by SIGCONT */
404 #define TS_PSTART 0x0800 /* setrun() by /proc */
405 #define TS_RESUME 0x1000 /* setrun() by CPR resume process */
406 #define TS_CREATE 0x2000 /* setrun() by syslwp_create() */
407 #define TS_RUNQMATCH 0x4000 /* exact run queue balancing by setbackdq() */
408 #define TS_ALLSTART \
409 (TS_CSTART|TS_UNPAUSE|TS_XSTART|TS_PSTART|TS_RESUME|TS_CREATE)
410 #define TS_ANYWAITQ (TS_PROJWAITQ|TS_ZONEWAITQ)

```

```

412 /*
413 * Thread binding types
414 */
415 #define TB_ALLHARD 0
416 #define TB_CPU_SOFT 0x01 /* soft binding to CPU */
417 #define TB_PSET_SOFT 0x02 /* soft binding to pset */

419 #define TB_CPU_SOFT_SET(t) ((t)->t_bindflag |= TB_CPU_SOFT)
420 #define TB_CPU_HARD_SET(t) ((t)->t_bindflag &= ~TB_CPU_SOFT)
421 #define TB_PSET_SOFT_SET(t) ((t)->t_bindflag |= TB_PSET_SOFT)
422 #define TB_PSET_HARD_SET(t) ((t)->t_bindflag &= ~TB_PSET_SOFT)
423 #define TB_CPU_IS_SOFT(t) ((t)->t_bindflag & TB_CPU_SOFT)
424 #define TB_CPU_IS_HARD(t) (!TB_CPU_IS_SOFT(t))
425 #define TB_PSET_IS_SOFT(t) ((t)->t_bindflag & TB_PSET_SOFT)

427 /*
428 * No locking needed for AST field.
429 */
430 #define aston(t) ((t)->t_astflag = 1)
431 #define astoff(t) ((t)->t_astflag = 0)

433 /* True if thread is stopped on an event of interest */
434 #define ISTOPPED(t) ((t)->t_state == TS_STOPPED && \
435 !((t)->t_schedflag & TS_PSTART))

437 /* True if thread is asleep and wakeable */
438 #define ISWAKEABLE(t) (((t)->t_state == TS_SLEEP && \
439 ((t)->t_flag & T_WAKEABLE)))

441 /* True if thread is on the wait queue */
442 #define ISWAITING(t) ((t)->t_state == TS_WAIT)

444 /* similar to ISTOPPED except the event of interest is CPR */
445 #define CPR_ISTOPPED(t) ((t)->t_state == TS_STOPPED && \
446 !((t)->t_schedflag & TS_RESUME))

448 /*
449 * True if thread is virtually stopped (is or was asleep in
450 * one of the lwp_*() system calls and marked to stop by /proc.)
451 */
452 #define VSTOPPED(t) ((t)->t_proc_flag & TP_PRVSTOP)

454 /* similar to VSTOPPED except the point of interest is CPR */
455 #define CPR_VSTOPPED(t) \
456 ((t)->t_state == TS_SLEEP && \
457 (t)->t_wchan0 != NULL && \
458 ((t)->t_flag & T_WAKEABLE) && \
459 ((t)->t_proc_flag & TP_CHKPT))

461 /* True if thread has been stopped by hold*() or was created stopped */
462 #define SUSPENDED(t) ((t)->t_state == TS_STOPPED && \
463 ((t)->t_schedflag & (TS_CSTART|TS_UNPAUSE)) != (TS_CSTART|TS_UNPAUSE))

465 /* True if thread possesses an inherited priority */
466 #define INHERITED(t) ((t)->t_epri != 0)

468 /* The dispatch priority of a thread */
469 #define DISP_PRIO(t) ((t)->t_epri > (t)->t_pri ? (t)->t_epri : (t)->t_pri)

471 /* The assigned priority of a thread */
472 #define ASSIGNED_PRIO(t) ((t)->t_pri)

474 /*
483 * Macros to determine whether a thread can be swapped.
484 * If t_lock is held, the thread is either on a processor or being swapped.

```



```
663         disp_lock_exit_high(tlp);           \
664     }

666 /*
667  * Interrupt threads are created in TS_FREE state, and their lock
668  * points at the associated CPU's lock.
669  */
670 #define THREAD_FREEINTR(tp, cpu) \
671     THREAD_SET_STATE(tp, TS_FREE, &(cpu)->cpu_thread_lock)

673 /* if tunable kmem_stackinfo is set, fill kthread stack with a pattern */
674 #define KMEM_STKINFO_PATTERN    0xbadcbaadcbadcbaadcbadcULL

676 /*
677  * If tunable kmem_stackinfo is set, log the latest KMEM_LOG_STK_USAGE_SIZE
678  * dead kthreads that used their kernel stack the most.
679  */
680 #define KMEM_STKINFO_LOG_SIZE    16

682 /* kthread name (cmd/lwpid) string size in the stackinfo log */
683 #define KMEM_STKINFO_STR_SIZE    64

685 /*
686  * stackinfo logged data.
687  */
688 typedef struct kmem_stkinfo {
689     caddr_t kthread;           /* kthread pointer */
690     caddr_t t_startpc;        /* where kthread started */
691     caddr_t start;            /* kthread stack start address */
692     size_t stksz;             /* kthread stack size */
693     size_t percent;           /* kthread stack high water mark */
694     id_t t_tid;               /* kthread id */
695     char cmd[KMEM_STKINFO_STR_SIZE]; /* kthread name (cmd/lwpid) */
696 } kmem_stkinfo_t;
unchanged portion omitted
```

```

*****
5125 Thu Feb 18 08:40:41 2016
new/usr/src/uts/common/sys/vmsystem.h
6583 remove whole-process swapping
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 /*      Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T      */
27 /*      All Rights Reserved      */

29 /*
30 * University Copyright- Copyright (c) 1982, 1986, 1988
31 * The Regents of the University of California
32 * All Rights Reserved
33 *
34 * University Acknowledgment- Portions of this document are derived from
35 * software developed by the University of California, Berkeley, and its
36 * contributors.
37 */

39 #ifndef _SYS_VMSYSTEM_H
40 #define _SYS_VMSYSTEM_H

42 #include <sys/proc.h>

44 #ifdef __cplusplus
45 extern "C" {
46 #endif

48 /*
49 * Miscellaneous virtual memory subsystem variables and structures.
50 */
51 #ifdef _KERNEL
52 extern pgcnt_t freemem;      /* remaining blocks of free memory */
53 extern pgcnt_t avefree;     /* 5 sec moving average of free memory */
54 extern pgcnt_t avefree30;   /* 30 sec moving average of free memory */
55 extern pgcnt_t deficit;     /* estimate of needs of new swapped in procs */
56 extern pgcnt_t nscan;      /* number of scans in last second */
57 extern pgcnt_t desscan;    /* desired pages scanned per second */
58 extern pgcnt_t slowscan;
59 extern pgcnt_t fastscan;
60 extern pgcnt_t pushes;     /* number of pages pushed to swap device */

```

```

62 /* writable copies of tunables */
63 extern pgcnt_t maxpggio;    /* max paging i/o per sec before start swaps */
64 extern pgcnt_t lotsfree;   /* max free before clock freezes */
65 extern pgcnt_t desfree;    /* minimum free pages before swapping begins */
66 extern pgcnt_t minifree;   /* no of pages to try to keep free via daemon */
67 extern pgcnt_t needfree;   /* no of pages currently being waited for */
68 extern pgcnt_t throttlefree; /* point at which we block PG_WAIT calls */
69 extern pgcnt_t pageout_reserve; /* point at which we deny non-PG_WAIT calls */
70 extern pgcnt_t pages_before_pager; /* XXX */

72 /*
73 * TRUE if the pageout daemon, fsflush daemon or the scheduler. These
74 * processes can't sleep while trying to free up memory since a deadlock
75 * will occur if they do sleep.
76 */
77 #define NOMEMWAIT() (ttoproc(curthread) == proc_pageout || \
78                    ttoproc(curthread) == proc_fsflush || \
79                    ttoproc(curthread) == proc_sched)

81 /* insure non-zero */
82 #define nz(x) ((x) != 0 ? (x) : 1)

84 /*
85 * Flags passed by the swapper to swapout routines of each
86 * scheduling class.
87 */
88 #define HARDSWAP 1
89 #define SOFTSWAP 2

91 /*
92 * Values returned by valid_usr_range()
93 */
94 #define RANGE_OKAY (0)
95 #define RANGE_BADADDR (1)
96 #define RANGE_BADPROT (2)

98 /*
99 * map_pgsz: temporary - subject to change.
100 */
101 #define MAPPGSZ_VA 0x01
102 #define MAPPGSZ_STK 0x02
103 #define MAPPGSZ_HEAP 0x04
104 #define MAPPGSZ_ISM 0x08

106 /*
107 * Flags for map_pgszcvect
108 */
109 #define MAPPGSZC_SHM 0x01
110 #define MAPPGSZC_PRIVM 0x02
111 #define MAPPGSZC_STACK 0x04
112 #define MAPPGSZC_HEAP 0x08

114 /*
115 * vacalign values for choose_addr
116 */
117 #define ADDR_NOVACALIGN 0
118 #define ADDR_VACALIGN 1

120 struct as;
121 struct page;
122 struct anon;

124 extern int maxslp;
124 extern ulong_t pginrate;
125 extern ulong_t pgoutrate;
127 extern void swapout_lwp(klwp_t *);

```

```
127 extern int valid_va_range(caddr_t *basep, size_t *lenp, size_t minlen,
128     int dir);
129 extern int valid_va_range_aligned(caddr_t *basep, size_t *lenp,
130     size_t minlen, int dir, size_t align, size_t redzone, size_t off);

132 extern int valid_usr_range(caddr_t, size_t, uint_t, struct as *, caddr_t);
133 extern int useracc(void *, size_t, int);
134 extern size_t map_pgsz(int maptype, struct proc *p, caddr_t addr, size_t len,
135     int memcntl);
136 extern uint_t map_pgszvec(caddr_t addr, size_t size, uintptr_t off, int flags,
137     int type, int memcntl);
138 extern int choose_addr(struct as *as, caddr_t *addrp, size_t len, offset_t off,
139     int vacalign, uint_t flags);
140 extern void map_addr(caddr_t *addrp, size_t len, offset_t off, int vacalign,
141     uint_t flags);
142 extern int map_addr_vacalign_check(caddr_t, u_offset_t);
143 extern void map_addr_proc(caddr_t *addrp, size_t len, offset_t off,
144     int vacalign, caddr_t userlimit, struct proc *p, uint_t flags);
145 extern void vmmeter(void);
146 extern int cow_mapin(struct as *, caddr_t, caddr_t, struct page **,
147     struct anon **, size_t *, int);

149 extern caddr_t ppmapin(struct page *, uint_t, caddr_t);
150 extern void ppmapout(caddr_t);

152 extern int pf_is_memory(pfn_t);

154 extern void dcache_flushall(void);

156 extern void *boot_virt_alloc(void *addr, size_t size);

158 extern size_t exec_get_spslew(void);

160 #endif /* _KERNEL */

162 #ifdef __cplusplus
163 }
_____unchanged_portion_omitted_
```

```

*****
18138 Thu Feb 18 08:40:41 2016
new/usr/src/uts/common/vm/anon.h
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

380 extern struct k_anoninfo k_anoninfo;

382 extern void anon_init(void);
383 extern struct anon *anon_alloc(struct vnode *, anoff_t);
384 extern void anon_dup(struct anon_hdr *, ulong_t,
385 struct anon_hdr *, ulong_t, size_t);
386 extern void anon_dup_fill_holes(struct anon_hdr *, ulong_t,
387 struct anon_hdr *, ulong_t, size_t, uint_t, int);
388 extern int anon_fill_cow_holes(struct seg *, caddr_t, struct anon_hdr *,
389 ulong_t, struct vnode *, u_offset_t, size_t, uint_t,
390 uint_t, struct vpage [], struct cred *);
391 extern void anon_free(struct anon_hdr *, ulong_t, size_t);
392 extern void anon_free_pages(struct anon_hdr *, ulong_t, size_t, uint_t);
393 extern void anon_disclaim(struct anon_map *, ulong_t, size_t);
394 extern int anon_getpage(struct anon **, uint_t *, struct page **,
395 size_t, struct seg *, caddr_t, enum seg_rw, struct cred *);
396 extern int swap_getconpage(struct vnode *, u_offset_t, size_t,
397 uint_t *, page_t *[], size_t, page_t *, uint_t *,
398 spgcnt_t *, struct seg *, caddr_t,
399 enum seg_rw, struct cred *);
400 extern int anon_map_getpages(struct anon_map *, ulong_t,
401 uint_t, struct seg *, caddr_t, uint_t,
402 uint_t *, page_t *[], uint_t *,
403 struct vpage [], enum seg_rw, int, int, int, struct cred *);
404 extern int anon_map_privatepages(struct anon_map *, ulong_t,
405 uint_t, struct seg *, caddr_t, uint_t,
406 page_t *[], struct vpage [], int, int, struct cred *);
407 extern struct page *anon_private(struct anon **, struct seg *,
408 caddr_t, uint_t, struct page *,
409 int, struct cred *);
410 extern struct page *anon_zero(struct seg *, caddr_t,
411 struct anon **, struct cred *);
412 extern int anon_map_createpages(struct anon_map *, ulong_t,
413 size_t, struct page **,
414 struct seg *, caddr_t,
415 enum seg_rw, struct cred *);
416 extern int anon_map_demotepages(struct anon_map *, ulong_t,
417 struct seg *, caddr_t, uint_t,
418 struct vpage [], struct cred *);
419 extern void anon_shmap_free_pages(struct anon_map *, ulong_t, size_t);
420 extern int anon_resvmem(size_t, boolean_t, zone_t *, int);
421 extern void anon_unresvmem(size_t, zone_t *);
422 extern struct anon_map *anonmap_alloc(size_t, size_t, int);
423 extern void anonmap_free(struct anon_map *);
424 extern void anonmap_purge(struct anon_map *);
425 extern void anon_swap_free(struct anon *, struct page *);
426 extern void anon_decref(struct anon *);
427 extern int non_anon(struct anon_hdr *, ulong_t, u_offset_t *, size_t *);
428 extern pgcnt_t anon_pages(struct anon_hdr *, ulong_t, pgcnt_t);
429 extern int anon_swap_adjust(pgcnt_t);
430 extern void anon_swap_restore(pgcnt_t);
431 extern struct anon_hdr *anon_create(pgcnt_t, int);
432 extern void anon_release(struct anon_hdr *, pgcnt_t);
433 extern struct anon *anon_get_ptr(struct anon_hdr *, ulong_t);
434 extern ulong_t *anon_get_slot(struct anon_hdr *, ulong_t);
435 extern struct anon *anon_get_next_ptr(struct anon_hdr *, ulong_t *);
436 extern int anon_set_ptr(struct anon_hdr *, ulong_t, struct anon *, int);
437 extern int anon_copy_ptr(struct anon_hdr *, ulong_t,
438 struct anon_hdr *, ulong_t, pgcnt_t, int);

```

```

439 extern pgcnt_t anon_grow(struct anon_hdr *, ulong_t *, pgcnt_t, pgcnt_t, int);
440 extern void anon_array_enter(struct anon_map *, ulong_t,
441 anon_sync_obj_t *);
442 extern int anon_array_try_enter(struct anon_map *, ulong_t,
443 anon_sync_obj_t *);
444 extern void anon_array_exit(anon_sync_obj_t *);

444 /*
445 * anon_resv checks to see if there is enough swap space to fulfill a
446 * request and if so, reserves the appropriate anonymous memory resources.
447 * anon_checkspace just checks to see if there is space to fulfill the request,
448 * without taking any resources. Both return 1 if successful and 0 if not.
449 *
450 * Macros are provided as anon reservation is usually charged to the zone of
451 * the current process. In some cases (such as anon reserved by tmpfs), a
452 * zone pointer is needed to charge the appropriate zone.
453 */
454 #define anon_unresv(size) anon_unresvmem(size, curproc->p_zone)
455 #define anon_unresv_zone(size, zone) anon_unresvmem(size, zone)
456 #define anon_resv(size) \
457 anon_resvmem((size), 1, curproc->p_zone, 1)
458 #define anon_resv_zone(size, zone) anon_resvmem((size), 1, zone, 1)
459 #define anon_checkspace(size, zone) anon_resvmem((size), 0, zone, 0)
460 #define anon_try_resv_zone(size, zone) anon_resvmem((size), 1, zone, 0)

462 /*
463 * Flags to anon_private
464 */
465 #define STEAL_PAGE 0x1 /* page can be stolen */
466 #define LOCK_PAGE 0x2 /* page must be 'logically' locked */

468 /*
469 * SEGKP ANON pages that are locked are assumed to be LWP stack pages
470 * and thus count towards the user pages locked count.
471 * This value is protected by the same lock as availrmem.
472 */
473 extern pgcnt_t anon_segkp_pages_locked;

475 extern int anon_debug;

477 #ifdef ANON_DEBUG

479 #define A_ANON 0x01
480 #define A_RESV 0x02
481 #define A_MRESV 0x04

483 /* vararg-like debugging macro. */
484 #define ANON_PRINT(f, printf_args) \
485 if (anon_debug & f) \
486 printf printf_args

488 #else /* ANON_DEBUG */

490 #define ANON_PRINT(f, printf_args)

492 #endif /* ANON_DEBUG */

494 #endif /* _KERNEL */

496 #ifdef __cplusplus
497 }
_____unchanged_portion_omitted_____

```

```

*****
11482 Thu Feb 18 08:40:41 2016
new/usr/src/uts/common/vm/as.h
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

213 #ifndef _KERNEL

215 /*
216  * Flags for as_gap.
217  */
218 #define AH_DIR      0x1    /* direction flag mask */
219 #define AH_LO      0x0    /* find lowest hole */
220 #define AH_HI      0x1    /* find highest hole */
221 #define AH_CONTAIN  0x2    /* hole must contain 'addr' */

223 extern struct as kas;    /* kernel's address space */

225 /*
226  * Macros for address space locking. Note that we use RW_READER_STARVEWRITER
227  * whenever we acquire the address space lock as reader to assure that it can
228  * be used without regard to lock order in conjunction with filesystem locks.
229  * This allows filesystems to safely induce user-level page faults with
230  * filesystem locks held while concurrently allowing filesystem entry points
231  * acquiring those same locks to be called with the address space lock held as
232  * reader. RW_READER_STARVEWRITER thus prevents reader/reader+RW_WRITE_WANTED
233  * deadlocks in the style of fop_write()+as_fault()/as_*(*)+fop_putpage() and
234  * fop_read()+as_fault()/as_*(*)+fop_getpage(). (See the Big Theory Statement
235  * in rwlock.c for more information on the semantics of and motivation behind
236  * RW_READER_STARVEWRITER.)
237  */
238 #define AS_LOCK_ENTER(as, type)      rw_enter(&(as)->a_lock, \
239 (type) == RW_READER ? RW_READER_STARVEWRITER : (type))
240 #define AS_LOCK_EXIT(as)            rw_exit(&(as)->a_lock)
241 #define AS_LOCK_DESTROY(as)        rw_destroy(&(as)->a_lock)
242 #define AS_LOCK_TRYENTER(as, type)  rw_tryenter(&(as)->a_lock, \
243 (type) == RW_READER ? RW_READER_STARVEWRITER : (type))

245 /*
246  * Macros to test lock states.
247  */
248 #define AS_LOCK_HELD(as)            RW_LOCK_HELD(&(as)->a_lock)
249 #define AS_READ_HELD(as)           RW_READ_HELD(&(as)->a_lock)
250 #define AS_WRITE_HELD(as)          RW_WRITE_HELD(&(as)->a_lock)

252 /*
253  * macros to walk thru segment lists
254  */
255 #define AS_SEGFIRST(as)             avl_first(&(as)->a_segtree)
256 #define AS_SEGNEXT(as, seg)        AVL_NEXT(&(as)->a_segtree, (seg))
257 #define AS_SEGPREV(as, seg)        AVL_PREV(&(as)->a_segtree, (seg))

259 void      as_init(void);
260 void      as_avlinit(struct as *);
261 struct    seg *as_segat(struct as *as, caddr_t addr);
262 void      as_rangelock(struct as *as);
263 void      as_rangeunlock(struct as *as);
264 struct    as *as_alloc();
265 void      as_free(struct as *as);
266 int       as_dup(struct as *as, struct proc *forkedproc);
267 struct    seg *as_findseg(struct as *as, caddr_t addr, int tail);
268 int       as_addseg(struct as *as, struct seg *newseg);
269 struct    seg *as_removeseg(struct as *as, struct seg *seg);
270 faultcode_t as_fault(struct hat *hat, struct as *as, caddr_t addr, size_t size,
271 enum fault_type type, enum seg_rw rw);

```

```

272 faultcode_t as_faulta(struct as *as, caddr_t addr, size_t size);
273 int      as_setprot(struct as *as, caddr_t addr, size_t size, uint_t prot);
274 int      as_checkprot(struct as *as, caddr_t addr, size_t size, uint_t prot);
275 int      as_unmap(struct as *as, caddr_t addr, size_t size);
276 int      as_map(struct as *as, caddr_t addr, size_t size, int ((*crfp)()),
277 void *argsp);
278 void      as_purge(struct as *as);
279 int      as_gap(struct as *as, size_t minlen, caddr_t *basep, size_t *lenp,
280 uint_t flags, caddr_t addr);
281 int      as_gap_aligned(struct as *as, size_t minlen, caddr_t *basep,
282 size_t *lenp, uint_t flags, caddr_t addr, size_t align,
283 size_t redzone, size_t off);

285 int      as_memory(struct as *as, caddr_t *basep, size_t *lenp);
286 size_t   as_swapout(struct as *as);
286 int      as_incore(struct as *as, caddr_t addr, size_t size, char *vec,
287 size_t *sizep);
288 int      as_ctl(struct as *as, caddr_t addr, size_t size, int func, int attr,
289 uintptr_t arg, ulong_t *lock_map, size_t pos);
290 int      as_pagelock(struct as *as, struct page ***ppp, caddr_t addr,
291 size_t size, enum seg_rw rw);
292 void     as_pageunlock(struct as *as, struct page **pp, caddr_t addr,
293 size_t size, enum seg_rw rw);
294 int      as_setpagesize(struct as *as, caddr_t addr, size_t size, uint_t szc,
295 boolean_t wait);
296 int      as_set_default_lpsize(struct as *as, caddr_t addr, size_t size);
297 void     as_setwatch(struct as *as);
298 void     as_clearwatch(struct as *as);
299 int      as_getmemid(struct as *, caddr_t, memid_t *);

301 int      as_add_callback(struct as *, void (*)(), void *, uint_t,
302 caddr_t, size_t, int);
303 uint_t   as_delete_callback(struct as *, void *);

305 #endif /* _KERNEL */

307 #ifdef __cplusplus
308 }
_____unchanged_portion_omitted_____

```

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

1

```
*****
19654 Thu Feb 18 08:40:42 2016
new/usr/src/uts/common/vm/hat.h
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

81 typedef void *hat_region_cookie_t;

83 #ifdef _KERNEL

85 /*
86  * One time hat initialization
87  */
88 void hat_init(void);

90 /*
91  * Notify hat of a system dump
92  */
93 void hat_dump(void);

95 /*
96  * Operations on an address space:
97  *
98  * struct hat *hat_alloc(as)
99  * allocated a hat structure for as.
100 *
101 * void hat_free_start(hat)
102 * informs hat layer process has finished executing but as has not
103 * been cleaned up yet.
104 *
105 * void hat_free_end(hat)
106 * informs hat layer as is being destroyed. hat layer cannot use as
107 * pointer after this call.
108 *
109 * void hat_swapin(hat)
110 * allocate any hat resources required for process being swapped in.
111 *
112 * void hat_swapout(hat)
113 * deallocate hat resources for process being swapped out.
114 *
115 * size_t hat_get_mapped_size(hat)
116 * returns number of bytes that have valid mappings in hat.
117 *
118 * void hat_stats_enable(hat)
119 * void hat_stats_disable(hat)
120 * enables/disables collection of stats for hat.
121 *
122 * int hat_dup(parenthat, childhat, addr, len, flags)
123 * Duplicate address translations of the parent to the child. Supports
124 * the entire address range or a range depending on flag,
125 * zero returned on success, non-zero on error
126 *
127 * void hat_thread_exit(thread)
128 * Notifies the HAT that a thread is exiting, called after it has been
129 * reassigned to the kernel AS.
130 */

126 struct hat *hat_alloc(struct as *);
127 void hat_free_start(struct hat *);
128 void hat_free_end(struct hat *);
129 int hat_dup(struct hat *, struct hat *, caddr_t, size_t, uint_t);
136 void hat_swapin(struct hat *);
137 void hat_swapout(struct hat *);
130 size_t hat_get_mapped_size(struct hat *);
131 int hat_stats_enable(struct hat *);
```

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

2

```
132 void hat_stats_disable(struct hat *);
133 void hat_thread_exit(kthread_t *);

135 /*
136  * Operations on a named address within a segment:
137  *
138  * void hat_memload(hat, addr, pp, attr, flags)
139  * load/lock the given page struct
140  *
141  * void hat_memload_array(hat, addr, len, ppa, attr, flags)
142  * load/lock the given array of page structs
143  *
144  * void hat_devload(hat, addr, len, pf, attr, flags)
145  * load/lock the given page frame number
146  *
147  * void hat_unlock(hat, addr, len)
148  * unlock a given range of addresses
149  *
150  * void hat_unload(hat, addr, len, flags)
151  * void hat_unload_callback(hat, addr, len, flags, callback)
152  * unload a given range of addresses (has optional callback)
153  *
154  * void hat_sync(hat, addr, len, flags)
155  * synchronize mapping with software data structures
156  *
157  * void hat_map(hat, addr, len, flags)
158  *
159  * void hat_setattr(hat, addr, len, attr)
160  * void hat_clrattr(hat, addr, len, attr)
161  * void hat_chgattr(hat, addr, len, attr)
162  * modify attributes for a range of addresses. skips any invalid mappings
163  *
164  * uint_t hat_getattr(hat, addr, *attr)
165  * returns attr for <hat,addr> in *attr. returns 0 if there was a
166  * mapping and *attr is valid, nonzero if there was no mapping and
167  * *attr is not valid.
168  *
169  * size_t hat_getpagesize(hat, addr)
170  * returns pagesize in bytes for <hat, addr>. returns -1 if there is
171  * no mapping. This is an advisory call.
172  *
173  * pfn_t hat_getpfn(hat, addr)
174  * returns pfn for <hat, addr> or PFN_INVALID if mapping is invalid.
175  *
176  * int hat_probe(hat, addr)
177  * return 0 if no valid mapping is present. Faster version
178  * of hat_getattr in certain architectures.
179  *
180  * int hat_share(dhat, daddr, shat, saddr, len, szc)
181  *
182  * void hat_unshare(hat, addr, len, szc)
183  *
184  * void hat_chgprot(hat, addr, len, vprot)
185  * This is a deprecated call. New segment drivers should store
186  * all attributes and use hat_*attr calls.
187  * Change the protections in the virtual address range
188  * given to the specified virtual protection. If vprot is ~PROT_WRITE,
189  * then remove write permission, leaving the other permissions
190  * unchanged. If vprot is ~PROT_USER, remove user permissions.
191  *
192  * void hat_flush_range(hat, addr, size)
193  * Invalidate a virtual address translation for the local CPU.
194  */

196 void hat_memload(struct hat *, caddr_t, struct page *, uint_t, uint_t);
197 void hat_memload_array(struct hat *, caddr_t, size_t, struct page **,
```



```

198         uint_t, uint_t);
199 void    hat_memload_region(struct hat *, caddr_t, struct page *, uint_t,
200         uint_t, hat_region_cookie_t);
201 void    hat_memload_array_region(struct hat *, caddr_t, size_t, struct page **,
202         uint_t, uint_t, hat_region_cookie_t);

204 void    hat_devload(struct hat *, caddr_t, size_t, pfn_t, uint_t, int);

206 void    hat_unlock(struct hat *, caddr_t, size_t);
207 void    hat_unlock_region(struct hat *, caddr_t, size_t, hat_region_cookie_t);

209 void    hat_unload(struct hat *, caddr_t, size_t, uint_t);
210 void    hat_unload_callback(struct hat *, caddr_t, size_t, uint_t,
211         hat_callback_t *);
212 void    hat_flush_range(struct hat *, caddr_t, size_t);
213 void    hat_sync(struct hat *, caddr_t, size_t, uint_t);
214 void    hat_map(struct hat *, caddr_t, size_t, uint_t);
215 void    hat_setattr(struct hat *, caddr_t, size_t, uint_t);
216 void    hat_clrattr(struct hat *, caddr_t, size_t, uint_t);
217 void    hat_chgattr(struct hat *, caddr_t, size_t, uint_t);
218 uint_t  hat_getattr(struct hat *, caddr_t, uint_t *);
219 ssize_t hat_getpagesize(struct hat *, caddr_t);
220 pfn_t   hat_getpfnnum(struct hat *, caddr_t);
221 int     hat_probe(struct hat *, caddr_t);
222 int     hat_share(struct hat *, caddr_t, struct hat *, caddr_t, size_t, uint_t);
223 void    hat_unshare(struct hat *, caddr_t, size_t, uint_t);
224 void    hat_chgprot(struct hat *, caddr_t, size_t, uint_t);
225 void    hat_reserve(struct as *, caddr_t, size_t);
226 pfn_t   va_to_pfn(void *);
227 uint64_t va_to_pa(void *);

229 /*
230  * Kernel Physical Mapping (segkpm) hat interface routines.
231  */
232 caddr_t hat_kpm_mapin(struct page *, struct kpme *);
233 void    hat_kpm_mapout(struct page *, struct kpme *, caddr_t);
234 caddr_t hat_kpm_mapin_pfn(pfn_t);
235 void    hat_kpm_mapout_pfn(pfn_t);
236 caddr_t hat_kpm_page2va(struct page *, int);
237 struct page *hat_kpm_vaddr2page(caddr_t);
238 int     hat_kpm_fault(struct hat *, caddr_t);
239 void    hat_kpm_mseghash_clear(int);
240 void    hat_kpm_mseghash_update(pgcnt_t, struct memseg *);
241 void    hat_kpm_addmem_mseg_update(struct memseg *, pgcnt_t, offset_t);
242 void    hat_kpm_addmem_mseg_insert(struct memseg *);
243 void    hat_kpm_addmem_memsegs_update(struct memseg *);
244 caddr_t hat_kpm_mseg_reuse(struct memseg *);
245 void    hat_kpm_delmem_mseg_update(struct memseg *, struct memseg **);
246 void    hat_kpm_split_mseg_update(struct memseg *, struct memseg **,
247         struct memseg *, struct memseg *, struct memseg *);
248 void    hat_kpm_walk(void (*)(void *, void *, size_t), void *);

250 /*
251  * Operations on all translations for a given page(s)
252  *
253  * void hat_page_setattr(pp, flag)
254  * void hat_page_clrattr(pp, flag)
255  *     used to set/clear red/mod bits.
256  *
257  * uint hat_page_getattr(pp, flag)
258  *     If flag is specified, returns 0 if attribute is disabled
259  *     and non zero if enabled.  If flag specifies multiple attributes
260  *     then returns 0 if ALL attributes are disabled.  This is an advisory
261  *     call.
262  *
263  * int hat_pageunload(pp, forceflag)

```

```

264 *     unload all translations attached to pp.
265 *
266 * uint_t hat_pagesync(pp, flags)
267 *     get hw stats from hardware into page struct and reset hw stats
268 *     returns attributes of page
269 *
270 * ulong_t hat_page_getshare(pp)
271 *     returns approx number of mappings to this pp.  A return of 0 implies
272 *     there are no mappings to the page.
273 *
274 * faultcode_t hat_softlock(hat, addr, lenp, ppp, flags);
275 *     called to softlock pages for zero copy tcp
276 *
277 * void hat_page_demote(pp);
278 *     unload all large mappings to pp and decrease p_szc of all
279 *     constituent pages according to the remaining mappings.
280 */

282 void    hat_page_setattr(struct page *, uint_t);
283 void    hat_page_clrattr(struct page *, uint_t);
284 uint_t  hat_page_getattr(struct page *, uint_t);
285 int     hat_pageunload(struct page *, uint_t);
286 uint_t  hat_pagesync(struct page *, uint_t);
287 ulong_t hat_page_getshare(struct page *);
288 int     hat_page_checkshare(struct page *, ulong_t);
289 faultcode_t hat_softlock(struct hat *, caddr_t, size_t *,
290         struct page **, uint_t);
291 void    hat_page_demote(struct page *);

293 /*
294  * Routine to expose supported HAT features to PIM.
295  */
296 enum hat_features {
297     HAT_SHARED_PT, /* Shared page tables */
298     HAT_DYNAMIC_ISM_UNMAP, /* hat_pageunload() handles ISM pages */
299     HAT_VMODSORT, /* support for VMODSORT flag of vnode */
300     HAT_SHARED_REGIONS /* shared regions support */
301 };
_____ unchanged_portion_omitted

```

```

*****
10402 Thu Feb 18 08:40:42 2016
new/usr/src/uts/common/vm/seg.h
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

115 #define S_PURGE          (0x01)          /* seg should be purged in as_gap() */

117 struct seg_ops {
118     int      (*dup)(struct seg *, struct seg *);
119     int      (*unmap)(struct seg *, caddr_t, size_t);
120     void     (*free)(struct seg *);
121     faultcode_t (*fault)(struct hat *, struct seg *, caddr_t, size_t,
122         enum fault_type, enum seg_rw);
123     faultcode_t (*faulta)(struct seg *, caddr_t);
124     int      (*setprot)(struct seg *, caddr_t, size_t, uint_t);
125     int      (*checkprot)(struct seg *, caddr_t, size_t, uint_t);
126     int      (*kluster)(struct seg *, caddr_t, ssize_t);
127     size_t   (*swapout)(struct seg *);
128     int      (*sync)(struct seg *, caddr_t, size_t, int, uint_t);
129     size_t   (*incore)(struct seg *, caddr_t, size_t, char *);
130     int      (*lockop)(struct seg *, caddr_t, size_t, int, int, ulong_t *,
131         size_t);
132     int      (*getprot)(struct seg *, caddr_t, size_t, uint_t *);
133     u_offset_t (*getoffset)(struct seg *, caddr_t);
134     int      (*gettyp)(struct seg *, caddr_t, struct vnode **);
135     int      (*advise)(struct seg *, caddr_t, size_t, uint_t);
136     void     (*dump)(struct seg *);
137     int      (*pagelock)(struct seg *, caddr_t, size_t, struct page ***,
138         enum lock_type, enum seg_rw);
139     int      (*setpagesize)(struct seg *, caddr_t, size_t, uint_t);
140     int      (*getmemid)(struct seg *, caddr_t, memid_t *);
141     struct lgrp_mem_policy_info (*getpolicy)(struct seg *, caddr_t);
142     int      (*capable)(struct seg *, seccapability_t);
143     int      (*inherit)(struct seg *, caddr_t, size_t, uint_t);
144 };

146 #ifdef _KERNEL

148 /*
149  * Generic segment operations
150  */
151 extern void seg_init(void);
152 extern struct seg *seg_alloc(struct as *as, caddr_t base, size_t size);
153 extern int seg_attach(struct as *as, caddr_t base, size_t size,
154     struct seg *seg);
155 extern void seg_unmap(struct seg *seg);
156 extern void seg_free(struct seg *seg);

158 /*
159  * functions for pagelock cache support
160  */
161 typedef int (*seg_preclaim_cbfunc_t)(void *, caddr_t, size_t,
162     struct page **, enum seg_rw, int);

164 extern struct page **seg_plookup(struct seg *seg, struct anon_map *amp,
165     caddr_t addr, size_t len, enum seg_rw rw, uint_t flags);
166 extern void seg_pinactive(struct seg *seg, struct anon_map *amp,
167     caddr_t addr, size_t len, struct page **pp, enum seg_rw rw,
168     uint_t flags, seg_preclaim_cbfunc_t callback);

170 extern void seg_ppurge(struct seg *seg, struct anon_map *amp,
171     uint_t flags);
172 extern void seg_ppurge_wiredpp(struct page **pp);

```

```

174 extern int seg_pinsert_check(struct seg *seg, struct anon_map *amp,
175     caddr_t addr, size_t len, uint_t flags);
176 extern int seg_pinsert(struct seg *seg, struct anon_map *amp,
177     caddr_t addr, size_t len, size_t wlen, struct page **pp, enum seg_rw rw,
178     uint_t flags, seg_preclaim_cbfunc_t callback);

180 extern void seg_pasync_thread(void);
181 extern void seg_preap(void);
182 extern int seg_p_disable(void);
183 extern void seg_p_enable(void);

185 extern segadvstat_t segadvstat;

187 /*
188  * Flags for pagelock cache support.
189  * Flags argument is passed as uint_t to pcache routines. upper 16 bits of
190  * the flags argument are reserved for alignment page shift when SEGP_PSHIFT
191  * is set.
192  */
193 #define SEGP_FORCE_WIRED          0x1      /* skip check against seg_pwindow */
194 #define SEGP_AMP                  0x2      /* anon map's pcache entry */
195 #define SEGP_PSHIFT               0x4      /* addr pgsz shift for hash function */

197 /*
198  * Return values for seg_pinsert and seg_pinsert_check functions.
199  */
200 #define SEGP_SUCCESS              0        /* seg_pinsert() succeeded */
201 #define SEGP_FAIL                 1        /* seg_pinsert() failed */

203 /* Page status bits for segop_incure */
204 #define SEGP_PAGE_INCORE          0x01    /* VA has a page backing it */
205 #define SEGP_PAGE_LOCKED         0x02    /* VA has a page that is locked */
206 #define SEGP_PAGE_HASCOW        0x04    /* VA has a page with a copy-on-write */
207 #define SEGP_PAGE_SOFTLOCK      0x08    /* VA has a page with softlock held */
208 #define SEGP_PAGE_VNODEBACKED   0x10    /* Segment is backed by a vnode */
209 #define SEGP_PAGE_ANON          0x20    /* VA has an anonymous page */
210 #define SEGP_PAGE_VNODE         0x40    /* VA has a vnode page backing it */

212 #define SEGOP_DUP(s, n)           (*(s)->s_ops->dup)((s), (n))
213 #define SEGOP_UNMAP(s, a, l)     (*(s)->s_ops->unmap)((s), (a), (l))
214 #define SEGOP_FREE(s)           (*(s)->s_ops->free)((s))
215 #define SEGOP_FAULT(h, s, a, l, t, rw) \
216     (*(s)->s_ops->fault)((h), (s), (a), (l), (t), (rw))
217 #define SEGOP_FAULTA(s, a)      (*(s)->s_ops->faulta)((s), (a))
218 #define SEGOP_SETPROT(s, a, l, p) (*(s)->s_ops->setprot)((s), (a), (l), (p))
219 #define SEGOP_CHECKPROT(s, a, l, p) (*(s)->s_ops->checkprot)((s), (a), (l), (p))
220 #define SEGOP_KLUSTER(s, a, d)  (*(s)->s_ops->kluster)((s), (a), (d))
221 #define SEGOP_SWAPOUT(s)        (*(s)->s_ops->swapout)((s))
222 #define SEGOP_SYNC(s, a, l, atr, f) \
223     (*(s)->s_ops->sync)((s), (a), (l), (atr), (f))
224 #define SEGOP_INCORE(s, a, l, v) (*(s)->s_ops->incore)((s), (a), (l), (v))
225 #define SEGOP_LOCKOP(s, a, l, atr, op, b, p) \
226     (*(s)->s_ops->lockop)((s), (a), (l), (atr), (op), (b), (p))
227 #define SEGOP_GETPROT(s, a, l, p) (*(s)->s_ops->getprot)((s), (a), (l), (p))
228 #define SEGOP_GETOFFSET(s, a)    (*(s)->s_ops->getoffset)((s), (a))
229 #define SEGOP_GETTYPE(s, a)     (*(s)->s_ops->gettype)((s), (a))
230 #define SEGOP_GETVPP(s, a, vpp) (*(s)->s_ops->getvp)((s), (a), (vpp))
231 #define SEGOP_ADVISE(s, a, l, b) (*(s)->s_ops->advise)((s), (a), (l), (b))
232 #define SEGOP_DUMP(s)           (*(s)->s_ops->dump)((s))
233 #define SEGOP_PAGELOCK(s, a, l, p, t, rw) \
234     (*(s)->s_ops->pagelock)((s), (a), (l), (p), (t), (rw))
235 #define SEGOP_SETPAGESIZE(s, a, l, szc) \
236     (*(s)->s_ops->setpagesize)((s), (a), (l), (szc))
237 #define SEGOP_GETMEMID(s, a, mp) (*(s)->s_ops->getmemid)((s), (a), (mp))
238 #define SEGOP_GETPOLICY(s, a)   (*(s)->s_ops->getpolicy)((s), (a))

```

```
238 #define SEGOP_CAPABLE(s, c)      (*(s)->s_ops->capable)((s), (c))
239 #define SEGOP_INHERIT(s, a, l, b) (*(s)->s_ops->inherit)((s), (a), (l), (b))

241 #define seg_page(seg, addr) \
242     (((uintptr_t)((addr) - (seg)->s_base)) >> PAGESHIFT)

244 #define seg_pages(seg) \
245     (((uintptr_t)((seg)->s_size + PAGEOFFSET)) >> PAGESHIFT)

247 #define IE_NOMEM      -1      /* internal to seg layer */
248 #define IE_RETRY      -2      /* internal to seg layer */
249 #define IE_REATTACH   -3      /* internal to seg layer */

251 /* Values for SEGOP_INHERIT */
252 #define SEGP_INH_ZERO  0x01

254 int seg_inherit_notsup(struct seg *, caddr_t, size_t, uint_t);

256 /* Delay/retry factors for seg_p_mem_config_pre_del */
257 #define SEGP_PREDEL_DELAY_FACTOR      4
258 /*
259  * As a workaround to being unable to purge the pagelock
260  * cache during a DR delete memory operation, we use
261  * a stall threshold that is twice the maximum seen
262  * during testing. This workaround will be removed
263  * when a suitable fix is found.
264  */
265 #define SEGP_STALL_SECONDS      25
266 #define SEGP_STALL_THRESHOLD \
267     (SEGP_STALL_SECONDS * SEGP_PREDEL_DELAY_FACTOR)

269 #ifdef VMDEBUG

271 uint_t seg_page(struct seg *, caddr_t);
272 uint_t seg_pages(struct seg *);

274 #endif /* VMDEBUG */

276 boolean_t seg_can_change_zones(struct seg *);
277 size_t seg_swresv(struct seg *);

279 #endif /* _KERNEL */

281 #ifdef __cplusplus
282 }
  unchanged portion omitted

```

```

*****
113563 Thu Feb 18 08:40:42 2016
new/usr/src/uts/common/vm/seg_dev.c
6583 remove whole-process swapping
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright 2010 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
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 - segment of a mapped device.
42  *
43  * This segment driver is used when mapping character special devices.
44  */

46 #include <sys/types.h>
47 #include <sys/t_lock.h>
48 #include <sys/sysmacros.h>
49 #include <sys/vtrace.h>
50 #include <sys/system.h>
51 #include <sys/vmsystem.h>
52 #include <sys/mman.h>
53 #include <sys/errno.h>
54 #include <sys/kmem.h>
55 #include <sys/cmn_err.h>
56 #include <sys/vnode.h>
57 #include <sys/proc.h>
58 #include <sys/conf.h>
59 #include <sys/debug.h>
60 #include <sys/ddidevmap.h>
61 #include <sys/ddi_implfuncs.h>

```

```

62 #include <sys/lgrp.h>

64 #include <vm/page.h>
65 #include <vm/hat.h>
66 #include <vm/as.h>
67 #include <vm/seg.h>
68 #include <vm/seg_dev.h>
69 #include <vm/seg_kp.h>
70 #include <vm/seg_kmem.h>
71 #include <vm/vpage.h>

73 #include <sys/sunddi.h>
74 #include <sys/esunddi.h>
75 #include <sys/fs/snodel.h>

78 #if DEBUG
79 int segdev_debug;
80 #define DEBUGF(level, args) { if (segdev_debug >= (level)) cmn_err args; }
81 #else
82 #define DEBUGF(level, args)
83 #endif

85 /* Default timeout for devmap context management */
86 #define CTX_TIMEOUT_VALUE 0

88 #define HOLD_DHP_LOCK(dhp) if (dhp->dh_flags & DEVMAP_ALLOW_REMAP) \
89     { mutex_enter(&dhp->dh_lock); }

91 #define RELE_DHP_LOCK(dhp) if (dhp->dh_flags & DEVMAP_ALLOW_REMAP) \
92     { mutex_exit(&dhp->dh_lock); }

94 #define round_down_p2(a, s)    ((a) & ~((s) - 1))
95 #define round_up_p2(a, s)    (((a) + (s) - 1) & ~((s) - 1))

97 /*
98  * VA_PA_ALIGNED checks to see if both VA and PA are on pgsz boundary
99  * VA_PA_PGSIZE_ALIGNED check to see if VA is aligned with PA w.r.t. pgsz
100 */
101 #define VA_PA_ALIGNED(uvaddr, paddr, pgsz) \
102     (((uvaddr | paddr) & (pgsz - 1)) == 0)
103 #define VA_PA_PGSIZE_ALIGNED(uvaddr, paddr, pgsz) \
104     (((uvaddr ^ paddr) & (pgsz - 1)) == 0)

106 #define vpgtob(n)            ((n) * sizeof(struct vpage)) /* For brevity */

108 #define VTOCVP(vp)          (VTOS(vp)->s_commonvp) /* we "know" it's an snode */

110 static struct devmap_ctx *devmapctx_list = NULL;
111 static struct devmap_softlock *devmap_slist = NULL;

113 /*
114  * mutex, vnode and page for the page of zeros we use for the trash mappings.
115  * One trash page is allocated on the first ddi_umem_setup call that uses it
116  * XXX Eventually, we may want to combine this with what segnf does when all
117  * hat layers implement HAT_NOFAULT.
118  *
119  * The trash page is used when the backing store for a userland mapping is
120  * removed but the application semantics do not take kindly to a SIGBUS.
121  * In that scenario, the applications pages are mapped to some dummy page
122  * which returns garbage on read and writes go into a common place.
123  * (Perfect for NO_FAULT semantics)
124  * The device driver is responsible to communicating to the app with some
125  * other mechanism that such remapping has happened and the app should take
126  * corrective action.
127  * We can also use an anonymous memory page as there is no requirement to

```

```

128 * keep the page locked, however this complicates the fault code. RFE.
129 */
130 static struct vnode trashvp;
131 static struct page *trashpp;

133 /* Non-pageable kernel memory is allocated from the umem_np_arena. */
134 static vmem_t *umem_np_arena;

136 /* Set the cookie to a value we know will never be a valid umem_cookie */
137 #define DEVMAP_DEVMEM_COOKIE ((ddi_umem_cookie_t)0x1)

139 /*
140 * Macros to check if type of devmap handle
141 */
142 #define cookie_is_devmem(c) \
143     ((c) == (struct ddi_umem_cookie *)DEVMAP_DEVMEM_COOKIE)

145 #define cookie_is_pmem(c) \
146     ((c) == (struct ddi_umem_cookie *)DEVMAP_PMEM_COOKIE)

148 #define cookie_is_kpmem(c) (!cookie_is_devmem(c) && !cookie_is_pmem(c) && \
149     ((c)->type == KMEM_PAGEABLE))

151 #define dhp_is_devmem(dhp) \
152     (cookie_is_devmem((struct ddi_umem_cookie *)((dhp)->dh_cookie)))

154 #define dhp_is_pmem(dhp) \
155     (cookie_is_pmem((struct ddi_umem_cookie *)((dhp)->dh_cookie)))

157 #define dhp_is_kpmem(dhp) \
158     (cookie_is_kpmem((struct ddi_umem_cookie *)((dhp)->dh_cookie)))

160 /*
161 * Private seg op routines.
162 */
163 static int segdev_dup(struct seg *, struct seg *);
164 static int segdev_unmap(struct seg *, caddr_t, size_t);
165 static void segdev_free(struct seg *);
166 static faultcode_t segdev_fault(struct hat *, struct seg *, caddr_t, size_t,
167     enum fault_type, enum seg_rw);
168 static faultcode_t segdev_faulta(struct seg *, caddr_t);
169 static int segdev_setprot(struct seg *, caddr_t, size_t, uint_t);
170 static int segdev_checkprot(struct seg *, caddr_t, size_t, uint_t);
171 static void segdev_badop(void);
172 static int segdev_sync(struct seg *, caddr_t, size_t, int, uint_t);
173 static size_t segdev_incore(struct seg *, caddr_t, size_t, char *);
174 static int segdev_lockop(struct seg *, caddr_t, size_t, int, int,
175     ulong_t *, size_t);
176 static int segdev_getprot(struct seg *, caddr_t, size_t, uint_t *);
177 static u_offset_t segdev_getoffset(struct seg *, caddr_t);
178 static int segdev_gettype(struct seg *, caddr_t);
179 static int segdev_getvp(struct seg *, caddr_t, struct vnode **);
180 static int segdev_advise(struct seg *, caddr_t, size_t, uint_t);
181 static void segdev_dump(struct seg *);
182 static int segdev_pagelock(struct seg *, caddr_t, size_t,
183     struct page ***, enum lock_type, enum seg_rw);
184 static int segdev_setpagesize(struct seg *, caddr_t, size_t, uint_t);
185 static int segdev_getmemid(struct seg *, caddr_t, memid_t *);
186 static lgrp_mem_policy_info_t *segdev_getpolicy(struct seg *, caddr_t);
187 static int segdev_capable(struct seg *, segcapability_t);

189 /*
190 * XXX this struct is used by rootnex_map_fault to identify
191 * the segment it has been passed. So if you make it
192 * "static" you'll need to fix rootnex_map_fault.
193 */

```

```

194 struct seg_ops segdev_ops = {
195     segdev_dup,
196     segdev_unmap,
197     segdev_free,
198     segdev_fault,
199     segdev_faulta,
200     segdev_setprot,
201     segdev_checkprot,
202     (int (*)( ))segdev_badop, /* kluster */
203     (size_t (*)(struct seg *))NULL, /* swapout */
204     segdev_sync, /* sync */
205     segdev_incore,
206     segdev_lockop, /* lockop */
207     segdev_getprot,
208     segdev_getoffset,
209     segdev_gettype,
210     segdev_getvp,
211     segdev_advise,
212     segdev_dump,
213     segdev_pagelock,
214     segdev_setpagesize,
215     segdev_getmemid,
216     segdev_getpolicy,
217     segdev_capable,
218     seg_inherit_notsup
};
_____unchanged_portion_omitted_____

```

new/usr/src/uts/common/vm/seg\_kmem.c

1

\*\*\*\*\*

45398 Thu Feb 18 08:40:42 2016

new/usr/src/uts/common/vm/seg\_kmem.c

6583 remove whole-process swapping

\*\*\*\*\*

unchanged portion omitted

```
776 static struct seg_ops segkmem_ops = {
777     SEGKMEM_BADOP(int),          /* dup */
778     SEGKMEM_BADOP(int),          /* unmap */
779     SEGKMEM_BADOP(void),         /* free */
780     segkmem_fault,
781     SEGKMEM_BADOP(faultcode_t), /* faulta */
782     segkmem_setprot,
783     segkmem_checkprot,
784     segkmem_kluster,
785     SEGKMEM_BADOP(size_t),       /* swapout */
786     SEGKMEM_BADOP(int),          /* sync */
787     SEGKMEM_BADOP(size_t),       /* incore */
788     SEGKMEM_BADOP(int),          /* lockop */
789     SEGKMEM_BADOP(int),          /* getprot */
790     SEGKMEM_BADOP(u_offset_t),   /* getoffset */
791     SEGKMEM_BADOP(int),          /* gettype */
792     SEGKMEM_BADOP(int),          /* getvp */
793     SEGKMEM_BADOP(int),          /* advise */
794     segkmem_dump,
795     segkmem_pagelock,
796     SEGKMEM_BADOP(int),          /* setpgsz */
797     segkmem_getmemid,
798     segkmem_getpolicy,           /* getpolicy */
799     segkmem_capable,             /* capable */
800     seg_inherit_notsup,          /* inherit */
801 };
```

unchanged portion omitted

new/usr/src/uts/common/vm/seg\_kp.c

1

```
*****
36713 Thu Feb 18 08:40:42 2016
new/usr/src/uts/common/vm/seg_kp.c
6583 remove whole-process swapping
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
23 */
24
25 /* Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
26 /* All Rights Reserved */
27
28 /*
29 * Portions of this source code were derived from Berkeley 4.3 BSD
30 * under license from the Regents of the University of California.
31 */
32
33 /*
34 * segkp is a segment driver that administers the allocation and deallocation
35 * of pageable variable size chunks of kernel virtual address space. Each
36 * allocated resource is page-aligned.
37 *
38 * The user may specify whether the resource should be initialized to 0,
39 * include a redzone, or locked in memory.
40 */
41
42 #include <sys/types.h>
43 #include <sys/t_lock.h>
44 #include <sys/thread.h>
45 #include <sys/param.h>
46 #include <sys/errno.h>
47 #include <sys/sysmacros.h>
48 #include <sys/system.h>
49 #include <sys/buf.h>
50 #include <sys/mman.h>
51 #include <sys/vnode.h>
52 #include <sys/cmn_err.h>
53 #include <sys/swap.h>
54 #include <sys/tuneable.h>
55 #include <sys/kmem.h>
56 #include <sys/vmem.h>
57 #include <sys/cred.h>
58 #include <sys/dumphdr.h>
59 #include <sys/debug.h>
60 #include <sys/vtrace.h>
61 #include <sys/stack.h>
```

new/usr/src/uts/common/vm/seg\_kp.c

2

```
62 #include <sys/atomic.h>
63 #include <sys/archsystem.h>
64 #include <sys/lgrp.h>
65
66 #include <vm/as.h>
67 #include <vm/seg.h>
68 #include <vm/seg_kp.h>
69 #include <vm/seg_kmem.h>
70 #include <vm/anon.h>
71 #include <vm/page.h>
72 #include <vm/hat.h>
73 #include <sys/bitmap.h>
74
75 /*
76  * Private seg op routines
77 */
78 static void segkp_badop(void);
79 static void segkp_dump(struct seg *seg);
80 static int segkp_checkprot(struct seg *seg, caddr_t addr, size_t len,
81                          uint_t prot);
82 static int segkp_kluster(struct seg *seg, caddr_t addr, ssize_t delta);
83 static int segkp_pagelock(struct seg *seg, caddr_t addr, size_t len,
84                          struct page ***page, enum lock_type type,
85                          enum seg_rw rw);
86 static void segkp_insert(struct seg *seg, struct segkp_data *kpd);
87 static void segkp_delete(struct seg *seg, struct segkp_data *kpd);
88 static caddr_t segkp_get_internal(struct seg *seg, size_t len, uint_t flags,
89                                  struct segkp_data **tkpd, struct anon_map *amp);
90 static void segkp_release_internal(struct seg *seg,
91                                   struct segkp_data *kpd, size_t len);
92 static int segkp_unlock(struct hat *hat, struct seg *seg, caddr_t vaddr,
93                       size_t len, struct segkp_data *kpd, uint_t flags);
94 static int segkp_load(struct hat *hat, struct seg *seg, caddr_t vaddr,
95                      size_t len, struct segkp_data *kpd, uint_t flags);
96 static struct segkp_data *segkp_find(struct seg *seg, caddr_t vaddr);
97 static int segkp_getmemid(struct seg *seg, caddr_t addr, memid_t *memidp);
98 static lgrp_mem_policy_info_t *segkp_getpolicy(struct seg *seg,
99                                                  caddr_t addr);
100 static int segkp_capable(struct seg *seg, segcapability_t capability);
101
102 /*
103  * Lock used to protect the hash table(s) and caches.
104 */
105 static kmutex_t segkp_lock;
106
107 /*
108  * The segkp caches
109 */
110 static struct segkp_cache segkp_cache[SEGKP_MAX_CACHE];
111
112 #define SEGKP_BADOP(t) (t(*)())segkp_badop
113
114 /*
115  * When there are fewer than red_minavail bytes left on the stack,
116  * segkp_map_red() will map in the redzone (if called). 5000 seems
117  * to work reasonably well...
118 */
119 long red_minavail = 5000;
120
121 /*
122  * will be set to 1 for 32 bit x86 systems only, in startup.c
123 */
124 int segkp_fromheap = 0;
125 ulong_t *segkp_bitmap;
126
127 /*
```

```

128 * If segkp_map_red() is called with the redzone already mapped and
129 * with less than RED_DEEP_THRESHOLD bytes available on the stack,
130 * then the stack situation has become quite serious; if much more stack
131 * is consumed, we have the potential of scrogging the next thread/LWP
132 * structure. To help debug the "can't happen" panics which may
133 * result from this condition, we record hrestime and the calling thread
134 * in red_deep_hires and red_deep_thread respectively.
135 */
136 #define RED_DEEP_THRESHOLD      2000

138 hrtime_t      red_deep_hires;
139 kthread_t     *red_deep_thread;

141 uint32_t      red_nmapped;
142 uint32_t      red_closest = UINT_MAX;
143 uint32_t      red_ndoubles;

145 pgcnt_t anon_segkp_pages_locked;      /* See vm/anon.h */
146 pgcnt_t anon_segkp_pages_resv;      /* anon reserved by seg_kp */

148 static struct seg_ops segkp_ops = {
149     SEGKP_BADOP(int),      /* dup */
150     SEGKP_BADOP(int),      /* unmap */
151     SEGKP_BADOP(void),      /* free */
152     segkp_fault,
153     SEGKP_BADOP(faultcode_t), /* faulta */
154     SEGKP_BADOP(int),      /* setprot */
155     segkp_checkprot,
156     segkp_kluster,
157     SEGKP_BADOP(size_t), /* swapout */
158     SEGKP_BADOP(int),      /* sync */
159     SEGKP_BADOP(size_t), /* incore */
160     SEGKP_BADOP(int),      /* lockop */
161     SEGKP_BADOP(int),      /* getprot */
162     SEGKP_BADOP(u_offset_t), /* getoffset */
163     SEGKP_BADOP(int),      /* gettype */
164     SEGKP_BADOP(int),      /* getvp */
165     SEGKP_BADOP(int),      /* advise */
166     segkp_dump,            /* dump */
167     segkp_pagelock,        /* pagelock */
168     SEGKP_BADOP(int),      /* setpgsz */
169     segkp_getmemid,        /* getmemid */
170     segkp_getpolicy,       /* getpolicy */
171     segkp_capable,         /* capable */
172     seg_inherit_notsup     /* inherit */
173 };

```

**unchanged portion omitted**

```

757 /*
758 * segkp_map_red() will check the current frame pointer against the
759 * stack base. If the amount of stack remaining is questionable
760 * (less than red_minavail), then segkp_map_red() will map in the redzone
761 * and return 1. Otherwise, it will return 0. segkp_map_red() can
762 * only be called when it is safe to sleep on page_create_va().
763 * only be called when:
764 *
765 * - it is safe to sleep on page_create_va().
766 * - the caller is non-swappable.
767 *
768 * It is up to the caller to remember whether segkp_map_red() successfully
769 * mapped the redzone, and, if so, to call segkp_unmap_red() at a later
770 * time.
771 * time. Note that the caller must remain non-swappable until after
772 * calling segkp_unmap_red().
773 *
774 * Currently, this routine is only called from pagefault() (which necessarily

```

```

769 * satisfies the above conditions).
770 */
771 #if defined(STACK_GROWTH_DOWN)
772 int
773 segkp_map_red(void)
774 {
775     uintptr_t fp = STACK_BIAS + (uintptr_t)getfp();
776     #ifndef _LP64
777         caddr_t stkbase;
778     #endif

785     ASSERT(curthread->t_schedflag & TS_DONT_SWAP);

780     /*
781      * Optimize for the common case where we simply return.
782      */
783     if ((curthread->t_red_pp == NULL) &&
784         (fp - (uintptr_t)curthread->t_stkbase >= red_minavail))
785         return (0);

787 #if defined(_LP64)
788     /*
789      * XXX We probably need something better than this.
790      */
791     panic("kernel stack overflow");
792     /*NOTREACHED*/
793 #else /* _LP64 */
794     if (curthread->t_red_pp == NULL) {
795         page_t *red_pp;
796         struct seg kseg;

798         caddr_t red_va = (caddr_t)
799             (((uintptr_t)curthread->t_stkbase & (uintptr_t)PAGEMASK) -
800              PAGESIZE);

802         ASSERT(page_exists(&kvp, (u_offset_t)(uintptr_t)red_va) ==
803                 NULL);

805         /*
806          * Allocate the physical for the red page.
807          */
808         /*
809          * No PG_NORELOC here to avoid waits. Unlikely to get
810          * a relocate happening in the short time the page exists
811          * and it will be OK anyway.
812          */

814         kseg.s_as = &kas;
815         red_pp = page_create_va(&kvp, (u_offset_t)(uintptr_t)red_va,
816                               PAGESIZE, PG_WAIT | PG_EXCL, &kseg, red_va);
817         ASSERT(red_pp != NULL);

819         /*
820          * So we now have a page to jam into the redzone...
821          */
822         page_io_unlock(red_pp);

824         hat_memload(kas.a_hat, red_va, red_pp,
825                    (PROT_READ|PROT_WRITE), HAT_LOAD_LOCK);
826         page_downgrade(red_pp);

828         /*
829          * The page is left SE_SHARED locked so we can hold on to
830          * the page_t pointer.
831          */
832         curthread->t_red_pp = red_pp;

```



```

834         atomic_inc_32(&red_nmapped);
835         while (fp - (uintptr_t)curthread->t_stkbase < red_closest) {
836             (void) atomic_cas_32(&red_closest, red_closest,
837                 (uint32_t)(fp - (uintptr_t)curthread->t_stkbase));
838         }
839         return (1);
840     }

842     stkbase = (caddr_t)((uintptr_t)curthread->t_stkbase &
843         (uintptr_t)PAGEMASK) - PAGESIZE);

845     atomic_inc_32(&red_ndoubles);

847     if (fp - (uintptr_t)stkbase < RED_DEEP_THRESHOLD) {
848         /*
849          * Oh boy. We're already deep within the mapped-in
850          * redzone page, and the caller is trying to prepare
851          * for a deep stack run. We're running without a
852          * redzone right now: if the caller plows off the
853          * end of the stack, it'll plow another thread or
854          * LWP structure. That situation could result in
855          * a very hard-to-debug panic, so, in the spirit of
856          * recording the name of one's killer in one's own
857          * blood, we're going to record hrestime and the calling
858          * thread.
859          */
860         red_deep_hires = hrestime.tv_nsec;
861         red_deep_thread = curthread;
862     }

864     /*
865      * If this is a DEBUG kernel, and we've run too deep for comfort, toss.
866      */
867     ASSERT(fp - (uintptr_t)stkbase >= RED_DEEP_THRESHOLD);
868     return (0);
869 #endif /* _LP64 */
870 }

872 void
873 segkp_unmap_red(void)
874 {
875     page_t *pp;
876     caddr_t red_va = (caddr_t)((uintptr_t)curthread->t_stkbase &
877         (uintptr_t)PAGEMASK) - PAGESIZE);

879     ASSERT(curthread->t_red_pp != NULL);
887     ASSERT(curthread->t_schedflag & TS_DONT_SWAP);

881     /*
882      * Because we locked the mapping down, we can't simply rely
883      * on page_destroy() to clean everything up; we need to call
884      * hat_unload() to explicitly unlock the mapping resources.
885      */
886     hat_unload(kas.a_hat, red_va, PAGESIZE, HAT_UNLOAD_UNLOCK);

888     pp = curthread->t_red_pp;

890     ASSERT(pp == page_find(&kvp, (u_offset_t)(uintptr_t)red_va));

892     /*
893      * Need to upgrade the SE_SHARED lock to SE_EXCL.
894      */
895     if (!page_tryupgrade(pp)) {
896         /*
897          * As there is now wait for upgrade, release the

```

```

898         * SE_SHARED lock and wait for SE_EXCL.
899         */
900         page_unlock(pp);
901         pp = page_lookup(&kvp, (u_offset_t)(uintptr_t)red_va, SE_EXCL);
902         /* pp may be NULL here, hence the test below */
903     }

905     /*
906      * Destroy the page, with dontfree set to zero (i.e. free it).
907      */
908     if (pp != NULL)
909         page_destroy(pp, 0);
910     curthread->t_red_pp = NULL;
911 }

    _____ unchanged_portion_omitted _____

1305 /*
1306  * Find the kpd associated with a vaddr.
1307  */
1308  * Most of the callers of segkp_find will pass the vaddr that
1309  * hashes to the desired index, but there are cases where
1310  * this is not true in which case we have to (potentially) scan
1311  * the whole table looking for it. This should be very rare
1312  * (e.g. a segkp_fault(F_INVALID) on an address somewhere in the
1313  * middle of the segkp_data region).
1314  */
1315  static struct segkp_data *
1316  segkp_find(struct seg *seg, caddr_t vaddr)
1317  {
1318      struct segkp_segdata *kpsd = (struct segkp_segdata *)seg->s_data;
1319      struct segkp_data *kpd;
1320      int i;
1321      int stop;

1323      i = stop = SEGKP_HASH(vaddr);
1324      mutex_enter(&segkp_lock);
1325      do {
1326          for (kpd = kpsd->kpsd_hash[i]; kpd != NULL;
1327              kpd = kpd->kp_next) {
1328              if (vaddr >= kpd->kp_base &&
1329                  vaddr < kpd->kp_base + kpd->kp_len) {
1330                  mutex_exit(&segkp_lock);
1331                  return (kpd);
1332              }
1333          }
1334          if (--i < 0)
1335              i = SEGKP_HASHSZ - 1; /* Wrap */
1336      } while (i != stop);
1337      mutex_exit(&segkp_lock);
1338      return (NULL); /* Not found */
1347 }

1349 /*
1350  * returns size of swappable area.
1351  */
1352  size_t
1353  swapsize(caddr_t v)
1354  {
1355      struct segkp_data *kpd;

1357      if ((kpd = segkp_find(segkp, v)) != NULL)
1358          return (SEGKP_MAPLEN(kpd->kp_len, kpd->kp_flags));
1359      else
1360          return (NULL);
1369 }

    _____ unchanged_portion_omitted _____

```

new/usr/src/uts/common/vm/seg\_kp.h

1

```
*****
5245 Thu Feb 18 08:40:42 2016
new/usr/src/uts/common/vm/seg_kp.h
6583 remove whole-process swapping
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License, Version 1.0 only
6  * (the "License"). You may not use this file except in compliance
7  * with the License.
8  *
9  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
10 * or http://www.opensolaris.org/os/licensing.
11 * See the License for the specific language governing permissions
12 * and limitations under the License.
13 *
14 * When distributing Covered Code, include this CDDL HEADER in each
15 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
16 * If applicable, add the following below this CDDL HEADER, with the
17 * fields enclosed by brackets "[]" replaced with your own identifying
18 * information: Portions Copyright [yyyy] [name of copyright owner]
19 *
20 * CDDL HEADER END
21 */
22 /*
23  * Copyright 2004 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25  */
27 #ifndef _VM_SEG_KP_H
28 #define _VM_SEG_KP_H
30 #pragma ident      "%Z%M% %I%      %E% SMI"
30 /*
31  * segkp (as in kernel pageable) is a segment driver that supports allocation
32  * of page-aligned variable size of vm resources.
33  *
34  * Each vm resource represents a page-aligned range of virtual addresses.
35  * The caller may specify whether the resource should include a redzone,
36  * be locked down, or be zero initialized.
37  */
39 #include <vm/seg.h>
40 #include <sys/vmem.h>
42 #ifdef __cplusplus
43 extern "C" {
44 #endif
46 #ifdef _KERNEL
48 /*
49  * Private information per overall segkp segment (as opposed
50  * to per resource within segment). There are as many anon slots
51  * allocated as there are pages in the segment.
52  */
53 struct segkp_segdata {
54     struct anon_hdr *kpsd_anon;    /* anon structs */
55     vmem_t *kpsd_arena;          /* virtual memory descriptor */
56     struct segkp_data **kpsd_hash; /* Hash table for lookups */
57 };
58 unchanged_portion_omitted
109 #define SEGKP_MAX_CACHE      4      /* Number of caches maintained */
```

new/usr/src/uts/common/vm/seg\_kp.h

2

```
111 /*
112  * Define redzone, and stack_to_memory macros.
113  * The redzone is PAGESIZE bytes.
114  */
115 #ifdef STACK_GROWTH_DOWN
116 #define KPD_REDZONE(kpd)      (0)
117 #define stom(v, flags)      (((flags) & KPD_HASREDZONE) ? (v) + PAGESIZE : (v))
119 #else /* STACK_GROWTH_DOWN */
121 #define KPD_REDZONE(kpd)      (btop(kpd->kp_len) - 1)
122 #define stom(v)      (v)
123 #endif /* STACK_GROWTH_DOWN */
125 #define SEGKP_MAPLEN(len, flags) \
126     (((flags) & KPD_HASREDZONE) ? (len) - PAGESIZE : (len))
128 extern struct seg *segkp;
129 /* If segkp becomes more than one seg this test will need changing. */
130 #define SEG_IS_SEGKP(SEG)      ((SEG) == segkp)
132 /*
133  * Public routine declarations not part of the segment ops vector go here.
134  */
135 int      segkp_create(struct seg *seg);
136 caddr_t  segkp_get(struct seg *seg, size_t len, uint_t flags);
137 void     segkp_release(struct seg *seg, caddr_t vaddr);
138 void *   segkp_cache_init(struct seg *seg, int maxsize, size_t len,
139     uint_t flags);
140 void     segkp_cache_free();
141 caddr_t  segkp_cache_get(void *cookie);
142 int      segkp_map_red(void);
143 void     segkp_unmap_red(void);
144 size_t   swapsize(caddr_t v);
146 /* Special currently only used by schedctl. */
146 struct anon_map; /* Make the compiler happy about the next line. */
147 caddr_t  segkp_get_withanonmap(struct seg *, size_t, uint_t, struct anon_map *);
149 /*
150  * We allow explicit calls to segkp_fault, even though it's part
151  * of the segkp ops vector.
152  */
153 faultcode_t segkp_fault(struct hat *hat, struct seg *seg, caddr_t addr,
154     size_t len, enum fault_type type, enum seg_rw rw);
156 #endif /* _KERNEL */
158 #ifdef __cplusplus
159 }
unchanged_portion_omitted
```

```

*****
9862 Thu Feb 18 08:40:42 2016
new/usr/src/uts/common/vm/seg_kpm.c
6583 remove whole-process swapping
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License, Version 1.0 only
6  * (the "License"). You may not use this file except in compliance
7  * with the License.
8  *
9  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
10 * or http://www.opensolaris.org/os/licensing.
11 * See the License for the specific language governing permissions
12 * and limitations under the License.
13 *
14 * When distributing Covered Code, include this CDDL HEADER in each
15 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
16 * If applicable, add the following below this CDDL HEADER, with the
17 * fields enclosed by brackets "[]" replaced with your own identifying
18 * information: Portions Copyright [yyyy] [name of copyright owner]
19 *
20 * CDDL HEADER END
21 */
22 /*
23 * Copyright 2006 Sun Microsystems, Inc. All rights reserved.
24 * Use is subject to license terms.
25 */

27 /*
28 * Kernel Physical Mapping (kpm) segment driver (segkpm).
29 *
30 * This driver delivers along with the hat_kpm* interfaces an alternative
31 * mechanism for kernel mappings within the 64-bit Solaris operating system,
32 * which allows the mapping of all physical memory into the kernel address
33 * space at once. This is feasible in 64 bit kernels, e.g. for Ultrasparc II
34 * and beyond processors, since the available VA range is much larger than
35 * possible physical memory. Momentarily all physical memory is supported,
36 * that is represented by the list of memory segments (memsegs).
37 *
38 * Segkpm mappings have also very low overhead and large pages are used
39 * (when possible) to minimize the TLB and TSB footprint. It is also
40 * extensible for other than Sparc architectures (e.g. AMD64). Main
41 * advantage is the avoidance of the TLB-shutdown X-calls, which are
42 * normally needed when a kernel (global) mapping has to be removed.
43 *
44 * First example of a kernel facility that uses the segkpm mapping scheme
45 * is seg_map, where it is used as an alternative to hat_memload().
46 * See also hat layer for more information about the hat_kpm* routines.
47 * The kpm facility can be turned off at boot time (e.g. /etc/system).
48 */

50 #include <sys/types.h>
51 #include <sys/param.h>
52 #include <sys/sysmacros.h>
53 #include <sys/system.h>
54 #include <sys/vnode.h>
55 #include <sys/cmn_err.h>
56 #include <sys/debug.h>
57 #include <sys/thread.h>
58 #include <sys/cpuvar.h>
59 #include <sys/bitmap.h>
60 #include <sys/atomic.h>
61 #include <sys/lgrp.h>

```

```

63 #include <vm/seg_kmem.h>
64 #include <vm/seg_kpm.h>
65 #include <vm/hat.h>
66 #include <vm/as.h>
67 #include <vm/seg.h>
68 #include <vm/page.h>

70 /*
71 * Global kpm controls.
72 * See also platform and mmu specific controls.
73 *
74 * kpm_enable -- global on/off switch for segkpm.
75 * . Set by default on 64bit platforms that have kpm support.
76 * . Will be disabled from platform layer if not supported.
77 * . Can be disabled via /etc/system.
78 *
79 * kpm_smallpages -- use only regular/system pagesize for kpm mappings.
80 * . Can be useful for critical debugging of kpm clients.
81 * . Set to zero by default for platforms that support kpm large pages.
82 * . The use of kpm large pages reduces the footprint of kpm meta data
83 * and has all the other advantages of using large pages (e.g TLB
84 * miss reduction).
85 * . Set by default for platforms that don't support kpm large pages or
86 * where large pages cannot be used for other reasons (e.g. there are
87 * only few full associative TLB entries available for large pages).
88 *
89 * segmap_kpm -- separate on/off switch for segmap using segkpm:
90 * . Set by default.
91 * . Will be disabled when kpm_enable is zero.
92 * . Will be disabled when MAXBSIZE != PAGESIZE.
93 * . Can be disabled via /etc/system.
94 *
95 */
96 int kpm_enable = 1;
97 int kpm_smallpages = 0;
98 int segmap_kpm = 1;

100 /*
101 * Private seg op routines.
102 */
103 faultcode_t segkpm_fault(struct hat *hat, struct seg *seg, caddr_t addr,
104                          size_t len, enum fault_type type, enum seg_rw rw);
105 static void segkpm_dump(struct seg *);
106 static void segkpm_badop(void);
107 static int segkpm_notsup(void);
108 static int segkpm_capable(struct seg *, segcapability_t);

110 #define SEGKPM_BADOP(t) (t(*)())segkpm_badop
111 #define SEGKPM_NOTSUP (int(*)())segkpm_notsup

113 static struct seg_ops segkpm_ops = {
114     SEGKPM_BADOP(int), /* dup */
115     SEGKPM_BADOP(int), /* unmap */
116     SEGKPM_BADOP(void), /* free */
117     segkpm_fault,
118     SEGKPM_BADOP(int), /* faulta */
119     SEGKPM_BADOP(int), /* setprot */
120     SEGKPM_BADOP(int), /* checkprot */
121     SEGKPM_BADOP(int), /* kluster */
122     SEGKPM_BADOP(size_t), /* swapout */
122     SEGKPM_BADOP(int), /* sync */
123     SEGKPM_BADOP(size_t), /* incore */
124     SEGKPM_BADOP(int), /* lockop */
125     SEGKPM_BADOP(int), /* getprot */
126     SEGKPM_BADOP(u_offset_t), /* getoffset */

```

```
127     SEGKPM_BADOP(int),      /* gettype */
128     SEGKPM_BADOP(int),      /* getvp */
129     SEGKPM_BADOP(int),      /* advise */
130     segkpm_dump,            /* dump */
131     SEGKPM_NOTSUP,          /* pagelock */
132     SEGKPM_BADOP(int),      /* setpgsz */
133     SEGKPM_BADOP(int),      /* getmemid */
134     SEGKPM_BADOP(lgrp_mem_policy_info_t), /* getpolicy */
135     segkpm_capable,         /* capable */
136     seg_inherit_notSUP      /* inherit */
137 };
unchanged portion omitted
```

```

*****
58029 Thu Feb 18 08:40:43 2016
new/usr/src/uts/common/vm/seg_map.c
6583 remove whole-process swapping
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 /*      Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T      */
27 /*      All Rights Reserved      */

29 /*
30 * Portions of this source code were derived from Berkeley 4.3 BSD
31 * under license from the Regents of the University of California.
32 */

34 /*
35 * VM - generic vnode mapping segment.
36 *
37 * The segmap driver is used only by the kernel to get faster (than seg_vn)
38 * mappings [lower routine overhead; more persistent cache] to random
39 * vnode/offsets. Note than the kernel may (and does) use seg_vn as well.
40 */

42 #include <sys/types.h>
43 #include <sys/t_lock.h>
44 #include <sys/param.h>
45 #include <sys/sysmacros.h>
46 #include <sys/buf.h>
47 #include <sys/system.h>
48 #include <sys/vnode.h>
49 #include <sys/mman.h>
50 #include <sys/errno.h>
51 #include <sys/cred.h>
52 #include <sys/kmem.h>
53 #include <sys/vtrace.h>
54 #include <sys/cmn_err.h>
55 #include <sys/debug.h>
56 #include <sys/thread.h>
57 #include <sys/dumphdr.h>
58 #include <sys/bitmap.h>
59 #include <sys/lgrp.h>

61 #include <vm/seg_kmem.h>

```

```

62 #include <vm/hat.h>
63 #include <vm/as.h>
64 #include <vm/seg.h>
65 #include <vm/seg_kpm.h>
66 #include <vm/seg_map.h>
67 #include <vm/page.h>
68 #include <vm/pvn.h>
69 #include <vm/rm.h>

71 /*
72  * Private seg op routines.
73 */
74 static void      segmap_free(struct seg *seg);
75 faultcode_t segmap_fault(struct hat *hat, struct seg *seg, caddr_t addr,
76                          size_t len, enum fault_type type, enum seg_rw rw);
77 static faultcode_t segmap_faulta(struct seg *seg, caddr_t addr);
78 static int      segmap_checkprot(struct seg *seg, caddr_t addr, size_t len,
79                                  uint_t prot);
80 static int      segmap_kluster(struct seg *seg, caddr_t addr, ssize_t);
81 static int      segmap_getprot(struct seg *seg, caddr_t addr, size_t len,
82                                  uint_t *protv);
83 static u_offset_t segmap_getoffset(struct seg *seg, caddr_t addr);
84 static int      segmap_gettype(struct seg *seg, caddr_t addr);
85 static int      segmap_getvp(struct seg *seg, caddr_t addr, struct vnode **vpp);
86 static void      segmap_dump(struct seg *seg);
87 static int      segmap_pagelock(struct seg *seg, caddr_t addr, size_t len,
88                                  struct page ***ppp, enum lock_type type,
89                                  enum seg_rw rw);
90 static void      segmap_badop(void);
91 static int      segmap_getmemid(struct seg *seg, caddr_t addr, memid_t *memidp);
92 static lgrp_mem_policy_info_t *segmap_getpolicy(struct seg *seg,
93                                                  caddr_t addr);
94 static int      segmap_capable(struct seg *seg, segcapability_t capability);

96 /* segkpm support */
97 static caddr_t segmap_pagecreate_kpm(struct seg *, vnode_t *, u_offset_t,
98                                       struct smap *, enum seg_rw);
99 struct smap      *get_smap_kpm(caddr_t, page_t **);

101 #define SEGMAP_BADOP(t) (t(*)())segmap_badop

103 static struct seg_ops segmap_ops = {
104     SEGMAP_BADOP(int),      /* dup */
105     SEGMAP_BADOP(int),      /* unmap */
106     segmap_free,
107     segmap_fault,
108     segmap_faulta,
109     SEGMAP_BADOP(int),      /* setprot */
110     segmap_checkprot,
111     segmap_kluster,
112     SEGMAP_BADOP(size_t),   /* swapout */
112     SEGMAP_BADOP(int),      /* sync */
113     SEGMAP_BADOP(size_t),   /* incore */
114     SEGMAP_BADOP(int),      /* lockop */
115     segmap_getprot,
116     segmap_getoffset,
117     segmap_gettype,
118     segmap_getvp,
119     SEGMAP_BADOP(int),      /* advise */
120     segmap_dump,
121     segmap_pagelock,        /* pagelock */
122     SEGMAP_BADOP(int),      /* setpgsz */
123     segmap_getmemid,        /* getmemid */
124     segmap_getpolicy,        /* getpolicy */
125     segmap_capable,         /* capable */
126     seg_inherit_notsup      /* inherit */

```

new/usr/src/uts/common/vm/seg\_map.c

3

```
127 };  
unchanged_portion_omitted
```

\*\*\*\*\*

82652 Thu Feb 18 08:40:43 2016

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

6583 remove whole-process swapping

\*\*\*\*\*

\_\_\_\_\_ unchanged portion omitted \_\_\_\_\_

```

86 #define SEGSP_T_BADOP(t) (t(*)())segspt_badop

88 struct seg_ops segspt_ops = {
89     SEGSP_T_BADOP(int),          /* dup */
90     segspt_unmap,
91     segspt_free,
92     SEGSP_T_BADOP(int),          /* fault */
93     SEGSP_T_BADOP(faultcode_t), /* faulta */
94     SEGSP_T_BADOP(int),          /* setprot */
95     SEGSP_T_BADOP(int),          /* checkprot */
96     SEGSP_T_BADOP(int),          /* kluster */
97     SEGSP_T_BADOP(size_t),       /* swapout */
98     SEGSP_T_BADOP(int),          /* sync */
99     SEGSP_T_BADOP(size_t),       /* incore */
100    SEGSP_T_BADOP(int),          /* lockop */
101    SEGSP_T_BADOP(int),          /* getprot */
102    SEGSP_T_BADOP(u_offset_t),    /* getoffset */
103    SEGSP_T_BADOP(int),          /* gettype */
104    SEGSP_T_BADOP(int),          /* getvp */
105    SEGSP_T_BADOP(void),          /* advise */
106    SEGSP_T_BADOP(void),          /* dump */
107    SEGSP_T_BADOP(int),          /* pagelock */
108    SEGSP_T_BADOP(int),          /* setpgsz */
109    SEGSP_T_BADOP(int),          /* getmemid */
110    segspt_getpolicy,            /* getpolicy */
111    SEGSP_T_BADOP(int),          /* capable */
112    seg_inherit_notsup           /* inherit */
};

114 static int segspt_shmdup(struct seg *seg, struct seg *newseg);
115 static int segspt_shmunmap(struct seg *seg, caddr_t raddr, size_t ssize);
116 static void segspt_shmfree(struct seg *seg);
117 static faultcode_t segspt_shmfault(struct hat *hat, struct seg *seg,
118     caddr_t addr, size_t len, enum fault_type type, enum seg_rw rw);
119 static faultcode_t segspt_shmfaultra(struct seg *seg, caddr_t addr);
120 static int segspt_shmsetprot(register struct seg *seg, register caddr_t addr,
121     register size_t len, register uint_t prot);
122 static int segspt_shmcheckprot(struct seg *seg, caddr_t addr, size_t size,
123     uint_t prot);
124 static int segspt_shmkluster(struct seg *seg, caddr_t addr, ssize_t delta);
125 static size_t segspt_shmswapout(struct seg *seg);
126 static size_t segspt_shmincore(struct seg *seg, caddr_t addr, size_t len,
127     register char *vec);
128 static int segspt_shmsync(struct seg *seg, register caddr_t addr, size_t len,
129     int attr, uint_t flags);
130 static int segspt_shmlockop(struct seg *seg, caddr_t addr, size_t len,
131     int attr, int op, ulong_t *lockmap, size_t pos);
132 static int segspt_shmgetprot(struct seg *seg, caddr_t addr, size_t len,
133     uint_t *protv);
134 static u_offset_t segspt_shmgetoffset(struct seg *seg, caddr_t addr);
135 static int segspt_shmgettype(struct seg *seg, caddr_t addr);
136 static int segspt_shmgetvp(struct seg *seg, caddr_t addr, struct vnode **vpp);
137 static int segspt_shmadvise(struct seg *seg, caddr_t addr, size_t len,
138     uint_t behav);
139 static void segspt_shmdump(struct seg *seg);
140 static int segspt_shmpagelock(struct seg *, caddr_t, size_t,
141     struct page ***, enum lock_type, enum seg_rw);
142 static int segspt_shmsetpgsz(struct seg *, caddr_t, size_t, uint_t);
143 static int segspt_shmgetmemid(struct seg *, caddr_t, memid_t *);

```

```

143 static lgrp_mem_policy_info_t *segspt_shmgetpolicy(struct seg *, caddr_t);
144 static int segspt_shmcapable(struct seg *, segcapability_t);

```

```

146 struct seg_ops segspt_shmops = {
147     segspt_shmdup,
148     segspt_shmunmap,
149     segspt_shmfree,
150     segspt_shmfault,
151     segspt_shmfaultra,
152     segspt_shmsetprot,
153     segspt_shmcheckprot,
154     segspt_shmkluster,
155     segspt_shmswapout,
156     segspt_shmsync,
157     segspt_shmincore,
158     segspt_shmlockop,
159     segspt_shmgetprot,
160     segspt_shmgetoffset,
161     segspt_shmgettype,
162     segspt_shmgetvp,          /* advise */
163     segspt_shmdump,
164     segspt_shmpagelock,
165     segspt_shmsetpgsz,
166     segspt_shmgetmemid,
167     segspt_shmgetpolicy,
168     segspt_shmcapable,
169     seg_inherit_notsup
170 };

```

\_\_\_\_\_ unchanged portion omitted \_\_\_\_\_

```

2236 /*ARGSUSED*/
2237 static int
2238 segspt_shmkluster(struct seg *seg, caddr_t addr, ssize_t delta)
2239 {
2240     return (0);
2241 }

2242 /*ARGSUSED*/
2243 static size_t
2244 segspt_shmswapout(struct seg *seg)
2245 {
2246     return (0);
2247 }

2248 /*ARGSUSED*/
2249 static int
2250 segspt_shmgetpolicy(struct seg *seg, caddr_t addr)
2251 {
2252     return (0);
2253 }

```

\_\_\_\_\_ unchanged portion omitted \_\_\_\_\_

```

*****
279551 Thu Feb 18 08:40:43 2016
new/usr/src/uts/common/vm/seg_vn.c
6583 remove whole-process swapping
*****
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) 1986, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright 2015, Joyent, Inc. All rights reserved.
24 * Copyright 2015 Nexenta Systems, 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 - shared or copy-on-write from a vnode/anonymous memory.
42 */

44 #include <sys/types.h>
45 #include <sys/param.h>
46 #include <sys/t_lock.h>
47 #include <sys/errno.h>
48 #include <sys/system.h>
49 #include <sys/mman.h>
50 #include <sys/debug.h>
51 #include <sys/cred.h>
52 #include <sys/vmsystem.h>
53 #include <sys/tuneable.h>
54 #include <sys/bitmap.h>
55 #include <sys/swap.h>
56 #include <sys/kmem.h>
57 #include <sys/sysmacros.h>
58 #include <sys/vtrace.h>
59 #include <sys/cmn_err.h>
60 #include <sys/callb.h>
61 #include <sys/vm.h>

```

```

62 #include <sys/dumphdr.h>
63 #include <sys/lgrp.h>

65 #include <vm/hat.h>
66 #include <vm/as.h>
67 #include <vm/seg.h>
68 #include <vm/seg_vn.h>
69 #include <vm/pvn.h>
70 #include <vm/anon.h>
71 #include <vm/page.h>
72 #include <vm/vpage.h>
73 #include <sys/proc.h>
74 #include <sys/task.h>
75 #include <sys/project.h>
76 #include <sys/zone.h>
77 #include <sys/shm_impl.h>

79 /*
80 * segvn_fault needs a temporary page list array. To avoid calling kmem all
81 * the time, it creates a small (PVN_GETPAGE_NUM entry) array and uses it if
82 * it can. In the rare case when this page list is not large enough, it
83 * goes and gets a large enough array from kmem.
84 *
85 * This small page list array covers either 8 pages or 64kB worth of pages -
86 * whichever is smaller.
87 */
88 #define PVN_MAX_GETPAGE_SZ      0x10000
89 #define PVN_MAX_GETPAGE_NUM    0x8

91 #if PVN_MAX_GETPAGE_SZ > PVN_MAX_GETPAGE_NUM * PAGESIZE
92 #define PVN_GETPAGE_SZ      ptob(PVN_MAX_GETPAGE_NUM)
93 #define PVN_GETPAGE_NUM    PVN_MAX_GETPAGE_NUM
94 #else
95 #define PVN_GETPAGE_SZ      PVN_MAX_GETPAGE_SZ
96 #define PVN_GETPAGE_NUM    btop(PVN_MAX_GETPAGE_SZ)
97 #endif

99 /*
100 * Private seg op routines.
101 */
102 static int      segvn_dup(struct seg *seg, struct seg *newseg);
103 static int      segvn_unmap(struct seg *seg, caddr_t addr, size_t len);
104 static void      segvn_free(struct seg *seg);
105 static faultcode_t segvn_fault(struct hat *hat, struct seg *seg,
106                               caddr_t addr, size_t len, enum fault_type type,
107                               enum seg_rw rw);
108 static faultcode_t segvn_faulta(struct seg *seg, caddr_t addr);
109 static int      segvn_setprot(struct seg *seg, caddr_t addr,
110                               size_t len, uint_t prot);
111 static int      segvn_checkprot(struct seg *seg, caddr_t addr,
112                               size_t len, uint_t prot);
113 static int      segvn_kluster(struct seg *seg, caddr_t addr, ssize_t delta);
114 static size_t   segvn_swapout(struct seg *seg);
115 static int      segvn_sync(struct seg *seg, caddr_t addr, size_t len,
116                               int attr, uint_t flags);
116 static size_t   segvn_incore(struct seg *seg, caddr_t addr, size_t len,
117                               char *vec);
118 static int      segvn_lockop(struct seg *seg, caddr_t addr, size_t len,
119                               int attr, int op, ulong_t *lockmap, size_t pos);
120 static int      segvn_getprot(struct seg *seg, caddr_t addr, size_t len,
121                               uint_t *protv);
122 static u_offset_t segvn_getoffset(struct seg *seg, caddr_t addr);
123 static int      segvn_gettype(struct seg *seg, caddr_t addr);
124 static int      segvn_getvp(struct seg *seg, caddr_t addr,
125                               struct vnode **vpp);
126 static int      segvn_advise(struct seg *seg, caddr_t addr, size_t len,

```



```

127         uint_t behav);
128 static void    segvn_dump(struct seg *seg);
129 static int     segvn_pagelock(struct seg *seg, caddr_t addr, size_t len,
130                 struct page ***ppp, enum lock_type type, enum seg_rw rw);
131 static int     segvn_setpagesize(struct seg *seg, caddr_t addr, size_t len,
132                 uint_t szc);
133 static int     segvn_getmemid(struct seg *seg, caddr_t addr,
134                 memid_t *memidp);
135 static lgrp_mem_policy_info_t *segvn_getpolicy(struct seg *, caddr_t);
136 static int     segvn_capable(struct seg *seg, segcapability_t capable);
137 static int     segvn_inherit(struct seg *, caddr_t, size_t, uint_t);

```

```

139 struct    seg_ops segvn_ops = {
140     segvn_dup,
141     segvn_unmap,
142     segvn_free,
143     segvn_fault,
144     segvn_faulta,
145     segvn_setprot,
146     segvn_checkprot,
147     segvn_kluster,
148     segvn_swapout,
149     segvn_sync,
149     segvn_incore,
150     segvn_lockop,
151     segvn_getprot,
152     segvn_getoffset,
153     segvn_gettype,
154     segvn_getvp,
155     segvn_advise,
156     segvn_dump,
157     segvn_pagelock,
158     segvn_setpagesize,
159     segvn_getmemid,
160     segvn_getpolicy,
161     segvn_capable,
162     segvn_inherit
163 };

```

unchanged portion omitted

```

6953 /*
6954  * Check to see if it makes sense to do kluster/read ahead to
6955  * addr + delta relative to the mapping at addr.  We assume here
6956  * that delta is a signed PAGE_SIZE'd multiple (which can be negative).
6957  *
6958  * For segvn, we currently "approve" of the action if we are
6959  * still in the segment and it maps from the same vp/off,
6960  * or if the advice stored in segvn_data or vpages allows it.
6961  * Currently, klustering is not allowed only if MADV_RANDOM is set.
6962  */
6963 static int
6964 segvn_kluster(struct seg *seg, caddr_t addr, ssize_t delta)
6965 {
6966     struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6967     struct anon *oap, *ap;
6968     ssize_t pd;
6969     size_t page;
6970     struct vnode *vp1, *vp2;
6971     u_offset_t off1, off2;
6972     struct anon_map *amp;

6974     ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));
6975     ASSERT(AS_WRITE_HELD(seg->s_as) ||
6976            SEGVN_LOCK_HELD(seg->s_as, &svd->lock));

6978     if (addr + delta < seg->s_base ||

```

```

6979         addr + delta >= (seg->s_base + seg->s_size))
6980             return (-1);          /* exceeded segment bounds */

6982     pd = delta / (ssize_t)PAGE_SIZE; /* divide to preserve sign bit */
6983     page = seg_page(seg, addr);

6985     /*
6986     * Check to see if either of the pages addr or addr + delta
6987     * have advice set that prevents klustering (if MADV_RANDOM advice
6988     * is set for entire segment, or MADV_SEQUENTIAL is set and delta
6989     * is negative).
6990     */
6991     if (svd->advice == MADV_RANDOM ||
6992         svd->advice == MADV_SEQUENTIAL && delta < 0)
6993         return (-1);
6994     else if (svd->pageadvice && svd->vpage) {
6995         struct vpage *bvpp, *evpp;

6997         bvpp = &svd->vpage[page];
6998         evpp = &svd->vpage[page + pd];
6999         if (VPP_ADVICE(bvpp) == MADV_RANDOM ||
7000             VPP_ADVICE(evpp) == MADV_SEQUENTIAL && delta < 0)
7001             return (-1);
7002         if (VPP_ADVICE(bvpp) != VPP_ADVICE(evpp) &&
7003             VPP_ADVICE(evpp) == MADV_RANDOM)
7004             return (-1);
7005     }

7007     if (svd->type == MAP_SHARED)
7008         return (0);          /* shared mapping - all ok */

7010     if ((amp = svd->amp) == NULL)
7011         return (0);          /* off original vnode */

7013     page += svd->anon_index;

7015     ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);

7017     oap = anon_get_ptr(amp->ahp, page);
7018     ap = anon_get_ptr(amp->ahp, page + pd);

7020     ANON_LOCK_EXIT(&amp->a_rwlock);

7022     if ((oap == NULL && ap != NULL) || (oap != NULL && ap == NULL)) {
7023         return (-1);          /* one with and one without an anon */
7024     }

7026     if (oap == NULL) {
7027         return (0);          /* implies that ap == NULL */
7028     }

7030     /*
7031     * Now we know we have two anon pointers - check to
7032     * see if they happen to be properly allocated.
7033     */

7035     /*
7036     * XXX We cheat here and don't lock the anon slots. We can't because
7037     * we may have been called from the anon layer which might already
7038     * have locked them. We are holding a refcnt on the slots so they
7039     * can't disappear. The worst that will happen is we'll get the wrong
7040     * names (vp, off) for the slots and make a poor klustering decision.
7041     */
7042     swap_xlate(ap, &vp1, &off1);
7043     swap_xlate(oap, &vp2, &off2);

```

```

7046     if (!VOP_CMP(vp1, vp2, NULL) || off1 - off2 != delta)
7047         return (-1);
7048     return (0);
7051 }

7053 /*
7054  * Swap the pages of seg out to secondary storage, returning the
7055  * number of bytes of storage freed.
7056  *
7057  * The basic idea is first to unload all translations and then to call
7058  * VOP_PUTPAGE() for all newly-unmapped pages, to push them out to the
7059  * swap device. Pages to which other segments have mappings will remain
7060  * mapped and won't be swapped. Our caller (as_swapout) has already
7061  * performed the unloading step.
7062  *
7063  * The value returned is intended to correlate well with the process's
7064  * memory requirements. However, there are some caveats:
7065  * 1) When given a shared segment as argument, this routine will
7066  *    only succeed in swapping out pages for the last sharer of the
7067  *    segment. (Previous callers will only have decremented mapping
7068  *    reference counts.)
7069  * 2) We assume that the hat layer maintains a large enough translation
7070  *    cache to capture process reference patterns.
7071  */
7072 static size_t
7073 segvn_swapout(struct seg *seg)
7074 {
7075     struct segvn_data *svd = (struct segvn_data *)seg->s_data;
7076     struct anon_map *amp;
7077     pgcnt_t pgcnt = 0;
7078     pgcnt_t npages;
7079     pgcnt_t page;
7080     ulong_t anon_index;

7082     ASSERT(seg->s_as && AS_LOCK_HELD(seg->s_as));

7084     SEGVN_LOCK_ENTER(seg->s_as, &svd->lock, RW_READER);
7085     /*
7086      * Find pages unmapped by our caller and force them
7087      * out to the virtual swap device.
7088      */
7089     if ((amp = svd->amp) != NULL)
7090         anon_index = svd->anon_index;
7091     npages = seg->s_size >> PAGESHIFT;
7092     for (page = 0; page < npages; page++) {
7093         page_t *pp;
7094         struct anon *ap;
7095         struct vnode *vp;
7096         u_offset_t off;
7097         anon_sync_obj_t cookie;

7099         /*
7100          * Obtain <vp, off> pair for the page, then look it up.
7101          *
7102          * Note that this code is willing to consider regular
7103          * pages as well as anon pages. Is this appropriate here?
7104          */
7105         ap = NULL;
7106         if (amp != NULL) {
7107             ANON_LOCK_ENTER(&amp->a_rwlock, RW_READER);
7108             if (anon_array_try_enter(amp, anon_index + page,
7109                 &cookie)) {
7110                 ANON_LOCK_EXIT(&amp->a_rwlock);
7111                 continue;
7112             }

```

```

7113         ap = anon_get_ptr(amp->ahp, anon_index + page);
7114         if (ap != NULL) {
7115             swap_xlate(ap, &vp, &off);
7116         } else {
7117             vp = svd->vp;
7118             off = svd->offset + ptob(page);
7119         }
7120         anon_array_exit(&cookie);
7121         ANON_LOCK_EXIT(&amp->a_rwlock);
7122     } else {
7123         vp = svd->vp;
7124         off = svd->offset + ptob(page);
7125     }
7126     if (vp == NULL) { /* untouched zfod page */
7127         ASSERT(ap == NULL);
7128         continue;
7129     }

7131     pp = page_lookup_nowait(vp, off, SE_SHARED);
7132     if (pp == NULL)
7133         continue;

7136     /*
7137      * Examine the page to see whether it can be tossed out,
7138      * keeping track of how many we've found.
7139      */
7140     if (!page_tryupgrade(pp)) {
7141         /*
7142          * If the page has an i/o lock and no mappings,
7143          * it's very likely that the page is being
7144          * written out as a result of klustering.
7145          * Assume this is so and take credit for it here.
7146          */
7147         if (!page_io_trylock(pp)) {
7148             if (!hat_page_is_mapped(pp))
7149                 pgcnt++;
7150         } else {
7151             page_io_unlock(pp);
7152         }
7153         page_unlock(pp);
7154         continue;
7155     }
7156     ASSERT(!page_iolock_assert(pp));

7159     /*
7160      * Skip if page is locked or has mappings.
7161      * We don't need the page_struct_lock to look at lkcncnt
7162      * and cowcnt because the page is exclusive locked.
7163      */
7164     if (pp->p_lkcncnt != 0 || pp->p_cowcnt != 0 ||
7165         hat_page_is_mapped(pp)) {
7166         page_unlock(pp);
7167         continue;
7168     }

7170     /*
7171      * dispose skips large pages so try to demote first.
7172      */
7173     if (pp->p_szc != 0 && !page_try_demote_pages(pp)) {
7174         page_unlock(pp);
7175         /*
7176          * XXX should skip the remaining page_t's of this
7177          * large page.
7178          */

```

```

7179         continue;
7180     }

7182     ASSERT(pp->p_szc == 0);

7184     /*
7185     * No longer mapped -- we can toss it out. How
7186     * we do so depends on whether or not it's dirty.
7187     */
7188     if (hat_ismod(pp) && pp->p_vnode) {
7189         /*
7190         * We must clean the page before it can be
7191         * freed. Setting B_FREE will cause pvn_done
7192         * to free the page when the i/o completes.
7193         * XXX: This also causes it to be accounted
7194         *       as a pageout instead of a swap: need
7195         *       B_SWAPOUT bit to use instead of B_FREE.
7196         *
7197         * Hold the vnode before releasing the page lock
7198         * to prevent it from being freed and re-used by
7199         * some other thread.
7200         */
7201         VN_HOLD(vp);
7202         page_unlock(pp);

7204         /*
7205         * Queue all i/o requests for the pageout thread
7206         * to avoid saturating the pageout devices.
7207         */
7208         if (!queue_io_request(vp, off))
7209             VN_RELE(vp);
7210     } else {
7211         /*
7212         * The page was clean, free it.
7213         *
7214         * XXX: Can we ever encounter modified pages
7215         *       with no associated vnode here?
7216         */
7217         ASSERT(pp->p_vnode != NULL);
7218         /*LINTED: constant in conditional context*/
7219         VN_DISPOSE(pp, B_FREE, 0, kcred);
7220     }

7222     /*
7223     * Credit now even if i/o is in progress.
7224     */
7225     pgcnt++;
7226 }
7227 SEGVN_LOCK_EXIT(seg->s_as, &svd->lock);

7229     /*
7230     * Wakeup pageout to initiate i/o on all queued requests.
7231     */
7232     cv_signal_pageout();
7233     return (ptob(pgcnt));
7049 }
_____unchanged_portion_omitted_

```

```

*****
89741 Thu Feb 18 08:40:43 2016
new/usr/src/uts/common/vm/vm_anon.c
6583 remove whole-process swapping
*****
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) 1986, 2010, Oracle and/or its affiliates. All rights reserved.
23 */

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

28 /*
29 * University Copyright- Copyright (c) 1982, 1986, 1988
30 * The Regents of the University of California
31 * All Rights Reserved
32 *
33 * University Acknowledgment- Portions of this document are derived from
34 * software developed by the University of California, Berkeley, and its
35 * contributors.
36 */

38 /*
39 * VM - anonymous pages.
40 *
41 * This layer sits immediately above the vm_swap layer.  It manages
42 * physical pages that have no permanent identity in the file system
43 * name space, using the services of the vm_swap layer to allocate
44 * backing storage for these pages.  Since these pages have no external
45 * identity, they are discarded when the last reference is removed.
46 *
47 * An important function of this layer is to manage low-level sharing
48 * of pages that are logically distinct but that happen to be
49 * physically identical (e.g., the corresponding pages of the processes
50 * resulting from a fork before one process or the other changes their
51 * contents).  This pseudo-sharing is present only as an optimization
52 * and is not to be confused with true sharing in which multiple
53 * address spaces deliberately contain references to the same object;
54 * such sharing is managed at a higher level.
55 *
56 * The key data structure here is the anon struct, which contains a
57 * reference count for its associated physical page and a hint about
58 * the identity of that page.  Anon structs typically live in arrays,
59 * with an instance's position in its array determining where the
60 * corresponding backing storage is allocated; however, the swap_xlate()
61 * routine abstracts away this representation information so that the

```

```

62 * rest of the anon layer need not know it.  (See the swap layer for
63 * more details on anon struct layout.)
64 *
65 * In the future versions of the system, the association between an
66 * anon struct and its position on backing store will change so that
67 * we don't require backing store all anonymous pages in the system.
68 * This is important for consideration for large memory systems.
69 * We can also use this technique to delay binding physical locations
70 * to anonymous pages until pageout time where we can make smarter
71 * allocation decisions to improve anonymous klustering.
72 * to anonymous pages until pageout/swapout time where we can make
73 * smarter allocation decisions to improve anonymous klustering.
74 *
75 * Many of the routines defined here take a (struct anon **) argument,
76 * which allows the code at this level to manage anon pages directly,
77 * so that callers can regard anon structs as opaque objects and not be
78 * concerned with assigning or inspecting their contents.
79 *
80 * Clients of this layer refer to anon pages indirectly.  That is, they
81 * maintain arrays of pointers to anon structs rather than maintaining
82 * anon structs themselves.  The (struct anon **) arguments mentioned
83 * above are pointers to entries in these arrays.  It is these arrays
84 * that capture the mapping between offsets within a given segment and
85 * the corresponding anonymous backing storage address.
86 */

86 #ifdef DEBUG
87 #define ANON_DEBUG
88 #endif

90 #include <sys/types.h>
91 #include <sys/t_lock.h>
92 #include <sys/param.h>
93 #include <sys/system.h>
94 #include <sys/mman.h>
95 #include <sys/cred.h>
96 #include <sys/thread.h>
97 #include <sys/vnode.h>
98 #include <sys/cpuvar.h>
99 #include <sys/swap.h>
100 #include <sys/cmn_err.h>
101 #include <sys/vtrace.h>
102 #include <sys/kmem.h>
103 #include <sys/sysmacros.h>
104 #include <sys/bitmap.h>
105 #include <sys/vmsystem.h>
106 #include <sys/tuneable.h>
107 #include <sys/debug.h>
108 #include <sys/fs/swapnode.h>
109 #include <sys/tnf_probe.h>
110 #include <sys/lgrp.h>
111 #include <sys/policy.h>
112 #include <sys/condvar_impl.h>
113 #include <sys/mutex_impl.h>
114 #include <sys/rctl.h>

116 #include <vm/as.h>
117 #include <vm/hat.h>
118 #include <vm/anon.h>
119 #include <vm/page.h>
120 #include <vm/vpage.h>
121 #include <vm/seg.h>
122 #include <vm/zm.h>

124 #include <fs/fs_subr.h>

```

```

126 struct vnode *anon_vp;

128 int anon_debug;

130 kmutex_t      anoninfo_lock;
131 struct        k_anoninfo k_anoninfo;
132 ani_free_t     *ani_free_pool;
133 pad_mutex_t    anon_array_lock[ANON_LOCKSIZE];
134 kcondvar_t     anon_array_cv[ANON_LOCKSIZE];

136 /*
137  * Global hash table for (vp, off) -> anon slot
138  */
139 extern int swap_maxcontig;
140 size_t anon_hash_size;
141 unsigned int anon_hash_shift;
142 struct anon **anon_hash;

144 static struct kmem_cache *anon_cache;
145 static struct kmem_cache *anonmap_cache;

147 pad_mutex_t     *anonhash_lock;

149 /*
150  * Used to make the increment of all refcnts of all anon slots of a large
151  * page appear to be atomic. The lock is grabbed for the first anon slot of
152  * a large page.
153  */
154 pad_mutex_t     *anonpages_hash_lock;

156 #define APH_MUTEX(vp, off) \
157     (&anonpages_hash_lock[(ANON_HASH((vp), (off)) & \
158         (AH_LOCK_SIZE - 1))].pad_mutex)

160 #ifdef VM_STATS
161 static struct anonvmstats_str {
162     ulong_t  getpages[30];
163     ulong_t  privatepages[10];
164     ulong_t  demotepages[9];
165     ulong_t  decrefpages[9];
166     ulong_t  dupfillholes[4];
167     ulong_t  freepages[1];
168 } anonvmstats;
169 unchanged portion omitted

```

```

3551 void
3552 anon_array_enter(struct anon_map *amp, ulong_t an_idx, anon_sync_obj_t *sobj)
3553 {
3554     ulong_t      *ap_slot;
3555     kmutex_t      *mtx;
3556     kcondvar_t    *cv;
3557     int           hash;

3559     /*
3560     * Use szc to determine anon slot(s) to appear atomic.
3561     * If szc = 0, then lock the anon slot and mark it busy.
3562     * If szc > 0, then lock the range of slots by getting the
3563     * anon_array_lock for the first anon slot, and mark only the
3564     * first anon slot busy to represent whole range being busy.
3565     */

3567     ASSERT(RW_READ_HELD(&amp->a_rwlock));
3568     an_idx = P2ALIGN(an_idx, page_get_pagecnt(amp->a_szc));
3569     hash = ANON_ARRAY_HASH(amp, an_idx);
3570     sobj->sync_mutex = mtx = &anon_array_lock[hash].pad_mutex;
3571     sobj->sync_cv = cv = &anon_array_cv[hash];

```

```

3572     mutex_enter(mtx);
3573     ap_slot = anon_get_slot(amp->ahp, an_idx);
3574     while (ANON_ISBUSY(ap_slot))
3575         cv_wait(cv, mtx);
3576     ANON_SETBUSY(ap_slot);
3577     sobj->sync_data = ap_slot;
3578     mutex_exit(mtx);
3579 }

3581 int
3582 anon_array_try_enter(struct anon_map *amp, ulong_t an_idx,
3583     anon_sync_obj_t *sobj)
3584 {
3585     ulong_t      *ap_slot;
3586     kmutex_t      *mtx;
3587     int           hash;

3589     /*
3590     * Try to lock a range of anon slots.
3591     * Use szc to determine anon slot(s) to appear atomic.
3592     * If szc = 0, then lock the anon slot and mark it busy.
3593     * If szc > 0, then lock the range of slots by getting the
3594     * anon_array_lock for the first anon slot, and mark only the
3595     * first anon slot busy to represent whole range being busy.
3596     * Fail if the mutex or the anon_array are busy.
3597     */

3599     ASSERT(RW_READ_HELD(&amp->a_rwlock));
3600     an_idx = P2ALIGN(an_idx, page_get_pagecnt(amp->a_szc));
3601     hash = ANON_ARRAY_HASH(amp, an_idx);
3602     sobj->sync_mutex = mtx = &anon_array_lock[hash].pad_mutex;
3603     sobj->sync_cv = &anon_array_cv[hash];
3604     if (!mutex_tryenter(mtx)) {
3605         return (EWOULDBLOCK);
3606     }
3607     ap_slot = anon_get_slot(amp->ahp, an_idx);
3608     if (ANON_ISBUSY(ap_slot)) {
3609         mutex_exit(mtx);
3610         return (EWOULDBLOCK);
3611     }
3612     ANON_SETBUSY(ap_slot);
3613     sobj->sync_data = ap_slot;
3614     mutex_exit(mtx);
3615     return (0);
3616 }

```

unchanged portion omitted

```

*****
88974 Thu Feb 18 08:40:43 2016
new/usr/src/uts/common/vm/vm_as.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

2052 /*
2053  * Swap the pages associated with the address space as out to
2054  * secondary storage, returning the number of bytes actually
2055  * swapped.
2056  *
2057  * The value returned is intended to correlate well with the process's
2058  * memory requirements. Its usefulness for this purpose depends on
2059  * how well the segment-level routines do at returning accurate
2060  * information.
2061  */
2062 size_t
2063 as_swapout(struct as *as)
2064 {
2065     struct seg *seg;
2066     size_t swpcnt = 0;

2068     /*
2069     * Kernel-only processes have given up their address
2070     * spaces. Of course, we shouldn't be attempting to
2071     * swap out such processes in the first place...
2072     */
2073     if (as == NULL)
2074         return (0);

2076     AS_LOCK_ENTER(as, RW_READER);

2078     /*
2079     * Free all mapping resources associated with the address
2080     * space. The segment-level swapout routines capitalize
2081     * on this unmapping by scavenging pages that have become
2082     * unmapped here.
2083     */
2084     hat_swapout(as->a_hat);

2086     /*
2087     * Call the swapout routines of all segments in the address
2088     * space to do the actual work, accumulating the amount of
2089     * space reclaimed.
2090     */
2091     for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {
2092         struct seg_ops *ov = seg->s_ops;

2094         /*
2095         * We have to check to see if the seg has
2096         * an ops vector because the seg may have
2097         * been in the middle of being set up when
2098         * the process was picked for swapout.
2099         */
2100         if ((ov != NULL) && (ov->swapout != NULL))
2101             swpcnt += SEGOP_SWAPOUT(seg);
2102     }
2103     AS_LOCK_EXIT(as);
2104     return (swpcnt);
2105 }

2107 /*
2053  * Determine whether data from the mappings in interval [addr, addr + size)
2054  * are in the primary memory (core) cache.
2055  */

```

```

2056 int
2057 as_incore(struct as *as, caddr_t addr,
2058           size_t size, char *vec, size_t *sizep)
2059 {
2060     struct seg *seg;
2061     size_t ssize;
2062     caddr_t raddr;          /* rounded down addr */
2063     size_t rsize;          /* rounded up size */
2064     size_t isize;          /* iteration size */
2065     int error = 0;         /* result, assume success */

2067     *sizep = 0;
2068     raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2069     rsize = (((size_t)addr + size) + PAGEOFFSET) & PAGEMASK -
2070             (size_t)raddr;

2072     if (raddr + rsize < raddr)          /* check for wraparound */
2073         return (ENOMEM);

2075     AS_LOCK_ENTER(as, RW_READER);
2076     seg = as_segat(as, raddr);
2077     if (seg == NULL) {
2078         AS_LOCK_EXIT(as);
2079         return (-1);
2080     }

2082     for (; rsize != 0; rsize -= ssize, raddr += ssize) {
2083         if (raddr >= seg->s_base + seg->s_size) {
2084             seg = AS_SEGNEXT(as, seg);
2085             if (seg == NULL || raddr != seg->s_base) {
2086                 error = -1;
2087                 break;
2088             }
2089         }
2090         if ((raddr + rsize) > (seg->s_base + seg->s_size))
2091             ssize = seg->s_base + seg->s_size - raddr;
2092         else
2093             ssize = rsize;
2094         *sizep += isize = SEGOP_INCORE(seg, raddr, ssize, vec);
2095         if (isize != ssize) {
2096             error = -1;
2097             break;
2098         }
2099         vec += btopr(ssize);
2100     }
2101     AS_LOCK_EXIT(as);
2102     return (error);
2103 }
_____unchanged_portion_omitted_____

```

new/usr/src/uts/i86pc/os/mlsetup.c

1

```
*****
13931 Thu Feb 18 08:40:44 2016
new/usr/src/uts/i86pc/os/mlsetup.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

96 /*
97  * Setup routine called right before main(). Interposing this function
98  * before main() allows us to call it in a machine-independent fashion.
99  */
100 void
101 mlsetup(struct regs *rp)
102 {
103     u_longlong_t prop_value;
104     extern struct classfuncs sys_classfuncs;
105     extern disp_t cpu0_disp;
106     extern char t0stack[];
107     extern int post_fastreboot;
108     extern uint64_t plat_dr_options;

110     ASSERT_STACK_ALIGNED();

112     /*
113      * initialize cpu_self
114      */
115     cpu[0]->cpu_self = cpu[0];

117 #if defined(__xpv)
118     /*
119      * Point at the hypervisor's virtual cpu structure
120      */
121     cpu[0]->cpu_m.mcpu_vcpu_info = &HYPERVISOR_shared_info->vcpu_info[0];
122 #endif

124     /*
125      * check if we've got special bits to clear or set
126      * when checking cpu features
127      */

129     if (bootprop_getval("cpuid_feature_ecx_include", &prop_value) != 0)
130         cpuid_feature_ecx_include = 0;
131     else
132         cpuid_feature_ecx_include = (uint32_t)prop_value;

134     if (bootprop_getval("cpuid_feature_ecx_exclude", &prop_value) != 0)
135         cpuid_feature_ecx_exclude = 0;
136     else
137         cpuid_feature_ecx_exclude = (uint32_t)prop_value;

139     if (bootprop_getval("cpuid_feature_edx_include", &prop_value) != 0)
140         cpuid_feature_edx_include = 0;
141     else
142         cpuid_feature_edx_include = (uint32_t)prop_value;

144     if (bootprop_getval("cpuid_feature_edx_exclude", &prop_value) != 0)
145         cpuid_feature_edx_exclude = 0;
146     else
147         cpuid_feature_edx_exclude = (uint32_t)prop_value;

149     /*
150      * Initialize idt0, gdt0, ldt0_default, ktss0 and dftss.
151      */
152     init_desctbls();
```

new/usr/src/uts/i86pc/os/mlsetup.c

2

```
154     /*
155      * lgrp_init() and possibly cpuid_pass1() need PCI config
156      * space access
157      */
158 #if defined(__xpv)
159     if (DOMAIN_IS_INITDOMAIN(xen_info))
160         pci_cfgspace_init();
161 #else
162     pci_cfgspace_init();
163     /*
164      * Initialize the platform type from CPU 0 to ensure that
165      * determine_platform() is only ever called once.
166      */
167     determine_platform();
168 #endif

170     /*
171      * The first lightweight pass (pass0) through the cpuid data
172      * was done in locore before mlsetup was called. Do the next
173      * pass in C code.
174      *
175      * The x86_featureset is initialized here based on the capabilities
176      * of the boot CPU. Note that if we choose to support CPUs that have
177      * different feature sets (at which point we would almost certainly
178      * want to set the feature bits to correspond to the feature
179      * minimum) this value may be altered.
180      */
181     cpuid_pass1(cpu[0], x86_featureset);

183 #if !defined(__xpv)
184     if ((get_hwenv() & HW_XEN_HVM) != 0)
185         xen_hvm_init();

187     /*
188      * Before we do anything with the TSCs, we need to work around
189      * Intel erratum BT81. On some CPUs, warm reset does not
190      * clear the TSC. If we are on such a CPU, we will clear TSC ourselves
191      * here. Other CPUs will clear it when we boot them later, and the
192      * resulting skew will be handled by tsc_sync_master()/_slave();
193      * note that such skew already exists and has to be handled anyway.
194      *
195      * We do this only on metal. This same problem can occur with a
196      * hypervisor that does not happen to virtualise a TSC that starts from
197      * zero, regardless of CPU type; however, we do not expect hypervisors
198      * that do not virtualise TSC that way to handle writes to TSC
199      * correctly, either.
200      */
201     if (get_hwenv() == HW_NATIVE &&
202         cpuid_getvendor(CPU) == X86_VENDOR_Intel &&
203         cpuid_getfamily(CPU) == 6 &&
204         (cpuid_getmodel(CPU) == 0x2d || cpuid_getmodel(CPU) == 0x3e) &&
205         is_x86_feature(x86_featureset, X86FSET_TSC)) {
206         (void) wrmsr(REG_TSC, 0UL);
207     }

209     /*
210      * Patch the tsc_read routine with appropriate set of instructions,
211      * depending on the processor family and architecture, to read the
212      * time-stamp counter while ensuring no out-of-order execution.
213      * Patch it while the kernel text is still writable.
214      *
215      * Note: tsc_read is not patched for intel processors whose family
216      * is >6 and for amd whose family >f (in case they don't support rdtscp
217      * instruction, unlikely). By default tsc_read will use cpuid for
218      * serialization in such cases. The following code needs to be
219      * revisited if intel processors of family >= f retains the
```

```

220  * instruction serialization nature of mfence instruction.
221  * Note: tsc_read is not patched for x86 processors which do
222  * not support "mfence". By default tsc_read will use cpuid for
223  * serialization in such cases.
224  *
225  * The Xen hypervisor does not correctly report whether rdtscp is
226  * supported or not, so we must assume that it is not.
227  */
228  if ((get_hwenv() & HW_XEN_HVM) == 0 &&
229      is_x86_feature(x86_featureset, X86FSET_TSCP))
230      patch_tsc_read(X86_HAVE_TSCP);
231  else if (cpuid_getvendor(CPU) == X86_VENDOR_AMD &&
232           cpuid_getfamily(CPU) <= 0xf &&
233           is_x86_feature(x86_featureset, X86FSET_SSE2))
234      patch_tsc_read(X86_TSC_MFENCE);
235  else if (cpuid_getvendor(CPU) == X86_VENDOR_Intel &&
236           cpuid_getfamily(CPU) <= 6 &&
237           is_x86_feature(x86_featureset, X86FSET_SSE2))
238      patch_tsc_read(X86_TSC_LFENCE);

240 #endif /* !__xpv */

242 #if defined(__i386) && !defined(__xpv)
243     /*
244      * Some i386 processors do not implement the rdtsc instruction,
245      * or at least they do not implement it correctly. Patch them to
246      * return 0.
247      */
248     if (!is_x86_feature(x86_featureset, X86FSET_TSC))
249         patch_tsc_read(X86_NO_TSC);
250 #endif /* __i386 && !__xpv */

252 #if defined(__amd64) && !defined(__xpv)
253     patch_memops(cpuid_getvendor(CPU));
254 #endif /* __amd64 && !__xpv */

256 #if !defined(__xpv)
257     /* XXPV what, if anything, should be dorked with here under xen? */

259     /*
260      * While we're thinking about the TSC, let's set up %cr4 so that
261      * userland can issue rdtsc, and initialize the TSC_AUX value
262      * (the cpuid) for the rdtscp instruction on appropriately
263      * capable hardware.
264      */
265     if (is_x86_feature(x86_featureset, X86FSET_TSC))
266         setcr4(getcr4() & ~CR4_TSD);

268     if (is_x86_feature(x86_featureset, X86FSET_TSCP))
269         (void) wrmsr(MSR_AMD_TSCAUX, 0);

271     /*
272      * Let's get the other %cr4 stuff while we're here.
273      */
274     if (is_x86_feature(x86_featureset, X86FSET_DE))
275         setcr4(getcr4() | CR4_DE);

277     if (is_x86_feature(x86_featureset, X86FSET_SMEP))
278         setcr4(getcr4() | CR4_SMEP);
279 #endif /* __xpv */

281     /*
282      * initialize t0
283      */
284     t0.t_stk = (caddr_t)rp - MINFRAME;
285     t0.t_stkbase = t0stack;

```

```

286     t0.t_pri = maxclsypri - 3;
287     t0.t_schedflag = 0;
288     t0.t_schedflag = TS_LOAD | TS_DONT_SWAP;
289     t0.t_procp = &p0;
290     t0.t_plockp = &p0lock.pl_lock;
291     t0.t_lwp = &lwp0;
292     t0.t_forw = &t0;
293     t0.t_back = &t0;
294     t0.t_next = &t0;
295     t0.t_prev = &t0;
296     t0.t_cpu = cpu[0];
297     t0.t_disp_queue = &cpu0_disp;
298     t0.t_bind_cpu = PBIND_NONE;
299     t0.t_bind_pset = PS_NONE;
300     t0.t_bindflag = (uchar_t)default_binding_mode;
301     t0.t_cpupart = &cp_default;
302     t0.t_clfuncs = &sys_classfuncs.thread;
303     t0.t_copyops = NULL;
304     THREAD_ONPROC(&t0, CPU);

305     lwp0.lwp_thread = &t0;
306     lwp0.lwp_regs = (void *)rp;
307     lwp0.lwp_procp = &p0;
308     t0.t_tid = p0.p_lwpcnt = p0.p_lwprcnt = p0.p_lwpid = 1;

310     p0.p_exec = NULL;
311     p0.p_stat = SRUN;
312     p0.p_flag = SSYS;
313     p0.p_tlist = &t0;
314     p0.p_stksize = 2*PAGESIZE;
315     p0.p_stkpageszc = 0;
316     p0.p_as = &kas;
317     p0.p_lockp = &p0lock;
318     p0.p_brkpageszc = 0;
319     p0.p_tl_lgrpid = LGRP_NONE;
320     p0.p_tr_lgrpid = LGRP_NONE;
321     sigorset(&p0.p_ignore, &ignoredefault);

323     CPU->cpu_thread = &t0;
324     bzero(&cpu0_disp, sizeof (disp_t));
325     CPU->cpu_disp = &cpu0_disp;
326     CPU->cpu_disp->disp_cpu = CPU;
327     CPU->cpu_dispthread = &t0;
328     CPU->cpu_idle_thread = &t0;
329     CPU->cpu_flags = CPU_READY | CPU_RUNNING | CPU_EXISTS | CPU_ENABLE;
330     CPU->cpu_dispatch_pri = t0.t_pri;

332     CPU->cpu_id = 0;

334     CPU->cpu_pri = 12; /* initial PIL for the boot CPU */

336     /*
337      * The kernel doesn't use LDTs unless a process explicitly requests one.
338      */
339     p0.p_ldt_desc = null_sdesc;

341     /*
342      * Initialize thread/cpu microstate accounting
343      */
344     init_mstate(&t0, LMS_SYSTEM);
345     init_cpu_mstate(CPU, CMS_SYSTEM);

347     /*
348      * Initialize lists of available and active CPUs.
349      */
350     cpu_list_init(CPU);

```



```

352     pg_cpu_bootstrap(CPU);
353
354     /*
355      * Now that we have taken over the GDT, IDT and have initialized
356      * active CPU list it's time to inform kmdb if present.
357      */
358     if (boothowto & RB_DEBUG)
359         kdi_idt_sync();
360
361     /*
362      * Explicitly set console to text mode (0x3) if this is a boot
363      * post Fast Reboot, and the console is set to CONS_SCREEN_TEXT.
364      */
365     if (post_fastreboot && boot_console_type(NULL) == CONS_SCREEN_TEXT)
366         set_console_mode(0x3);
367
368     /*
369      * If requested (boot -d) drop into kmdb.
370      *
371      * This must be done after cpu_list_init() on the 64-bit kernel
372      * since taking a trap requires that we re-compute gsbase based
373      * on the cpu list.
374      */
375     if (boothowto & RB_DEBUGENTER)
376         kmdb_enter();
377
378     cpu_vm_data_init(CPU);
379
380     rp->r_fp = 0; /* terminate kernel stack traces! */
381
382     prom_init("kernel", (void *)NULL);
383
384     /* User-set option overrides firmware value. */
385     if (bootprop_getval(PLAT_DR_OPTIONS_NAME, &prop_value) == 0) {
386         plat_dr_options = (uint64_t)prop_value;
387     }
388     #if defined(__xpv)
389     /* No support of DR operations on xpv */
390     plat_dr_options = 0;
391     #else /* __xpv */
392     /* Flag PLAT_DR_FEATURE_ENABLED should only be set by DR driver. */
393     plat_dr_options &= ~PLAT_DR_FEATURE_ENABLED;
394     #ifndef __amd64
395     /* Only enable CPU/memory DR on 64 bits kernel. */
396     plat_dr_options &= ~PLAT_DR_FEATURE_MEMORY;
397     plat_dr_options &= ~PLAT_DR_FEATURE_CPU;
398     #endif /* __amd64 */
399     #endif /* __xpv */
400
401     /*
402      * Get value of "plat_dr_physmax" boot option.
403      * It overrides values calculated from MSCT or SRAT table.
404      */
405     if (bootprop_getval(PLAT_DR_PHYSMAX_NAME, &prop_value) == 0) {
406         plat_dr_physmax = ((uint64_t)prop_value) >> PAGESHIFT;
407     }
408
409     /* Get value of boot_ncpus. */
410     if (bootprop_getval(BOOT_NCPUS_NAME, &prop_value) != 0) {
411         boot_ncpus = NCPU;
412     } else {
413         boot_ncpus = (int)prop_value;
414         if (boot_ncpus <= 0 || boot_ncpus > NCPU)
415             boot_ncpus = NCPU;
416     }

```

```

418     /*
419      * Set max_ncpus and boot_max_ncpus to boot_ncpus if platform doesn't
420      * support CPU DR operations.
421      */
422     if (plat_dr_support_cpu() == 0) {
423         max_ncpus = boot_max_ncpus = boot_ncpus;
424     } else {
425         if (bootprop_getval(PLAT_MAX_NCPUS_NAME, &prop_value) != 0) {
426             max_ncpus = NCPU;
427         } else {
428             max_ncpus = (int)prop_value;
429             if (max_ncpus <= 0 || max_ncpus > NCPU) {
430                 max_ncpus = NCPU;
431             }
432             if (boot_ncpus > max_ncpus) {
433                 boot_ncpus = max_ncpus;
434             }
435         }
436     }
437
438     if (bootprop_getval(BOOT_MAX_NCPUS_NAME, &prop_value) != 0) {
439         boot_max_ncpus = boot_ncpus;
440     } else {
441         boot_max_ncpus = (int)prop_value;
442         if (boot_max_ncpus <= 0 || boot_max_ncpus > NCPU) {
443             boot_max_ncpus = boot_ncpus;
444         } else if (boot_max_ncpus > max_ncpus) {
445             boot_max_ncpus = max_ncpus;
446         }
447     }
448
449     /*
450      * Initialize the lgrp framework
451      */
452     lgrp_init(LGRP_INIT_STAGE1);
453
454     if (boothowto & RB_HALT) {
455         prom_printf("unix: kernel halted by -h flag\n");
456         prom_enter_mon();
457     }
458
459     ASSERT_STACK_ALIGNED();
460
461     /*
462      * Fill out cpu_ucode_info. Update microcode if necessary.
463      */
464     ucode_check(CPU);
465
466     if (workaround_errata(CPU) != 0)
467         panic("critical workaround(s) missing for boot cpu");
468 }

```

unchanged portion omitted

new/usr/src/uts/i86pc/os/trap.c

1

```
*****
61922 Thu Feb 18 08:40:44 2016
new/usr/src/uts/i86pc/os/trap.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

453 #endif /* OPTERON_ERRATUM_91 */

455 /*
456 * Called from the trap handler when a processor trap occurs.
457 *
458 * Note: All user-level traps that might call stop() must exit
459 * trap() by 'goto out' or by falling through.
460 * Note Also: trap() is usually called with interrupts enabled, (PS_IE == 1)
461 * however, there are paths that arrive here with PS_IE == 0 so special care
462 * must be taken in those cases.
463 */
464 void
465 trap(struct regs *rp, caddr_t addr, processorid_t cpuid)
466 {
467     kthread_t *ct = curthread;
468     enum seg_rw rw;
469     unsigned type;
470     proc_t *p = ttoproc(ct);
471     klwp_t *lwp = ttolwp(ct);
472     uintptr_t lofault;
473     label_t *onfault;
474     faultcode_t pagefault(), res, errcode;
475     enum fault_type fault_type;
476     k_siginfo_t siginfo;
477     uint_t fault = 0;
478     int mstate;
479     int sicode = 0;
480     int watchcode;
481     int watchpage;
482     caddr_t vaddr;
483     int singlestep_twiddle;
484     size_t sz;
485     int ta;
486 #ifdef __amd64
487     uchar_t instr;
488 #endif

490     ASSERT_STACK_ALIGNED();

492     type = rp->r_trapno;
493     CPU_STATS_ADDQ(CPU, sys, trap, 1);
494     ASSERT(ct->t_schedflag & TS_DONT_SWAP);

495     if (type == T_PGFLT) {

497         errcode = rp->r_err;
498         if (errcode & PF_ERR_WRITE)
499             rw = S_WRITE;
500         else if ((caddr_t)rp->r_pc == addr ||
501                (mmu.pt_nx != 0 && (errcode & PF_ERR_EXEC)))
502             rw = S_EXEC;
503         else
504             rw = S_READ;

506 #if defined(__i386)
507     /*
508     * Pentium Pro work-around
509     */
510     if ((errcode & PF_ERR_PROT) && pentiumpro_bug4046376) {
```

new/usr/src/uts/i86pc/os/trap.c

2

```
511         uint_t attr;
512         uint_t priv_violation;
513         uint_t access_violation;

515         if (hat_getattr(addr < (caddr_t)kernelbase ?
516             curproc->p_as->a_hat : kas.a_hat, addr, &attr)
517             == -1) {
518             errcode &= ~PF_ERR_PROT;
519         } else {
520             priv_violation = (errcode & PF_ERR_USER) &&
521                 !(attr & PROT_USER);
522             access_violation = (errcode & PF_ERR_WRITE) &&
523                 !(attr & PROT_WRITE);
524             if (!priv_violation && !access_violation)
525                 goto cleanup;
526         }
527     }
528 #endif /* __i386 */

530     } else if (type == T_SGLSTP && lwp != NULL)
531         lwp->lwp_pcb.pcb_drstat = (uintptr_t)addr;

533     if (tdebug)
534         showregs(type, rp, addr);

536     if (USERMODE(rp->r_cs)) {
537         /*
538         * Set up the current cred to use during this trap. u_cred
539         * no longer exists. t_cred is used instead.
540         * The current process credential applies to the thread for
541         * the entire trap. If trapping from the kernel, this
542         * should already be set up.
543         */
544         if (ct->t_cred != p->p_cred) {
545             cred_t *oldcred = ct->t_cred;
546             /*
547             * DTrace accesses t_cred in probe context. t_cred
548             * must always be either NULL, or point to a valid,
549             * allocated cred structure.
550             */
551             ct->t_cred = crgetcred();
552             crfree(oldcred);
553         }
554         ASSERT(lwp != NULL);
555         type |= USER;
556         ASSERT(lwptoregs(lwp) == rp);
557         lwp->lwp_state = LWP_SYS;

559         switch (type) {
560         case T_PGFLT + USER:
561             if ((caddr_t)rp->r_pc == addr)
562                 mstate = LMS_TFAULT;
563             else
564                 mstate = LMS_DFAULT;
565             break;
566         default:
567             mstate = LMS_TRAP;
568             break;
569         }
570         /* Kernel probe */
571         TNF_PROBE_1(thread_state, "thread", /* CSTYLED */,
572             tnf_microstate, state, mstate);
573         mstate = new_mstate(ct, mstate);

575         bzero(&siginfo, sizeof (siginfo));
576     }
```

```

578     switch (type) {
579     case T_PGFLT + USER:
580     case T_SGLSTP:
581     case T_SGLSTP + USER:
582     case T_BPTFLT + USER:
583         break;

585     default:
586         FTRACE_2("trap(): type=0x%lx, regs=0x%lx",
587             (ulong_t)type, (ulong_t)rp);
588         break;
589     }

591     switch (type) {
592     case T_SIMDFPE:
593         /* Make sure we enable interrupts before die()ing */
594         sti(); /* The SIMD exception comes in via cmninttrap */
595         /*FALLTHROUGH*/
596     default:
597         if (type & USER) {
598             if (tudebug)
599                 showregs(type, rp, (caddr_t)0);
600             printf("trap: Unknown trap type %d in user mode\n",
601                 type & ~USER);
602             siginfo.si_signo = SIGILL;
603             siginfo.si_code = ILL_ILLTRP;
604             siginfo.si_addr = (caddr_t)rp->r_pc;
605             siginfo.si_trapno = type & ~USER;
606             fault = FLTILL;
607             break;
608         } else {
609             (void) die(type, rp, addr, cpuid);
610             /*NOTREACHED*/
611         }

613     case T_PGFLT: /* system page fault */
614         /*
615          * If we're under on_trap() protection (see <sys/ontrap.h>),
616          * set ot_trap and bounce back to the on_trap() call site
617          * via the installed trampoline.
618          */
619         if ((ct->t_ontrap != NULL) &&
620             (ct->t_ontrap->ot_prot & OT_DATA_ACCESS)) {
621             ct->t_ontrap->ot_trap |= OT_DATA_ACCESS;
622             rp->r_pc = ct->t_ontrap->ot_trampoline;
623             goto cleanup;
624         }

626         /*
627          * If we have an Instruction fault in kernel mode, then that
628          * means we've tried to execute a user page (SMEP) or both of
629          * PAE and NXE are enabled. In either case, given that it's a
630          * kernel fault, we should panic immediately and not try to make
631          * any more forward progress. This indicates a bug in the
632          * kernel, which if execution continued, could be exploited to
633          * wreak havoc on the system.
634          */
635         if (errcode & PF_ERR_EXEC) {
636             (void) die(type, rp, addr, cpuid);
637         }

639         /*
640          * See if we can handle as pagefault. Save lofault and onfault
641          * across this. Here we assume that an address less than
642          * KERNELBASE is a user fault. We can do this as copy.s

```

```

643         * routines verify that the starting address is less than
644         * KERNELBASE before starting and because we know that we
645         * always have KERNELBASE mapped as invalid to serve as a
646         * "barrier".
647         */
648         lofault = ct->t_lofault;
649         onfault = ct->t_onfault;
650         ct->t_lofault = 0;

652         mstate = new_mstate(ct, LMS_KFAULT);

654         if (addr < (caddr_t)kernelbase) {
655             res = pagefault(addr,
656                 (errcode & PF_ERR_PROT)? F_PROT: F_INVALID, rw, 0);
657             if (res == FC_NOMAP &&
658                 addr < p->p_usrstack &&
659                 grow(addr))
660                 res = 0;
661         } else {
662             res = pagefault(addr,
663                 (errcode & PF_ERR_PROT)? F_PROT: F_INVALID, rw, 1);
664         }
665         (void) new_mstate(ct, mstate);

667         /*
668          * Restore lofault and onfault. If we resolved the fault, exit.
669          * If we didn't and lofault wasn't set, die.
670          */
671         ct->t_lofault = lofault;
672         ct->t_onfault = onfault;
673         if (res == 0)
674             goto cleanup;

676 #if defined(OPTERON_ERRATUM_93) && defined(LP64)
677     if (lofault == 0 && opteron_erratum_93) {
678         /*
679          * Workaround for Opteron Erratum 93. On return from
680          * a System Management Interrupt at a HLT instruction
681          * the %rip might be truncated to a 32 bit value.
682          * BIOS is supposed to fix this, but some don't.
683          * If this occurs we simply restore the high order bits.
684          * The HLT instruction is 1 byte of 0xf4.
685          */
686         uintptr_t rip = rp->r_pc;

688         if ((rip & 0xfffffffful) == rip) {
689             rip |= 0xfffffffful << 32;
690             if (hat_getpfnum(kas.a_hat, (caddr_t)rip) !=
691                 PFN_INVALID &&
692                 (*(uchar_t *)rip == 0xf4 ||
693                 *(uchar_t *)rip - 1 == 0xf4)) {
694                 rp->r_pc = rip;
695                 goto cleanup;
696             }
697         }
698     }
699 #endif /* OPTERON_ERRATUM_93 && LP64 */

701 #ifdef OPTERON_ERRATUM_91
702     if (lofault == 0 && opteron_erratum_91) {
703         /*
704          * Workaround for Opteron Erratum 91. Prefetches may
705          * generate a page fault (they're not supposed to do
706          * that!). If this occurs we simply return back to the
707          * instruction.
708          */

```

```

709         caddr_t         pc = (caddr_t)rp->r_pc;
710
711         /*
712          * If the faulting PC is not mapped, this is a
713          * legitimate kernel page fault that must result in a
714          * panic. If the faulting PC is mapped, it could contain
715          * a prefetch instruction. Check for that here.
716          */
717         if (hat_getpfnum(kas.a_hat, pc) != PFN_INVALID) {
718             if (cmp_to_prefetch((uchar_t *)pc)) {
719 #ifdef DEBUG
720                 cmn_err(CE_WARN, "Opteron erratum 91 "
721                     "occurred: kernel prefetch"
722                     " at %p generated a page fault!",
723                     (void *)rp->r_pc);
724 #endif /* DEBUG */
725                 goto cleanup;
726             }
727         }
728         (void) die(type, rp, addr, cpuid);
729     }
730 #endif /* OPTERON_ERRATUM_91 */
731
732     if (lofault == 0)
733         (void) die(type, rp, addr, cpuid);
734
735     /*
736      * Cannot resolve fault. Return to lofault.
737      */
738     if (lodebug) {
739         showregs(type, rp, addr);
740         traceregs(rp);
741     }
742     if (FC_CODE(res) == FC_OBJERR)
743         res = FC_ERRNO(res);
744     else
745         res = EFAULT;
746     rp->r_r0 = res;
747     rp->r_pc = ct->t_lofault;
748     goto cleanup;
749
750     case T_PGFLT + USER: /* user page fault */
751         if (faultdebug) {
752             char *fault_str;
753
754             switch (rw) {
755                 case S_READ:
756                     fault_str = "read";
757                     break;
758                 case S_WRITE:
759                     fault_str = "write";
760                     break;
761                 case S_EXEC:
762                     fault_str = "exec";
763                     break;
764                 default:
765                     fault_str = "";
766                     break;
767             }
768             printf("user %s fault: addr=0x%lx errcode=0x%x\n",
769                 fault_str, (uintptr_t)addr, errcode);
770         }
771
772 #if defined(OPTERON_ERRATUM_100) && defined(LP64)
773     /*
774      * Workaround for AMD erratum 100

```

```

775     *
776     * A 32-bit process may receive a page fault on a non
777     * 32-bit address by mistake. The range of the faulting
778     * address will be
779     *
780     * 0xffffffff80000000 .. 0xffffffffffffffff or
781     * 0x0000000100000000 .. 0x000000017fffffff
782     *
783     * The fault is always due to an instruction fetch, however
784     * the value of r_pc should be correct (in 32 bit range),
785     * so we ignore the page fault on the bogus address.
786     */
787     if (p->p_model == DATAMODEL_ILP32 &&
788         (0xffffffff80000000 <= (uintptr_t)addr ||
789          (0x100000000 <= (uintptr_t)addr &&
790           (uintptr_t)addr <= 0x17fffffff))) {
791         if (!opteron_erratum_100)
792             panic("unexpected erratum #100");
793         if (rp->r_pc <= 0xffffffff)
794             goto out;
795     }
796 #endif /* OPTERON_ERRATUM_100 && LP64 */
797
798     ASSERT(!(curthread->t_flag & T_WATCHPT));
799     watchpage = (pr_watch_active(p) && pr_is_watchpage(addr, rw));
800 #ifdef __i386
801     /*
802      * In 32-bit mode, the lcall (system call) instruction fetches
803      * one word from the stack, at the stack pointer, because of the
804      * way the call gate is constructed. This is a bogus
805      * read and should not be counted as a read watchpoint.
806      * We work around the problem here by testing to see if
807      * this situation applies and, if so, simply jumping to
808      * the code in locore.s that fields the system call trap.
809      * The registers on the stack are already set up properly
810      * due to the match between the call gate sequence and the
811      * trap gate sequence. We just have to adjust the pc.
812      */
813     if (watchpage && addr == (caddr_t)rp->r_sp &&
814         rw == S_READ && instr_is_lcall_syscall((caddr_t)rp->r_pc)) {
815         extern void watch_syscall(void);
816
817         rp->r_pc += LCALLSIZE;
818         watch_syscall(); /* never returns */
819         /* NOTREACHED */
820     }
821 #endif /* __i386 */
822     vaddr = addr;
823     if (!watchpage || (sz = instr_size(rp, &vaddr, rw)) <= 0)
824         fault_type = (errcode & PF_ERR_PROT)? F_PROT: F_INVALID;
825     else if ((watchcode = pr_is_watchpoint(&vaddr, &ta,
826         sz, NULL, rw)) != 0) {
827         if (ta) {
828             do_watch_step(vaddr, sz, rw,
829                 watchcode, rp->r_pc);
830             fault_type = F_INVALID;
831         } else {
832             bzero(&siginfo, sizeof (siginfo));
833             siginfo.si_signo = SIGTRAP;
834             siginfo.si_code = watchcode;
835             siginfo.si_addr = vaddr;
836             siginfo.si_trapafter = 0;
837             siginfo.si_pc = (caddr_t)rp->r_pc;
838             fault = FLTWATCH;
839             break;
840         }

```

```

841     } else {
842         /* XXX pr_watch_emul() never succeeds (for now) */
843         if (rw != S_EXEC && pr_watch_emul(rp, vaddr, rw))
844             goto out;
845         do_watch_step(vaddr, sz, rw, 0, 0);
846         fault_type = F_INVALID;
847     }
848
849     res = pagefault(addr, fault_type, rw, 0);
850
851     /*
852     * If pagefault() succeeded, ok.
853     * Otherwise attempt to grow the stack.
854     */
855     if (res == 0 ||
856         (res == FC_NOMAP &&
857          addr < p->p_usrstack &&
858          grow(addr))) {
859         lwp->lwp_lastfault = FLTPAGE;
860         lwp->lwp_lastfaddr = addr;
861         if (prismember(&p->p_fltmask, FLTPAGE)) {
862             bzero(&siginfo, sizeof(siginfo));
863             siginfo.si_addr = addr;
864             (void) stop_on_fault(FLTPAGE, &siginfo);
865         }
866         goto out;
867     } else if (res == FC_PROT && addr < p->p_usrstack &&
868               (mmu.pt_nx != 0 && (errcode & PF_ERR_EXEC))) {
869         report_stack_exec(p, addr);
870     }
871
872 #ifndef OPTERON_ERRATUM_91
873     /*
874     * Workaround for Opteron Erratum 91. Prefetches may generate a
875     * page fault (they're not supposed to do that!). If this
876     * occurs we simply return back to the instruction.
877     *
878     * We rely on copyin to properly fault in the page with r_pc.
879     */
880     if (opteron_erratum_91 &&
881         addr != (caddr_t)rp->r_pc &&
882         instr_is_prefetch((caddr_t)rp->r_pc)) {
883 #ifdef DEBUG
884         cmn_err(CE_WARN, "Opteron erratum 91 occurred: "
885              "prefetch at %p in pid %d generated a trap!",
886              (void *)rp->r_pc, p->p_pid);
887 #endif /* DEBUG */
888         goto out;
889     }
890 #endif /* OPTERON_ERRATUM_91 */
891
892     if (tudebug)
893         showregs(type, rp, addr);
894     /*
895     * In the case where both pagefault and grow fail,
896     * set the code to the value provided by pagefault.
897     * We map all errors returned from pagefault() to SIGSEGV.
898     */
899     bzero(&siginfo, sizeof(siginfo));
900     siginfo.si_addr = addr;
901     switch (FC_CODE(res)) {
902     case FC_HWERR:
903     case FC_NOSUPPORT:
904         siginfo.si_signo = SIGBUS;
905         siginfo.si_code = BUS_ADRERR;
906         fault = FLTACCESS;

```

```

907         break;
908     case FC_ALIGN:
909         siginfo.si_signo = SIGBUS;
910         siginfo.si_code = BUS_ADRALN;
911         fault = FLTACCESS;
912         break;
913     case FC_OBJERR:
914         if ((siginfo.si_errno = FC_ERRNO(res)) != EINTR) {
915             siginfo.si_signo = SIGBUS;
916             siginfo.si_code = BUS_OBJERR;
917             fault = FLTACCESS;
918         }
919         break;
920     default: /* FC_NOMAP or FC_PROT */
921         siginfo.si_signo = SIGSEGV;
922         siginfo.si_code =
923             (res == FC_NOMAP)? SEGV_MAPERR : SEGV_ACCERR;
924         fault = FLTBOUNDS;
925         break;
926     }
927     break;
928
929     case T_ILLINST + USER: /* invalid opcode fault */
930     /*
931     * If the syscall instruction is disabled due to LDT usage, a
932     * user program that attempts to execute it will trigger a #ud
933     * trap. Check for that case here. If this occurs on a CPU which
934     * doesn't even support syscall, the result of all of this will
935     * be to emulate that particular instruction.
936     */
937     if (p->p_ldt != NULL &&
938         ldt_rewrite_syscall(rp, p, X86FSET_ASYS))
939         goto out;
940
941 #ifdef __amd64
942     /*
943     * Emulate the LAHF and SAHF instructions if needed.
944     * See the instr_is_lsahf function for details.
945     */
946     if (p->p_model == DATAMODEL_LP64 &&
947         instr_is_lsahf((caddr_t)rp->r_pc, &instr)) {
948         emulate_lsahf(rp, instr);
949         goto out;
950     }
951 #endif
952
953     /*FALLTHROUGH*/
954
955     if (tudebug)
956         showregs(type, rp, (caddr_t)0);
957     siginfo.si_signo = SIGILL;
958     siginfo.si_code = ILL_ILLOPC;
959     siginfo.si_addr = (caddr_t)rp->r_pc;
960     fault = FLTILL;
961     break;
962
963     case T_ZERODIV + USER: /* integer divide by zero */
964     if (tudebug && tudebugfpe)
965         showregs(type, rp, (caddr_t)0);
966     siginfo.si_signo = SIGFPE;
967     siginfo.si_code = FPE_INTDIV;
968     siginfo.si_addr = (caddr_t)rp->r_pc;
969     fault = FLTIZDIV;
970     break;
971
972     case T_OVFLW + USER: /* integer overflow */

```

```

973     if (tudebug && tudebugfpe)
974         showregs(type, rp, (caddr_t)0);
975     siginfo.si_signo = SIGFPE;
976     siginfo.si_code = FPE_INTOVF;
977     siginfo.si_addr = (caddr_t)rp->r_pc;
978     fault = FLTIOVF;
979     break;

981 case T_NOEXTFLT + USER: /* math coprocessor not available */
982     if (tudebug && tudebugfpe)
983         showregs(type, rp, addr);
984     if (fpnoextflt(rp)) {
985         siginfo.si_signo = SIGILL;
986         siginfo.si_code = ILL_ILLOPC;
987         siginfo.si_addr = (caddr_t)rp->r_pc;
988         fault = FLTILL;
989     }
990     break;

992 case T_EXTOVRFLT: /* extension overrun fault */
993     /* check if we took a kernel trap on behalf of user */
994     {
995         extern void ndptrap_frstor(void);
996         if (rp->r_pc != (uintptr_t)ndptrap_frstor) {
997             sti(); /* T_EXTOVRFLT comes in via cmninttrap */
998             (void) die(type, rp, addr, cpuid);
999         }
1000         type |= USER;
1001     }
1002     /*FALLTHROUGH*/
1003 case T_EXTOVRFLT + USER: /* extension overrun fault */
1004     if (tudebug && tudebugfpe)
1005         showregs(type, rp, addr);
1006     if (fpextovrflt(rp)) {
1007         siginfo.si_signo = SIGSEGV;
1008         siginfo.si_code = SEGV_MAPERR;
1009         siginfo.si_addr = (caddr_t)rp->r_pc;
1010         fault = FLTBOUNDS;
1011     }
1012     break;

1014 case T_EXTERRFLT: /* x87 floating point exception pending */
1015     /* check if we took a kernel trap on behalf of user */
1016     {
1017         extern void ndptrap_frstor(void);
1018         if (rp->r_pc != (uintptr_t)ndptrap_frstor) {
1019             sti(); /* T_EXTERRFLT comes in via cmninttrap */
1020             (void) die(type, rp, addr, cpuid);
1021         }
1022         type |= USER;
1023     }
1024     /*FALLTHROUGH*/

1026 case T_EXTERRFLT + USER: /* x87 floating point exception pending */
1027     if (tudebug && tudebugfpe)
1028         showregs(type, rp, addr);
1029     if (sicode = fpexterrflt(rp)) {
1030         siginfo.si_signo = SIGFPE;
1031         siginfo.si_code = sicode;
1032         siginfo.si_addr = (caddr_t)rp->r_pc;
1033         fault = FLTFPE;
1034     }
1035     break;

1037 case T_SIMDFPE + USER: /* SSE and SSE2 exceptions */
1038     if (tudebug && tudebugsse)

```

```

1039         showregs(type, rp, addr);
1040     if (!is_x86_feature(x86_featureset, X86FSET_SSE) &&
1041         !is_x86_feature(x86_featureset, X86FSET_SSE2)) {
1042         /*
1043          * There are rumours that some user instructions
1044          * on older CPUs can cause this trap to occur; in
1045          * which case send a SIGILL instead of a SIGFPE.
1046          */
1047         siginfo.si_signo = SIGILL;
1048         siginfo.si_code = ILL_ILLTRP;
1049         siginfo.si_addr = (caddr_t)rp->r_pc;
1050         siginfo.si_trapno = type & ~USER;
1051         fault = FLTILL;
1052     } else if ((sicode = fpsimderrflt(rp)) != 0) {
1053         siginfo.si_signo = SIGFPE;
1054         siginfo.si_code = sicode;
1055         siginfo.si_addr = (caddr_t)rp->r_pc;
1056         fault = FLTFPE;
1057     }

1059     sti(); /* The SIMD exception comes in via cmninttrap */
1060     break;

1062 case T_BPTFLT: /* breakpoint trap */
1063     /*
1064      * Kernel breakpoint traps should only happen when kmdb is
1065      * active, and even then, it'll have interposed on the IDT, so
1066      * control won't get here. If it does, we've hit a breakpoint
1067      * without the debugger, which is very strange, and very
1068      * fatal.
1069      */
1070     if (tudebug && tudebugbpt)
1071         showregs(type, rp, (caddr_t)0);

1073     (void) die(type, rp, addr, cpuid);
1074     break;

1076 case T_SGLSTP: /* single step/hw breakpoint exception */

1078     /* Now evaluate how we got here */
1079     if (lwp != NULL && (lwp->lwp_pcb.pcb_drstat & DR_SINGLESTEP)) {
1080         /*
1081          * i386 single-steps even through lcalls which
1082          * change the privilege level. So we take a trap at
1083          * the first instruction in privileged mode.
1084          *
1085          * Set a flag to indicate that upon completion of
1086          * the system call, deal with the single-step trap.
1087          *
1088          * The same thing happens for sysenter, too.
1089          */
1090         singlestep_twiddle = 0;
1091         if (rp->r_pc == (uintptr_t)sys_sysenter ||
1092             rp->r_pc == (uintptr_t)brand_sys_sysenter) {
1093             singlestep_twiddle = 1;
1094 #if defined(__amd64)
1095             /*
1096              * Since we are already on the kernel's
1097              * %gs, on 64-bit systems the sysenter case
1098              * needs to adjust the pc to avoid
1099              * executing the swags instruction at the
1100              * top of the handler.
1101              */
1102             if (rp->r_pc == (uintptr_t)sys_sysenter)
1103                 rp->r_pc = (uintptr_t)
1104                     _sys_sysenter_post_swags;

```

```

1105         else
1106             rp->r_pc = (uintptr_t)
1107                 _brand_sys_sysenter_post_swaps;
1108 #endif
1109     }
1110 #if defined(__i386)
1111     else if (rp->r_pc == (uintptr_t)sys_call ||
1112             rp->r_pc == (uintptr_t)brand_sys_call) {
1113         singlestep_twiddle = 1;
1114     }
1115 #endif
1116     else {
1117         /* not on sysenter/syscall; uregs available */
1118         if (tudebug && tudebugbpt)
1119             showregs(type, rp, (caddr_t)0);
1120     }
1121     if (singlestep_twiddle) {
1122         rp->r_ps &= ~PS_T; /* turn off trace */
1123         lwp->lwp_pcb.pcb_flags |= DEBUG_PENDING;
1124         ct->t_post_sys = 1;
1125         aston(curthread);
1126         goto cleanup;
1127     }
1128 }
1129 /* XXX - needs review on debugger interface? */
1130 if (boothowto & RB_DEBUG)
1131     debug_enter((char *)NULL);
1132 else
1133     (void) die(type, rp, addr, cpuid);
1134 break;

1136 case T_NMIFLT: /* NMI interrupt */
1137     printf("Unexpected NMI in system mode\n");
1138     goto cleanup;

1140 case T_NMIFLT + USER: /* NMI interrupt */
1141     printf("Unexpected NMI in user mode\n");
1142     break;

1144 case T_GPFFLT: /* general protection violation */
1145     /*
1146      * Any #GP that occurs during an on_trap .. no_trap bracket
1147      * with OT_DATA_ACCESS or OT_SEGMENT_ACCESS protection,
1148      * or in a on_fault .. no_fault bracket, is forgiven
1149      * and we trampoline. This protection is given regardless
1150      * of whether we are 32/64 bit etc - if a distinction is
1151      * required then define new on_trap protection types.
1152      *
1153      * On amd64, we can get a #gp from referencing addresses
1154      * in the virtual address hole e.g. from a copyin or in
1155      * update_sregs while updating user segment registers.
1156      *
1157      * On the 32-bit hypervisor we could also generate one in
1158      * mfn_to_pfn by reaching around or into where the hypervisor
1159      * lives which is protected by segmentation.
1160      */

1162     /*
1163      * If we're under on_trap() protection (see <sys/ontrap.h>),
1164      * set ot_trap and trampoline back to the on_trap() call site
1165      * for OT_DATA_ACCESS or OT_SEGMENT_ACCESS.
1166      */
1167     if (ct->t_ontrap != NULL) {
1168         int ttype = ct->t_ontrap->ot_prot &
1169                 (OT_DATA_ACCESS | OT_SEGMENT_ACCESS);

```

```

1171         if (ttype != 0) {
1172             ct->t_ontrap->ot_trap |= ttype;
1173             if (tudebug)
1174                 showregs(type, rp, (caddr_t)0);
1175             rp->r_pc = ct->t_ontrap->ot_trampoline;
1176             goto cleanup;
1177         }
1178     }

1180     /*
1181      * If we're under lofault protection (copyin etc.),
1182      * longjmp back to lofault with an EFAULT.
1183      */
1184     if (ct->t_lofault) {
1185         /*
1186          * Fault is not resolvable, so just return to lofault
1187          */
1188         if (lodebug) {
1189             showregs(type, rp, addr);
1190             traceregs(rp);
1191         }
1192         rp->r_r0 = EFAULT;
1193         rp->r_pc = ct->t_lofault;
1194         goto cleanup;
1195     }

1197     /*
1198      * We fall through to the next case, which repeats
1199      * the OT_SEGMENT_ACCESS check which we've already
1200      * done, so we'll always fall through to the
1201      * T_STKFLT case.
1202      */
1203     /*FALLTHROUGH*/
1204 case T_SEGFLT: /* segment not present fault */
1205     /*
1206      * One example of this is #NP in update_sregs while
1207      * attempting to update a user segment register
1208      * that points to a descriptor that is marked not
1209      * present.
1210      */
1211     if (ct->t_ontrap != NULL &&
1212         ct->t_ontrap->ot_prot & OT_SEGMENT_ACCESS) {
1213         ct->t_ontrap->ot_trap |= OT_SEGMENT_ACCESS;
1214         if (tudebug)
1215             showregs(type, rp, (caddr_t)0);
1216         rp->r_pc = ct->t_ontrap->ot_trampoline;
1217         goto cleanup;
1218     }
1219     /*FALLTHROUGH*/
1220 case T_STKFLT: /* stack fault */
1221 case T_TSSFLT: /* invalid TSS fault */
1222     if (tudebug)
1223         showregs(type, rp, (caddr_t)0);
1224     if (kern_gpfault(rp))
1225         (void) die(type, rp, addr, cpuid);
1226     goto cleanup;

1228     /*
1229      * ONLY 32-bit PROCESSES can USE a PRIVATE LDT! 64-bit apps
1230      * should have no need for them, so we put a stop to it here.
1231      *
1232      * So: not-present fault is ONLY valid for 32-bit processes with
1233      * a private LDT trying to do a system call. Emulate it.
1234      *
1235      * #gp fault is ONLY valid for 32-bit processes also, which DO NOT
1236      * have a private LDT, and are trying to do a system call. Emulate it.

```

```

1237      */
1239      case T_SEGFLT + USER: /* segment not present fault */
1240      case T_GPFLT + USER: /* general protection violation */
1241 #ifdef _SYSCALL32_IMPL
1242      if (p->p_model != DATAMODEL_NATIVE) {
1243 #endif /* _SYSCALL32_IMPL */
1244      if (instr_is_lcall_syscall((caddr_t)rp->r_pc) {
1245          if (type == T_SEGFLT + USER)
1246              ASSERT(p->p_ldt != NULL);
1248      if ((p->p_ldt == NULL && type == T_GPFLT + USER) ||
1249          type == T_SEGFLT + USER) {
1251          /*
1252          * The user attempted a system call via the obsolete
1253          * call gate mechanism. Because the process doesn't have
1254          * an LDT (i.e. the ldtr contains 0), a #gp results.
1255          * Emulate the syscall here, just as we do above for a
1256          * #np trap.
1257          */
1259          /*
1260          * Since this is a not-present trap, rp->r_pc points to
1261          * the trapping lcall instruction. We need to bump it
1262          * to the next insn so the app can continue on.
1263          */
1264          rp->r_pc += LCALLSIZE;
1265          lwp->lwp_regs = rp;
1267          /*
1268          * Normally the microstate of the LWP is forced back to
1269          * LMS_USER by the syscall handlers. Emulate that
1270          * behavior here.
1271          */
1272          mstate = LMS_USER;
1274          dosyscall();
1275          goto out;
1276      }
1277  }
1278 #ifdef _SYSCALL32_IMPL
1279  }
1280 #endif /* _SYSCALL32_IMPL */
1281  /*
1282  * If the current process is using a private LDT and the
1283  * trapping instruction is sysenter, the sysenter instruction
1284  * has been disabled on the CPU because it destroys segment
1285  * registers. If this is the case, rewrite the instruction to
1286  * be a safe system call and retry it. If this occurs on a CPU
1287  * which doesn't even support sysenter, the result of all of
1288  * this will be to emulate that particular instruction.
1289  */
1290  if (p->p_ldt != NULL &&
1291      ldt_rewrite_syscall(rp, p, X86FSET_SEP))
1292      goto out;
1294      /*FALLTHROUGH*/
1296      case T_BOUNDFLT + USER: /* bound fault */
1297      case T_STKFLT + USER: /* stack fault */
1298      case T_TSSFLT + USER: /* invalid TSS fault */
1299          if (tudebug)
1300              showregs(type, rp, (caddr_t)0);
1301          siginfo.si_signo = SIGSEGV;
1302          siginfo.si_code = SEGV_MAPERR;

```

```

1303          siginfo.si_addr = (caddr_t)rp->r_pc;
1304          fault = FLTBOUNDS;
1305          break;
1307      case T_ALIGNMENT + USER: /* user alignment error (486) */
1308          if (tudebug)
1309              showregs(type, rp, (caddr_t)0);
1310          bzero(&siginfo, sizeof (siginfo));
1311          siginfo.si_signo = SIGBUS;
1312          siginfo.si_code = BUS_ADRALN;
1313          siginfo.si_addr = (caddr_t)rp->r_pc;
1314          fault = FLTACCESS;
1315          break;
1317      case T_SGLSTP + USER: /* single step/hw breakpoint exception */
1318          if (tudebug && tudebugbpt)
1319              showregs(type, rp, (caddr_t)0);
1321          /* Was it single-stepping? */
1322          if (lwp->lwp_pcb.pcb_drstat & DR_SINGLESTEP) {
1323              pcb_t *pcb = &lwp->lwp_pcb;
1325              rp->r_ps &= ~PS_T;
1326              /*
1327              * If both NORMAL_STEP and WATCH_STEP are in effect,
1328              * give precedence to WATCH_STEP. If neither is set,
1329              * user must have set the PS_T bit in %efl; treat this
1330              * as NORMAL_STEP.
1331              */
1332              if (((fault = undo_watch_step(&siginfo)) == 0 &&
1333                  ((pcb->pcb_flags & NORMAL_STEP) ||
1334                   !(pcb->pcb_flags & WATCH_STEP))) {
1335                  siginfo.si_signo = SIGTRAP;
1336                  siginfo.si_code = TRAP_TRACE;
1337                  siginfo.si_addr = (caddr_t)rp->r_pc;
1338                  fault = FLITRACE;
1339              }
1340              pcb->pcb_flags &= ~(NORMAL_STEP|WATCH_STEP);
1341          }
1342          break;
1344      case T_BPTFLT + USER: /* breakpoint trap */
1345          if (tudebug && tudebugbpt)
1346              showregs(type, rp, (caddr_t)0);
1347          /*
1348          * int 3 (the breakpoint instruction) leaves the pc referring
1349          * to the address one byte after the breakpointed address.
1350          * If the P_PR_BPTADJ flag has been set via /proc, We adjust
1351          * it back so it refers to the breakpointed address.
1352          */
1353          if (p->p_proc_flag & P_PR_BPTADJ)
1354              rp->r_pc--;
1355          siginfo.si_signo = SIGTRAP;
1356          siginfo.si_code = TRAP_BRKPT;
1357          siginfo.si_addr = (caddr_t)rp->r_pc;
1358          fault = FLTBP;
1359          break;
1361      case T_AST:
1362          /*
1363          * This occurs only after the cs register has been made to
1364          * look like a kernel selector, either through debugging or
1365          * possibly by functions like setcontext(). The thread is
1366          * about to cause a general protection fault at common_iret()
1367          * in locore. We let that happen immediately instead of
1368          * doing the T_AST processing.

```



```

1369     */
1370     goto cleanup;

1372     case T_AST + USER: /* profiling, resched, h/w error pseudo trap */
1373         if (lwp->lwp_pcb.pcb_flags & ASYNC_HWERR) {
1374             proc_t *p = ttoproc(curthread);
1375             extern void print_msg_hwerr(ctid_t ct_id, proc_t *p);

1377             lwp->lwp_pcb.pcb_flags &= ~ASYNC_HWERR;
1378             print_msg_hwerr(p->p_ct_process->conp_contract.ct_id,
1379                 p);
1380             contract_process_hwerr(p->p_ct_process, p);
1381             siginfo.si_signo = SIGKILL;
1382             siginfo.si_code = SI_NOINFORM;
1383         } else if (lwp->lwp_pcb.pcb_flags & CPC_OVERFLOW) {
1384             lwp->lwp_pcb.pcb_flags &= ~CPC_OVERFLOW;
1385             if (kcpc_overflow_ast()) {
1386                 /*
1387                  * Signal performance counter overflow
1388                  */
1389                 if (tudebug)
1390                     showregs(type, rp, (caddr_t)0);
1391                 bzero(&siginfo, sizeof (siginfo));
1392                 siginfo.si_signo = SIGEMT;
1393                 siginfo.si_code = EMT_CPCOVF;
1394                 siginfo.si_addr = (caddr_t)rp->r_pc;
1395                 fault = FLT_CPCOVF;
1396             }
1397         }

1399     }
1400 }

1402 /*
1403  * We can't get here from a system trap
1404  */
1405 ASSERT(type & USER);

1407 if (fault) {
1408     /* We took a fault so abort single step. */
1409     lwp->lwp_pcb.pcb_flags &= ~(NORMAL_STEP|WATCH_STEP);
1410     /*
1411     * Remember the fault and fault address
1412     * for real-time (SIGPROF) profiling.
1413     */
1414     lwp->lwp_lastfault = fault;
1415     lwp->lwp_lastfaddr = siginfo.si_addr;

1417     DTRACE_PROG2(fault, int, fault, &siginfo_t *, &siginfo);

1419     /*
1420     * If a debugger has declared this fault to be an
1421     * event of interest, stop the lwp. Otherwise just
1422     * deliver the associated signal.
1423     */
1424     if (siginfo.si_signo != SIGKILL &&
1425         prismember(&p->p_fltmask, fault) &&
1426         stop_on_fault(fault, &siginfo) == 0)
1427         siginfo.si_signo = 0;
1428 }

1430 if (siginfo.si_signo)
1431     trapsig(&siginfo, (fault != FLT_FPE && fault != FLT_CPCOVF));

1433 if (lwp->lwp_oweupc)
1434     profil_tick(rp->r_pc);

```

```

1436     if (ct->t_astflag | ct->t_sig_check) {
1437         /*
1438          * Turn off the AST flag before checking all the conditions that
1439          * may have caused an AST. This flag is on whenever a signal or
1440          * unusual condition should be handled after the next trap or
1441          * syscall.
1442          */
1443         astoff(ct);
1444         /*
1445          * If a single-step trap occurred on a syscall (see above)
1446          * recognize it now. Do this before checking for signals
1447          * because deferred_singlestep_trap() may generate a SIGTRAP to
1448          * the LWP or may otherwise mark the LWP to call issig(FORREAL).
1449          */
1450         if (lwp->lwp_pcb.pcb_flags & DEBUG_PENDING)
1451             deferred_singlestep_trap((caddr_t)rp->r_pc);

1453         ct->t_sig_check = 0;

1455         mutex_enter(&p->p_lock);
1456         if (curthread->t_proc_flag & TP_CHANGEBIND) {
1457             timer_lwpbind();
1458             curthread->t_proc_flag &= ~TP_CHANGEBIND;
1459         }
1460         mutex_exit(&p->p_lock);

1462         /*
1463          * for kaio requests that are on the per-process poll queue,
1464          * aiop->aio_pollq, they're AIO_POLL bit is set, the kernel
1465          * should copyout their result_t to user memory. by copying
1466          * out the result_t, the user can poll on memory waiting
1467          * for the kaio request to complete.
1468          */
1469         if (p->p_aio)
1470             aio_cleanup(0);

1471         /*
1472          * If this LWP was asked to hold, call holdlwp(), which will
1473          * stop. holdlwps() sets this up and calls pokelwps() which
1474          * sets the AST flag.
1475          *
1476          * Also check TP_EXITLWP, since this is used by fresh new LWPs
1477          * through lwp_rtt(). That flag is set if the lwp_create(2)
1478          * syscall failed after creating the LWP.
1479          */
1480         if (ISHOLD(p))
1481             holdlwp();

1483         /*
1484          * All code that sets signals and makes ISSIG evaluate true must
1485          * set t_astflag afterwards.
1486          */
1487         if (ISSIG_PENDING(ct, lwp, p)) {
1488             if (issig(FORREAL))
1489                 psig();
1490             ct->t_sig_check = 1;
1491         }

1493         if (ct->t_rprof != NULL) {
1494             realsigprof(0, 0, 0);
1495             ct->t_sig_check = 1;
1496         }

1498         /*
1499          * /proc can't enable/disable the trace bit itself
1500          * because that could race with the call gate used by

```

```
1501         * system calls via "lcall". If that happened, an
1502         * invalid EFLAGS would result. prstep()/prnostep()
1503         * therefore schedule an AST for the purpose.
1504         */
1505         if (lwp->lwp_pcb.pcb_flags & REQUEST_STEP) {
1506             lwp->lwp_pcb.pcb_flags &= ~REQUEST_STEP;
1507             rp->r_ps |= PS_T;
1508         }
1509         if (lwp->lwp_pcb.pcb_flags & REQUEST_NOSTEP) {
1510             lwp->lwp_pcb.pcb_flags &= ~REQUEST_NOSTEP;
1511             rp->r_ps &= ~PS_T;
1512         }
1513     }
1514
1515 out:    /* We can't get here from a system trap */
1516     ASSERT(type & USER);
1517
1518     if (ISHOLD(p))
1519         holdlwp();
1520
1521     /*
1522     * Set state to LWP_USER here so preempt won't give us a kernel
1523     * priority if it occurs after this point. Call CL_TRAPRET() to
1524     * restore the user-level priority.
1525     *
1526     * It is important that no locks (other than spinlocks) be entered
1527     * after this point before returning to user mode (unless lwp_state
1528     * is set back to LWP_SYS).
1529     */
1530     lwp->lwp_state = LWP_USER;
1531
1532     if (ct->t_trapret) {
1533         ct->t_trapret = 0;
1534         thread_lock(ct);
1535         CL_TRAPRET(ct);
1536         thread_unlock(ct);
1537     }
1538     if (CPU->cpu_runrun || curthread->t_schedflag & TS_ANYWAITQ)
1539         preempt();
1540     prunstop();
1541     (void) new_mstate(ct, mstate);
1542
1543     /* Kernel probe */
1544     TNF_PROBE_1(thread_state, "thread", /* CSTYLEL */);
1545     tnf_microstate, state, LMS_USER);
1546
1547     return;
1548
1549 cleanup: /* system traps end up here */
1550     ASSERT(!(type & USER));
1551 }
```

unchanged\_portion\_omitted

```

*****
105436 Thu Feb 18 08:40:44 2016
new/usr/src/uts/i86pc/vm/hat_i86.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

1127 /*
1128  * Allocate any hat resources required for a process being swapped in.
1129  */
1130 /*ARGSUSED*/
1131 void
1132 hat_swapin(hat_t *hat)
1133 {
1134     /* do nothing - we let everything fault back in */
1135 }

1137 /*
1138  * Unload all translations associated with an address space of a process
1139  * that is being swapped out.
1140  */
1141 void
1142 hat_swapout(hat_t *hat)
1143 {
1144     uintptr_t    vaddr = (uintptr_t)0;
1145     uintptr_t    eaddr = _userlimit;
1146     htable_t     *ht = NULL;
1147     level_t      l;

1149     XPV_DISALLOW_MIGRATE();
1150     /*
1151      * We can't just call hat_unload(hat, 0, _userlimit...) here, because
1152      * seg_spt and shared pagetables can't be swapped out.
1153      * Take a look at segspt_shmswapout() - it's a big no-op.
1154      *
1155      * Instead we'll walk through all the address space and unload
1156      * any mappings which we are sure are not shared, not locked.
1157      */
1158     ASSERT(IS_PAGEALIGNED(vaddr));
1159     ASSERT(IS_PAGEALIGNED(eaddr));
1160     ASSERT(AS_LOCK_HELD(hat->hat_as));
1161     if ((uintptr_t)hat->hat_as->a_userlimit < eaddr)
1162         eaddr = (uintptr_t)hat->hat_as->a_userlimit;

1164     while (vaddr < eaddr) {
1165         (void) htable_walk(hat, &ht, &vaddr, eaddr);
1166         if (ht == NULL)
1167             break;

1169         ASSERT(!IN_VA_HOLE(vaddr));

1171         /*
1172          * If the page table is shared skip its entire range.
1173          */
1174         l = ht->ht_level;
1175         if (ht->ht_flags & HTABLE_SHARED_PFN) {
1176             vaddr = ht->ht_vaddr + LEVEL_SIZE(l + 1);
1177             htable_release(ht);
1178             ht = NULL;
1179             continue;
1180         }

1182         /*
1183          * If the page table has no locked entries, unload this one.
1184          */
1185         if (ht->ht_lock_cnt == 0)

```

```

1186         hat_unload(hat, (caddr_t)vaddr, LEVEL_SIZE(l),
1187                     HAT_UNLOAD_UNMAP);

1189         /*
1190          * If we have a level 0 page table with locked entries,
1191          * skip the entire page table, otherwise skip just one entry.
1192          */
1193         if (ht->ht_lock_cnt > 0 && l == 0)
1194             vaddr = ht->ht_vaddr + LEVEL_SIZE(1);
1195         else
1196             vaddr += LEVEL_SIZE(1);
1197     }
1198     if (ht)
1199         htable_release(ht);

1201     /*
1202      * We're in swapout because the system is low on memory, so
1203      * go back and flush all the htables off the cached list.
1204      */
1205     htable_purge_hat(hat);
1206     XPV_ALLOW_MIGRATE();
1207 }

1209 /*
1210  * returns number of bytes that have valid mappings in hat.
1211  */
1212 size_t
1213 hat_get_mapped_size(hat_t *hat)
1214 {
1215     size_t total = 0;
1216     int l;

1218     for (l = 0; l <= mmu.max_page_level; l++)
1219         total += (hat->hat_pages_mapped[l] << LEVEL_SHIFT(l));
1220     total += hat->hat_ism_pgcnt;

1222     return (total);
1223 }
_____unchanged_portion_omitted_____

```

new/usr/src/uts/i86xpv/vm/seg\_mf.c

1

\*\*\*\*\*

16753 Thu Feb 18 08:40:44 2016

new/usr/src/uts/i86xpv/vm/seg\_mf.c

6583 remove whole-process swapping

\*\*\*\*\*

unchanged\_portion\_omitted

```
760 static struct seg_ops segmf_ops = {
761     segmf_dup,
762     segmf_unmap,
763     segmf_free,
764     segmf_fault,
765     segmf_faulta,
766     segmf_setprot,
767     segmf_checkprot,
768     (int (*)())segmf_kluster,
769     (size_t (*)(struct seg *))NULL, /* swapout */
769     segmf_sync,
770     segmf_incore,
771     segmf_lockop,
772     segmf_getprot,
773     segmf_getoffset,
774     segmf_gettype,
775     segmf_getvp,
776     segmf_advise,
777     segmf_dump,
778     segmf_pagelock,
779     segmf_setpagesize,
780     segmf_getmemid,
781     segmf_getpolicy,
782     segmf_capable,
783     seg_inherit_notsup
784 };
```

unchanged\_portion\_omitted

```

*****
35882 Thu Feb 18 08:40:44 2016
new/usr/src/uts/intel/ia32/os/syscall.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

137 /*
138  * Called from syscall() when a non-trivial 32-bit system call occurs.
139  *   Sets up the args and returns a pointer to the handler.
140  */
141 struct sysent *
142 syscall_entry(kthread_t *t, long *argp)
143 {
144     klwp_t *lwp = ttolwp(t);
145     struct regs *rp = lwptoregs(lwp);
146     unsigned int code;
147     struct sysent *callp;
148     struct sysent *se = LWP_GETSYSENT(lwp);
149     int error = 0;
150     uint_t nargs;

152     ASSERT(t == curthread);
152     ASSERT(t == curthread && curthread->t_schedflag & TS_DONT_SWAP);

154     lwp->lwp_ru.sysc++;
155     lwp->lwp_eosys = NORMALRETURN; /* assume this will be normal */

157     /*
158     * Set lwp_ap to point to the args, even if none are needed for this
159     * system call. This is for the loadable-syscall case where the
160     * number of args won't be known until the system call is loaded, and
161     * also maintains a non-NULL lwp_ap setup for get_syscall_args(). Note
162     * that lwp_ap MUST be set to a non-NULL value BEFORE t_sysnum is
163     * set to non-zero; otherwise get_syscall_args(), seeing a non-zero
164     * t_sysnum for this thread, will charge ahead and dereference lwp_ap.
165     */
166     lwp->lwp_ap = argp; /* for get_syscall_args */

168     code = rp->r_r0;
169     t->t_sysnum = (short)code;
170     callp = code >= NSYSCALL ? &nosys_ent : se + code;

172     if ((t->t_pre_sys | syscalltrace) != 0) {
173         error = pre_syscall();

175         /*
176         * pre_syscall() has taken care so that lwp_ap is current;
177         * it either points to syscall-entry-saved amd64 regs,
178         * or it points to lwp_arg[], which has been re-copied from
179         * the ia32 ustack, but either way, it's a current copy after
180         * /proc has possibly mucked with the syscall args.
181         */

183         if (error)
184             return (&sysent_err); /* use dummy handler */
185     }

187     /*
188     * Fetch the system call arguments to the kernel stack copy used
189     * for syscall handling.
190     * Note: for loadable system calls the number of arguments required
191     * may not be known at this point, and will be zero if the system call
192     * was never loaded. Once the system call has been loaded, the number
193     * of args is not allowed to be changed.
194     */

```

```

195     if ((nargs = (uint_t)callp->sy_narg) != 0 &&
196         COPYIN_ARGS32(rp, argp, nargs)) {
197         (void) set_errno(EFAULT);
198         return (&sysent_err); /* use dummy handler */
199     }

201     return (callp); /* return sysent entry for caller */
202 }
_____unchanged_portion_omitted_____

227 /*
228  * Perform pre-system-call processing, including stopping for tracing,
229  * auditing, etc.
230  *
231  * This routine is called only if the t_pre_sys flag is set. Any condition
232  * requiring pre-syscall handling must set the t_pre_sys flag. If the
233  * condition is persistent, this routine will repost t_pre_sys.
234  */
235 int
236 pre_syscall()
237 {
238     kthread_t *t = curthread;
239     unsigned code = t->t_sysnum;
240     klwp_t *lwp = ttolwp(t);
241     proc_t *p = ttoproc(t);
242     int repost;

244     t->t_pre_sys = repost = 0; /* clear pre-syscall processing flag */

246     ASSERT(t->t_schedflag & TS_DONT_SWAP);

248 #if defined(DEBUG)
249     /*
250     * On the i386 kernel, lwp_ap points at the piece of the thread
251     * stack that we copy the users arguments into.
252     *
253     * On the amd64 kernel, the syscall arguments in the rdi..r9
254     * registers should be pointed at by lwp_ap. If the args need to
255     * be copied so that those registers can be changed without losing
256     * the ability to get the args for /proc, they can be saved by
257     * save_syscall_args(), and lwp_ap will be restored by post_syscall().
258     */
259     if (lwp_getdatamodel(lwp) == DATAMODEL_NATIVE) {
260         #if defined(LP64)
261             ASSERT(lwp->lwp_ap == (long *)&lwptoregs(lwp)->r_rdi);
262         } else {
263             ASSERT((caddr_t)lwp->lwp_ap > t->t_stkbase &&
264                 (caddr_t)lwp->lwp_ap < t->t_stk);
265         }
266     } /* DEBUG */

267     /*
268     * Make sure the thread is holding the latest credentials for the
269     * process. The credentials in the process right now apply to this
270     * thread for the entire system call.
271     */
272     if (t->t_cred != p->p_cred) {
273         cred_t *oldcred = t->t_cred;
274         /*
275         * DTrace accesses t_cred in probe context. t_cred must
276         * always be either NULL, or point to a valid, allocated cred
277         * structure.
278         */
279         t->t_cred = crgetcred();
280         crfree(oldcred);

```

```

281     }
282
283     /*
284     * From the proc(4) manual page:
285     * When entry to a system call is being traced, the traced process
286     * stops after having begun the call to the system but before the
287     * system call arguments have been fetched from the process.
288     */
289     if (PTOU(p)->u_systrap) {
290         if (prismember(&PTOU(p)->u_entrymask, code)) {
291             mutex_enter(&p->p_lock);
292             /*
293              * Recheck stop condition, now that lock is held.
294              */
295             if (PTOU(p)->u_systrap &&
296                 prismember(&PTOU(p)->u_entrymask, code)) {
297                 stop(PR_SYSENTRY, code);
298
299                 /*
300                  * /proc may have modified syscall args,
301                  * either in regs for amd64 or on ustack
302                  * for ia32. Either way, arrange to
303                  * copy them again, both for the syscall
304                  * handler and for other consumers in
305                  * post_syscall (like audit). Here, we
306                  * only do amd64, and just set lwp_ap
307                  * back to the kernel-entry stack copy;
308                  * the syscall ml code redoes
309                  * move-from-regs to set up for the
310                  * syscall handler after we return. For
311                  * ia32, save_syscall_args() below makes
312                  * an lwp_ap-accessible copy.
313                  */
314                 #if defined(LP64)
315                 if (lwp_getdatamodel(lwp) == DATAMODEL_NATIVE) {
316                     lwp->lwp_argsaved = 0;
317                     lwp->lwp_ap =
318                         (long *)&lwptoregs(lwp)->r_rdi;
319                 }
320             #endif
321             }
322             mutex_exit(&p->p_lock);
323         }
324         repost = 1;
325     }
326
327     /*
328     * ia32 kernel, or ia32 proc on amd64 kernel: keep args in
329     * lwp_arg for post-syscall processing, regardless of whether
330     * they might have been changed in /proc above.
331     */
332     #if defined(LP64)
333     if (lwp_getdatamodel(lwp) != DATAMODEL_NATIVE)
334     #endif
335         (void) save_syscall_args();
336
337     if (lwp->lwp_sysabort) {
338         /*
339          * lwp_sysabort may have been set via /proc while the process
340          * was stopped on PR_SYSENTRY. If so, abort the system call.
341          * Override any error from the copyin() of the arguments.
342          */
343         lwp->lwp_sysabort = 0;
344         (void) set_errno(EINTR); /* forces post_sys */
345         t->t_pre_sys = 1; /* repost anyway */
346         return (1); /* don't do system call, return EINTR */

```

```

347     }
348
349     /*
350     * begin auditing for this syscall if the c2audit module is loaded
351     * and auditing is enabled
352     */
353     if (audit_active == C2AUDIT_LOADED) {
354         uint32_t auditing = au_zone_getstate(NULL);
355
356         if (auditing & AU_AUDIT_MASK) {
357             int error;
358             if (error = audit_start(T_SYSCALL, code, auditing, \
359                 0, lwp)) {
360                 t->t_pre_sys = 1; /* repost anyway */
361                 (void) set_errno(error);
362                 return (1);
363             }
364             repost = 1;
365         }
366     }
367
368     #ifndef NPROBE
369     /* Kernel probe */
370     if (tnf_tracing_active) {
371         TNF_PROBE_1(syscall_start, "syscall thread", /* CSTYLE */
372             tnf_sysnum, sysnum, t->t_sysnum);
373         t->t_post_sys = 1; /* make sure post_syscall runs */
374         repost = 1;
375     }
376     #endif /* NPROBE */
377
378     #ifdef SYSCALLTRACE
379     if (syscalltrace) {
380         int i;
381         long *ap;
382         char *cp;
383         char *sysname;
384         struct sysent *callp;
385
386         if (code >= NSYSCALL)
387             callp = &nosys_ent; /* nosys has no args */
388         else
389             callp = LWP_GETSYSENT(lwp) + code;
390         (void) save_syscall_args();
391         mutex_enter(&systrace_lock);
392         printf("%d: ", p->p_pid);
393         if (code >= NSYSCALL)
394             printf("0x%x", code);
395         else {
396             sysname = mod_getsysname(code);
397             printf("%s[0x%x/0x%p]", sysname == NULL ? "NULL" :
398                 sysname, code, callp->sy_callc);
399         }
400         cp = "(";
401         for (i = 0, ap = lwp->lwp_ap; i < callp->sy_narg; i++, ap++) {
402             printf("%s%lx", cp, *ap);
403             cp = ", ";
404         }
405         if (i)
406             printf(")");
407         printf(" %s id=0x%p\n", PTOU(p)->u_commm, curthread);
408         mutex_exit(&systrace_lock);
409     }
410     #endif /* SYSCALLTRACE */
411
412     /*

```

```
413     * If there was a continuing reason for pre-syscall processing,
414     * set the t_pre_sys flag for the next system call.
415     */
416     if (repost)
417         t->t_pre_sys = 1;
418     lwp->lwp_error = 0; /* for old drivers */
419     lwp->lwp_badpriv = PRIV_NONE;
420     return (0);
421 }
_____unchanged_portion_omitted_____
```

```

*****
413119 Thu Feb 18 08:40:44 2016
new/usr/src/uts/sfmmu/vm/hat_sfmmu.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

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 }

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     }; /* free list of TSBs */
2002     struct free_tsb *freelist, *last, *next;

2004     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) {

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_vcnt && !hmeblkp->hblk_hmccnt) {
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 }

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);

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 }

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 }

2115 /*
2173  * Duplicate the translations of an as into another newas
2174  */
2175 /* ARGSUSED */
2176 int
2177 hat_dup(struct hat *hat, struct hat *newhat, caddr_t addr, size_t len,
2178         uint_t flag)
2179 {
2180     sf_srd_t *srdp;
2181     sf_scd_t *scdp;
2182     int i;
2183     extern uint_t get_color_start(struct as *);

2185     ASSERT((flag == 0) || (flag == HAT_DUP_ALL) || (flag == HAT_DUP_COW) ||
2186           (flag == HAT_DUP_SRD));
2187     ASSERT(hat != ksfmmup);
2188     ASSERT(newhat != ksfmmup);
2189     ASSERT(flag != HAT_DUP_ALL || hat->sfmmu_srdp == newhat->sfmmu_srdp);

2191     if (flag == HAT_DUP_COW) {
2192         panic("hat_dup: HAT_DUP_COW not supported");
2193     }

2195     if (flag == HAT_DUP_SRD && ((srdp = hat->sfmmu_srdp) != NULL)) {
2196         ASSERT(srdp->srd_evp != NULL);
2197         VN_HOLD(srdp->srd_evp);
2198         ASSERT(srdp->srd_refcnt > 0);
2199         newhat->sfmmu_srdp = srdp;
2200         atomic_inc_32((volatile uint_t *)&srdp->srd_refcnt);
2201     }

2203     /*
2204     * HAT_DUP_ALL flag is used after as duplication is done.
2205     */
2206     if (flag == HAT_DUP_ALL && ((srdp = newhat->sfmmu_srdp) != NULL)) {
2207         ASSERT(newhat->sfmmu_srdp->srd_refcnt >= 2);
2208         newhat->sfmmu_rtteflags = hat->sfmmu_rtteflags;
2209         if (hat->sfmmu_flags & HAT_4MTEXT_FLAG) {
2210             newhat->sfmmu_flags |= HAT_4MTEXT_FLAG;
2211         }

2213         /* check if need to join scd */
2214         if ((scdp = hat->sfmmu_scdp) != NULL &&
2215             newhat->sfmmu_scdp != scdp) {
2216             int ret;
2217             SF_RGNMAP_IS_SUBSET(&newhat->sfmmu_region_map,
2218                               &scdp->scd_region_map, ret);
2219             ASSERT(ret);

```

```

2200         sfmmu_join_scd(scdp, newhat);
2201         ASSERT(newhat->sfmmu_scdp == scdp &&
2202               scdp->scd_refcnt >= 2);
2203         for (i = 0; i < max_mmu_page_sizes; i++) {
2204             newhat->sfmmu_ismttecnt[i] =
2205                 hat->sfmmu_ismttecnt[i];
2206             newhat->sfmmu_scdismttecnt[i] =
2207                 hat->sfmmu_scdismttecnt[i];
2208         }
2209     }

2231     sfmmu_check_page_sizes(newhat, 1);
2232 }

2234     if (flag == HAT_DUP_ALL && consistent_coloring == 0 &&
2235         update_proc_pcolorbase_after_fork != 0) {
2236         hat->sfmmu_clrbin = get_color_start(hat->sfmmu_as);
2237     }
2238     return (0);
2239 }

_____unchanged_portion_omitted_____

9668 /*
9669  * Replace the specified TSB with a new TSB. This function gets called when
9670  * we grow, or shrink a TSB. When swapping in a TSB (TSB_SWAPIN), the
9671  * we grow, shrink or swapin a TSB. When swapping in a TSB (TSB_SWAPIN), the
9672  * TSB_FORCEALLOC flag may be used to force allocation of a minimum-sized TSB
9673  * (8K).
9674  * Caller must hold the HAT lock, but should assume any tsb_info
9675  * pointers it has are no longer valid after calling this function.
9676  *
9677  * Return values:
9678  *   TSB_ALLOCFAIL   Failed to allocate a TSB, due to memory constraints
9679  *   TSB_LOSTRACE   HAT is busy, i.e. another thread is already doing
9680  *                   something to this tsbinfo/TSB
9681  *   TSB_SUCCESS    Operation succeeded
9682  */
9683 static tsb_replace_rc_t
9684 sfmmu_replace_tsb(sfmmu_t *sfmmup, struct tsb_info *old_tsbinfo, uint_t szc,
9685                  hatlock_t *hatlockp, uint_t flags)
9686 {
9687     struct tsb_info *new_tsbinfo = NULL;
9688     struct tsb_info *curtsb, *prevtsb;
9689     uint_t tte_sz_mask;
9690     int i;

9692     ASSERT(sfmmup != ksfmmup);
9693     ASSERT(sfmmup->sfmmu_ismhat == 0);
9694     ASSERT(sfmmu_hat_lock_held(sfmmup));
9695     ASSERT(szc <= tsb_max_growsize);

9697     if (SFMMU_FLAGS_ISSET(sfmmup, HAT_BUSY))
9698         return (TSB_LOSTRACE);

9700     /*
9701     * Find the tsb_info ahead of this one in the list, and
9702     * also make sure that the tsb_info passed in really
9703     * exists!
9704     */
9705     for (prevtsb = NULL, curtsb = sfmmup->sfmmu_tsb;
9706         curtsb != old_tsbinfo && curtsb != NULL;
9707         prevtsb = curtsb, curtsb = curtsb->tsb_next)
9708         ;
9709     ASSERT(curtsb != NULL);

```

```

9711     if (!(flags & TSB_SWAPIN) && SFMMU_FLAGS_ISSET(sfmmup, HAT_SWAPPED)) {
9712         /*
9713          * The process is swapped out, so just set the new size
9714          * code. When it swaps back in, we'll allocate a new one
9715          * of the new chosen size.
9716          */
9717         curtsb->tsb_szc = szc;
9718         return (TSB_SUCCESS);
9719     }
9720     SFMMU_FLAGS_SET(sfmmup, HAT_BUSY);

9722     tte_sz_mask = old_tsbinfo->tsb_ttesz_mask;

9724     /*
9725     * All initialization is done inside of sfmmu_tsbinfo_alloc().
9726     * If we fail to allocate a TSB, exit.
9727     *
9728     * If tsb grows with new tsb size > 4M and old tsb size < 4M,
9729     * then try 4M slab after the initial alloc fails.
9730     *
9731     * If tsb swapin with tsb size > 4M, then try 4M after the
9732     * initial alloc fails.
9733     */
9734     sfmmu_hat_exit(hatlockp);
9735     if (sfmmu_tsbinfo_alloc(&new_tsbinfo, szc,
9736         tte_sz_mask, flags, sfmmup) &&
9737         (!(flags & (TSB_GROW | TSB_SWAPIN)) || (szc <= TSB_4M_SZCODE) ||
9738         (!(flags & TSB_SWAPIN) &&
9739         (old_tsbinfo->tsb_szc >= TSB_4M_SZCODE)) ||
9740         sfmmu_tsbinfo_alloc(&new_tsbinfo, TSB_4M_SZCODE,
9741         tte_sz_mask, flags, sfmmup))) {
9742         (void) sfmmu_hat_enter(sfmmup);
9743         if (!(flags & TSB_SWAPIN))
9744             SFMMU_STAT(sf_tsb_resize_failures);
9745         SFMMU_FLAGS_CLEAR(sfmmup, HAT_BUSY);
9746         return (TSB_ALLOCFAIL);
9747     }
9748     (void) sfmmu_hat_enter(sfmmup);

9750     /*
9751     * Re-check to make sure somebody else didn't muck with us while we
9752     * didn't hold the HAT lock.  If the process swapped out, fine, just
9753     * exit; this can happen if we try to shrink the TSB from the context
9754     * of another process (such as on an ISM unmap), though it is rare.
9755     */
9756     if (!(flags & TSB_SWAPIN) && SFMMU_FLAGS_ISSET(sfmmup, HAT_SWAPPED)) {
9757         SFMMU_STAT(sf_tsb_resize_failures);
9758         SFMMU_FLAGS_CLEAR(sfmmup, HAT_BUSY);
9759         sfmmu_hat_exit(hatlockp);
9760         sfmmu_tsbinfo_free(new_tsbinfo);
9761         (void) sfmmu_hat_enter(sfmmup);
9762         return (TSB_LOSTRACE);
9763     }

9765 #ifdef DEBUG
9766     /* Reverify that the tsb_info still exists.. for debugging only */
9767     for (prevtsb = NULL, curtsb = sfmmup->sfmmu_tsb;
9768         curtsb != old_tsbinfo && curtsb != NULL;
9769         prevtsb = curtsb, curtsb = curtsb->tsb_next)
9770         ;
9771     ASSERT(curtsb != NULL);
9772 #endif /* DEBUG */

9774     /*
9775     * Quiesce any CPUs running this process on their next TLB miss

```

```

9776     * so they atomically see the new tsb_info.  We temporarily set the
9777     * context to invalid context so new threads that come on processor
9778     * after we do the xcall to cpusran will also serialize behind the
9779     * HAT lock on TLB miss and will see the new TSB.  Since this short
9780     * race with a new thread coming on processor is relatively rare,
9781     * this synchronization mechanism should be cheaper than always
9782     * pausing all CPUs for the duration of the setup, which is what
9783     * the old implementation did.  This is particularly true if we are
9784     * copying a huge chunk of memory around during that window.
9785     *
9786     * The memory barriers are to make sure things stay consistent
9787     * with resume() since it does not hold the HAT lock while
9788     * walking the list of tsb_info structures.
9789     */
9790     if ((flags & TSB_SWAPIN) != TSB_SWAPIN) {
9791         /* The TSB is either growing or shrinking. */
9792         sfmmu_invalidate_ctx(sfmmup);
9793     } else {
9794         /*
9795          * It is illegal to swap in TSBs from a process other
9796          * than a process being swapped in.  This in turn
9797          * implies we do not have a valid MMU context here
9798          * since a process needs one to resolve translation
9799          * misses.
9800          */
9801         ASSERT(curthread->t_procp->p_as->a_hat == sfmmup);
9802     }

9804 #ifdef DEBUG
9805     ASSERT(max_mmu_ctxdoms > 0);

9807     /*
9808     * Process should have INVALID_CONTEXT on all MMUs
9809     */
9810     for (i = 0; i < max_mmu_ctxdoms; i++) {

9812         ASSERT(sfmmup->sfmmu_ctxs[i].cnum == INVALID_CONTEXT);
9813     }
9814 #endif

9816     new_tsbinfo->tsb_next = old_tsbinfo->tsb_next;
9817     membar_stst(); /* strict ordering required */
9818     if (prevtsb)
9819         prevtsb->tsb_next = new_tsbinfo;
9820     else
9821         sfmmup->sfmmu_tsb = new_tsbinfo;
9822     membar_enter(); /* make sure new TSB globally visible */

9824     /*
9825     * We need to migrate TSB entries from the old TSB to the new TSB
9826     * if tsb_remap_ttes is set and the TSB is growing.
9827     */
9828     if (tsb_remap_ttes && ((flags & TSB_GROW) == TSB_GROW))
9829         sfmmu_copy_tsb(old_tsbinfo, new_tsbinfo);

9831     SFMMU_FLAGS_CLEAR(sfmmup, HAT_BUSY);

9833     /*
9834     * Drop the HAT lock to free our old tsb_info.
9835     */
9836     sfmmu_hat_exit(hatlockp);

9838     if ((flags & TSB_GROW) == TSB_GROW) {
9839         SFMMU_STAT(sf_tsb_grow);
9840     } else if ((flags & TSB_SHRINK) == TSB_SHRINK) {
9841         SFMMU_STAT(sf_tsb_shrink);

```

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

7

```
9842     }
9844     sfmmu_tsbinfo_free(old_tsbinfo);
9846     (void) sfmmu_hat_enter(sfmmup);
9847     return (TSB_SUCCESS);
9848 }
unchanged_portion_omitted
```

```

*****
31077 Thu Feb 18 08:40:45 2016
new/usr/src/uts/sparc/os/syscall.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

342 /*
343  * Perform pre-system-call processing, including stopping for tracing,
344  * auditing, microstate-accounting, etc.
345  *
346  * This routine is called only if the t_pre_sys flag is set. Any condition
347  * requiring pre-syscall handling must set the t_pre_sys flag. If the
348  * condition is persistent, this routine will repost t_pre_sys.
349  */
350 int
351 pre_syscall(int arg0)
352 {
353     unsigned int code;
354     kthread_t *t = curthread;
355     proc_t *p = ttoproc(t);
356     klpw_t *lwp = ttolwp(t);
357     struct regs *rp = lwptoregs(lwp);
358     int repost;

360     t->t_pre_sys = repost = 0;    /* clear pre-syscall processing flag */

362     ASSERT(t->t_schedflag & TS_DONT_SWAP);

362     syscall_mstate(LMS_USER, LMS_SYSTEM);

364     /*
365     * The syscall arguments in the out registers should be pointed to
366     * by lwp_ap. If the args need to be copied so that the outs can
367     * be changed without losing the ability to get the args for /proc,
368     * they can be saved by save_syscall_args(), and lwp_ap will be
369     * restored by post_syscall().
370     */
371     ASSERT(lwp->lwp_ap == (long *)&rp->r_o0);

373     /*
374     * Make sure the thread is holding the latest credentials for the
375     * process. The credentials in the process right now apply to this
376     * thread for the entire system call.
377     */
378     if (t->t_cred != p->p_cred) {
379         cred_t *oldcred = t->t_cred;
380         /*
381         * DTrace accesses t_cred in probe context. t_cred must
382         * always be either NULL, or point to a valid, allocated cred
383         * structure.
384         */
385         t->t_cred = crgetcred();
386         crfree(oldcred);
387     }

389     /*
390     * Undo special arrangements to single-step the lwp
391     * so that a debugger will see valid register contents.
392     * Also so that the pc is valid for syncfpu().
393     * Also so that a syscall like exec() can be stepped.
394     */
395     if (lwp->lwp_pcb.pcb_step != STEP_NONE) {
396         (void) prundostep();
397         repost = 1;
398     }

```

```

400     /*
401     * Check for indirect system call in case we stop for tracing.
402     * Don't allow multiple indirection.
403     */
404     code = t->t_sysnum;
405     if (code == 0 && arg0 != 0) {    /* indirect syscall */
406         code = arg0;
407         t->t_sysnum = arg0;
408     }

410     /*
411     * From the proc(4) manual page:
412     * When entry to a system call is being traced, the traced process
413     * stops after having begun the call to the system but before the
414     * system call arguments have been fetched from the process.
415     * If proc changes the args we must refetch them after starting.
416     */
417     if (PTOU(p)->u_systrap) {
418         if (prismember(&PTOU(p)->u_entrymask, code)) {
419             /*
420             * Recheck stop condition, now that lock is held.
421             */
422             mutex_enter(&p->p_lock);
423             if (PTOU(p)->u_systrap &&
424                 prismember(&PTOU(p)->u_entrymask, code)) {
425                 stop(PR_SYSENTRY, code);
426                 /*
427                 * Must refetch args since they were
428                 * possibly modified by /proc. Indicate
429                 * that the valid copy is in the
430                 * registers.
431                 */
432                 lwp->lwp_argsaved = 0;
433                 lwp->lwp_ap = (long *)&rp->r_o0;
434             }
435             mutex_exit(&p->p_lock);
436         }
437         repost = 1;
438     }

440     if (lwp->lwp_sysabort) {
441         /*
442         * lwp_sysabort may have been set via /proc while the process
443         * was stopped on PR_SYSENTRY. If so, abort the system call.
444         * Override any error from the copyin() of the arguments.
445         */
446         lwp->lwp_sysabort = 0;
447         (void) set_errno(EINTR); /* sets post-sys processing */
448         t->t_pre_sys = 1;    /* repost anyway */
449         return (1);    /* don't do system call, return EINTR */
450     }

452     /* begin auditing for this syscall */
453     if (audit_active == C2AUDIT_LOADED) {
454         uint32_t auditing = au_zone_getstate(NULL);

456         if (auditing & AU_AUDIT_MASK) {
457             int error;
458             if (error = audit_start(T_SYSCALL, code, auditing, \
459                 0, lwp)) {
460                 t->t_pre_sys = 1;    /* repost anyway */
461                 lwp->lwp_error = 0; /* for old drivers */
462                 return (error);
463             }
464             repost = 1;

```

```

465     }
466 }

468 #ifndef NPROBE
469 /* Kernel probe */
470 if (tnf_tracing_active) {
471     TNF_PROBE_1(syscall_start, "syscall thread", /* CSTYLEL */,
472               tnf_sysnum, sysnum, t->t_sysnum);
473     t->t_post_sys = 1; /* make sure post_syscall runs */
474     repost = 1;
475 }
476 #endif /* NPROBE */

478 #ifdef SYSCALLTRACE
479 if (syscalltrace) {
480     int i;
481     long *ap;
482     char *cp;
483     char *sysname;
484     struct sysent *callp;

486     if (code >= NSYSCALL)
487         callp = &nosys_ent; /* nosys has no args */
488     else
489         callp = LWP_GETSYSENT(lwp) + code;
490     (void) save_syscall_args();
491     mutex_enter(&systrace_lock);
492     printf("%d: ", p->p_pid);
493     if (code >= NSYSCALL)
494         printf("0x%x", code);
495     else {
496         sysname = mod_getsysname(code);
497         printf("%s[0x%x]", sysname == NULL ? "NULL" :
498               sysname, code);
499     }
500     cp = "(";
501     for (i = 0, ap = lwp->lwp_ap; i < callp->sy_narg; i++, ap++) {
502         printf("%s%lx", cp, *ap);
503         cp = ", ";
504     }
505     if (i)
506         printf(")");
507     printf(" %s id=0x%p\n", PTOU(p)->u_comm, curthread);
508     mutex_exit(&systrace_lock);
509 }
510 #endif /* SYSCALLTRACE */

512 /*
513  * If there was a continuing reason for pre-syscall processing,
514  * set the t_pre_sys flag for the next system call.
515  */
516 if (repost)
517     t->t_pre_sys = 1;
518 lwp->lwp_error = 0; /* for old drivers */
519 lwp->lwp_badpriv = PRIV_NONE; /* for privilege tracing */
520 return (0);
521 }

```

unchanged portion omitted

```

*****
50041 Thu Feb 18 08:40:45 2016
new/usr/src/uts/sparc/v9/os/v9dep.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

863 void
864 lwp_swapin(kthread_t *tp)
865 {
866     struct machpcb *mpcb = lwptompcb(ttolwp(tp));

868     mpcb->mpcb_pa = va_to_pa(mpcb);
869     mpcb->mpcb_wbuf_pa = va_to_pa(mpcb->mpcb_wbuf);
870 }

863 /*
864  * Construct the execution environment for the user's signal
865  * handler and arrange for control to be given to it on return
866  * to userland. The library code now calls setcontext() to
867  * clean up after the signal handler, so sigret() is no longer
868  * needed.
869  */
870 int
871 sendsig(int sig, k_siginfo_t *sip, void (*hdlr)())
872 {
873     /*
874      * 'volatile' is needed to ensure that values are
875      * correct on the error return from on_fault().
876      */
877     volatile int minstacksz; /* min stack required to catch signal */
878     int newstack = 0; /* if true, switching to altstack */
879     label_t ljb;
880     caddr_t sp;
881     struct regs *volatile rp;
882     klwp_t *lwp = ttolwp(curthread);
883     proc_t *volatile p = ttoproc(curthread);
884     int fpq_size = 0;
885     struct sigframe {
886         struct frame frwin;
887         ucontext_t uc;
888     };
889     siginfo_t *sip_addr;
890     struct sigframe *volatile fp;
891     ucontext_t *volatile tuc = NULL;
892     char *volatile xregs = NULL;
893     volatile size_t xregs_size = 0;
894     gwindows_t *volatile gwp = NULL;
895     volatile int gwin_size = 0;
896     kfpu_t *fpp;
897     struct machpcb *mpcb;
898     volatile int watched = 0;
899     volatile int watched2 = 0;
900     caddr_t tos;

902     /*
903      * Make sure the current last user window has been flushed to
904      * the stack save area before we change the sp.
905      * Restore register window if a debugger modified it.
906      */
907     (void) flush_user_windows_to_stack(NULL);
908     if (lwp->lwp_pcb.pcb_xregstat != XREGNONE)
909         xregrestore(lwp, 0);

911     mpcb = lwptompcb(lwp);
912     rp = lwptoregs(lwp);

```

```

914     /*
915      * Clear the watchpoint return stack pointers.
916      */
917     mpcb->mpcb_rsp[0] = NULL;
918     mpcb->mpcb_rsp[1] = NULL;

920     minstacksz = sizeof (struct sigframe);

922     /*
923      * We know that sizeof (siginfo_t) is stack-aligned:
924      * 128 bytes for ILP32, 256 bytes for LP64.
925      */
926     if (sip != NULL)
927         minstacksz += sizeof (siginfo_t);

929     /*
930      * These two fields are pointed to by ABI structures and may
931      * be of arbitrary length. Size them now so we know how big
932      * the signal frame has to be.
933      */
934     fpp = lwptofpu(lwp);
935     fpp->fpu_fprs = _fp_read_fprs();
936     if ((fpp->fpu_en) || (fpp->fpu_fprs & FPRS_FEF)) {
937         fpq_size = fpp->fpu_q_entrysize * fpp->fpu_qcnt;
938         minstacksz += SA(fpq_size);
939     }

941     mpcb = lwptompcb(lwp);
942     if (mpcb->mpcb_wbcnt != 0) {
943         gwin_size = (mpcb->mpcb_wbcnt * sizeof (struct rwindow) +
944                     (SPARC_MAXREGWINDOW * sizeof (caddr_t)) + sizeof (long));
945         minstacksz += SA(gwin_size);
946     }

948     /*
949      * Extra registers, if support by this platform, may be of arbitrary
950      * length. Size them now so we know how big the signal frame has to be.
951      * For sparcv9_LP64 user programs, use asrs instead of the xregs.
952      */
953     minstacksz += SA(xregs_size);

955     /*
956      * Figure out whether we will be handling this signal on
957      * an alternate stack specified by the user. Then allocate
958      * and validate the stack requirements for the signal handler
959      * context. on_fault will catch any faults.
960      */
961     newstack = (sigismember(&PTOU(curproc)->u_sigonstack, sig) &&
962               !(lwp->lwp_sigaltstack.ss_flags & (SS_ONSTACK|SS_DISABLE)));

964     tos = (caddr_t)rp->r_sp + STACK_BIAS;
965     /*
966      * Force proper stack pointer alignment, even in the face of a
967      * misaligned stack pointer from user-level before the signal.
968      * Don't use the SA() macro because that rounds up, not down.
969      */
970     tos = (caddr_t)((uintptr_t)tos & ~(STACK_ALIGN - 1ul));

972     if (newstack != 0) {
973         fp = (struct sigframe *)
974             (SA((uintptr_t)lwp->lwp_sigaltstack.ss_sp) +
975              SA((int)lwp->lwp_sigaltstack.ss_size) - STACK_ALIGN -
976              SA(minstacksz));
977     } else {
978         /*

```

```

979     * If we were unable to flush all register windows to
980     * the stack and we are not now on an alternate stack,
981     * just dump core with a SIGSEGV back in psig().
982     */
983     if (sig == SIGSEGV &&
984         mpcb->mpcb_wbcnt != 0 &&
985         !(lwp->lwp_sigaltstack.ss_flags & SS_ONSTACK))
986         return (0);
987     fp = (struct sigframe *) (tos - SA(minstacksz));
988     /*
989     * Could call grow here, but stack growth now handled below
990     * in code protected by on_fault().
991     */
992 }
993 sp = (caddr_t)fp + sizeof (struct sigframe);

995 /*
996 * Make sure process hasn't trashed its stack.
997 */
998 if ((caddr_t)fp >= p->p_usrstack ||
999     (caddr_t)fp + SA(minstacksz) >= p->p_usrstack) {
1000 #ifdef DEBUG
1001     printf("sendsig: bad signal stack cmd=%s, pid=%d, sig=%d\n",
1002           PTOU(p)->u_comm, p->p_pid, sig);
1003     printf("sigsp = 0x%p, action = 0x%p, upc = 0x%lx\n",
1004           (void *)fp, (void *)hdlr, rp->r_pc);
1005     printf("fp above USRSTACK\n");
1006 #endif
1007     return (0);
1008 }

1010 watched = watch_disable_addr((caddr_t)fp, SA(minstacksz), S_WRITE);
1011 if (on_fault(&ljb))
1012     goto badstack;

1014 tuc = kmem_alloc(sizeof (ucontext_t), KM_SLEEP);
1015 savecontext(tuc, &lwp->lwp_sigoldmask);

1017 /*
1018 * save extra register state if it exists
1019 */
1020 if (xregs_size != 0) {
1021     xregs_setptr(lwp, tuc, sp);
1022     xregs = kmem_alloc(xregs_size, KM_SLEEP);
1023     xregs_get(lwp, xregs);
1024     copyout_noerr(xregs, sp, xregs_size);
1025     kmem_free(xregs, xregs_size);
1026     xregs = NULL;
1027     sp += SA(xregs_size);
1028 }

1030 copyout_noerr(tuc, &fp->uc, sizeof (*tuc));
1031 kmem_free(tuc, sizeof (*tuc));
1032 tuc = NULL;

1034 if (sip != NULL) {
1035     zoneid_t zoneid;

1037     uzero(sp, sizeof (siginfo_t));
1038     if (SI_FROMUSER(sip) &&
1039         (zoneid = p->p_zone->zone_id) != GLOBAL_ZONEID &&
1040         zoneid != sip->si_zoneid) {
1041         k_siginfo_t sani_sip = *sip;
1042         sani_sip.si_pid = p->p_zone->zone_zsched->p_pid;
1043         sani_sip.si_uid = 0;
1044         sani_sip.si_ctid = -1;

```

```

1045         sani_sip.si_zoneid = zoneid;
1046         copyout_noerr(&sani_sip, sp, sizeof (sani_sip));
1047     } else {
1048         copyout_noerr(sip, sp, sizeof (*sip));
1049     }
1050     sip_addr = (siginfo_t *)sp;
1051     sp += sizeof (siginfo_t);

1053     if (sig == SIGPROF &&
1054         curthread->t_rprof != NULL &&
1055         curthread->t_rprof->rp_anystate) {
1056         /*
1057         * We stand on our head to deal with
1058         * the real time profiling signal.
1059         * Fill in the stuff that doesn't fit
1060         * in a normal k_siginfo structure.
1061         */
1062         int i = sip->si_sysarg;
1063         while (--i >= 0) {
1064             sulword_noerr(
1065                 (ulong_t *)&sip_addr->si_sysarg[i],
1066                 (ulong_t)lwp->lwp_arg[i]);
1067         }
1068         copyout_noerr(curthread->t_rprof->rp_state,
1069                     sip_addr->si_mstate,
1070                     sizeof (curthread->t_rprof->rp_state));
1071     }
1072 } else {
1073     sip_addr = (siginfo_t *)NULL;
1074 }

1076 /*
1077 * When flush_user_windows_to_stack() can't save all the
1078 * windows to the stack, it puts them in the lwp's pcb.
1079 */
1080 if (gwin_size != 0) {
1081     gwp = kmem_alloc(gwin_size, KM_SLEEP);
1082     getgwins(lwp, gwp);
1083     sulword_noerr(&fp->uc.mcontext.gwins, (ulong_t)sp);
1084     copyout_noerr(gwp, sp, gwin_size);
1085     kmem_free(gwp, gwin_size);
1086     gwp = NULL;
1087     sp += SA(gwin_size);
1088 } else
1089     sulword_noerr(&fp->uc.mcontext.gwins, (ulong_t)NULL);

1091 if (fpq_size != 0) {
1092     struct_fq *fq = (struct_fq *)sp;
1093     sulword_noerr(&fp->uc.mcontext.fpregs.fpu_q, (ulong_t)fq);
1094     copyout_noerr(mpcb->mpcb_fpu_q, fq, fpq_size);

1096     /*
1097     * forget the fp queue so that the signal handler can run
1098     * without being harrassed--it will do a setcontext that will
1099     * re-establish the queue if there still is one
1100     *
1101     * NOTE: fp_runq() relies on the qcnt field being zeroed here
1102     * to terminate its processing of the queue after signal
1103     * delivery.
1104     */
1105     mpcb->mpcb_fpu->fpu_qcnt = 0;
1106     sp += SA(fpq_size);

1108     /* Also, syscall needs to know about this */
1109     mpcb->mpcb_flags |= FP_TRAPPED;

```

```

1111     } else {
1112         sulword_noerr(&fp->uc_mcontext.fpregs.fpu_q, (ulong_t)NULL);
1113         suword8_noerr(&fp->uc_mcontext.fpregs.fpu_qcnt, 0);
1114     }

1117     /*
1118     * Since we flushed the user's windows and we are changing his
1119     * stack pointer, the window that the user will return to will
1120     * be restored from the save area in the frame we are setting up.
1121     * We copy in save area for old stack pointer so that debuggers
1122     * can do a proper stack backtrace from the signal handler.
1123     */
1124     if (mpcb->mpcb_wbcnt == 0) {
1125         watched2 = watch_disable_addr(tos, sizeof (struct rwindow),
1126             S_READ);
1127         ucopy(tos, &fp->frwin, sizeof (struct rwindow));
1128     }

1130     lwp->lwp_oldcontext = (uintptr_t)&fp->uc;

1132     if (newstack != 0) {
1133         lwp->lwp_sigaltstack.ss_flags |= SS_ONSTACK;

1135         if (lwp->lwp_ustack) {
1136             copyout_noerr(&lwp->lwp_sigaltstack,
1137                 (stack_t *)lwp->lwp_ustack, sizeof (stack_t));
1138         }
1139     }

1141     no_fault();
1142     mpcb->mpcb_wbcnt = 0;          /* let user go on */

1144     if (watched2)
1145         watch_enable_addr(tos, sizeof (struct rwindow), S_READ);
1146     if (watched)
1147         watch_enable_addr((caddr_t)fp, SA(minstacksz), S_WRITE);

1149     /*
1150     * Set up user registers for execution of signal handler.
1151     */
1152     rp->r_sp = (uintptr_t)fp - STACK_BIAS;
1153     rp->r_pc = (uintptr_t)hdlr;
1154     rp->r_npc = (uintptr_t)hdlr + 4;
1155     /* make sure %asi is ASI_PNF */
1156     rp->r_tstate &= ~(uint64_t)TSTATE_ASI_MASK << TSTATE_ASI_SHIFT;
1157     rp->r_tstate |= ((uint64_t)ASI_PNF << TSTATE_ASI_SHIFT);
1158     rp->r_o0 = sig;
1159     rp->r_o1 = (uintptr_t)sip_addr;
1160     rp->r_o2 = (uintptr_t)&fp->uc;
1161     /*
1162     * Don't set lwp_eosys here.  sendsig() is called via psig() after
1163     * lwp_eosys is handled, so setting it here would affect the next
1164     * system call.
1165     */
1166     return (1);

1168 badstack:
1169     no_fault();
1170     if (watched2)
1171         watch_enable_addr(tos, sizeof (struct rwindow), S_READ);
1172     if (watched)
1173         watch_enable_addr((caddr_t)fp, SA(minstacksz), S_WRITE);
1174     if (tuc)
1175         kmem_free(tuc, sizeof (ucontext_t));
1176     if (xregs)

```

```

1177         kmem_free(xregs, xregs_size);
1178         if (gwp)
1179             kmem_free(gwp, gwin_size);
1180 #ifdef DEBUG
1181         printf("sendsig: bad signal stack cmd=%s, pid=%d, sig=%d\n",
1182             PTOU(p)->u_comm, p->p_pid, sig);
1183         printf("on fault, sigsp = %p, action = %p, upc = 0x%lx\n",
1184             (void *)fp, (void *)hdlr, rp->r_pc);
1185 #endif
1186         return (0);
1187     }
    _____ unchanged portion omitted

```



```

*****
12166 Thu Feb 18 08:40:45 2016
new/usr/src/uts/sparc/v9/vm/seg_nf.c
6583 remove whole-process swapping
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2006 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 /* Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T */
27 /* All Rights Reserved */

29 /*
30 * Portions of this source code were derived from Berkeley 4.3 BSD
31 * under license from the Regents of the University of California.
32 */

34 /*
35 * VM - segment for non-faulting loads.
36 */

38 #include <sys/types.h>
39 #include <sys/t_lock.h>
40 #include <sys/param.h>
41 #include <sys/mman.h>
42 #include <sys/errno.h>
43 #include <sys/kmem.h>
44 #include <sys/cmn_err.h>
45 #include <sys/vnode.h>
46 #include <sys/proc.h>
47 #include <sys/conf.h>
48 #include <sys/debug.h>
49 #include <sys/archsystem.h>
50 #include <sys/lgrp.h>

52 #include <vm/page.h>
53 #include <vm/hat.h>
54 #include <vm/as.h>
55 #include <vm/seg.h>
56 #include <vm/vpage.h>

58 /*
59 * Private seg op routines.
60 */
61 static int      segnf_dup(struct seg *seg, struct seg *newseg);

```

```

62 static int      segnf_unmap(struct seg *seg, caddr_t addr, size_t len);
63 static void     segnf_free(struct seg *seg);
64 static faultcode_t segnf_nomap(void);
65 static int      segnf_setprot(struct seg *seg, caddr_t addr,
66                             size_t len, uint_t prot);
67 static int      segnf_checkprot(struct seg *seg, caddr_t addr,
68                                size_t len, uint_t prot);
69 static void     segnf_badop(void);
70 static int      segnf_nop(void);
71 static int      segnf_getprot(struct seg *seg, caddr_t addr,
72                               size_t len, uint_t *protv);
73 static u_offset_t segnf_getoffset(struct seg *seg, caddr_t addr);
74 static int      segnf_gettype(struct seg *seg, caddr_t addr);
75 static int      segnf_getvp(struct seg *seg, caddr_t addr, struct vnode **vpp);
76 static void     segnf_dump(struct seg *seg);
77 static int      segnf_pagelock(struct seg *seg, caddr_t addr, size_t len,
78                               struct page ***ppp, enum lock_type type, enum seg_rw rw);
79 static int      segnf_setpagesize(struct seg *seg, caddr_t addr, size_t len,
80                                  uint_t szc);
81 static int      segnf_getmemid(struct seg *seg, caddr_t addr, memid_t *memidp);
82 static lgrp_mem_policy_info_t *segnf_getpolicy(struct seg *seg,
83                                                 caddr_t addr);

86 struct seg_ops segnf_ops = {
87     segnf_dup,
88     segnf_unmap,
89     segnf_free,
90     (faultcode_t (*)(struct hat *, struct seg *, caddr_t, size_t,
91                     enum fault_type, enum seg_rw))
92     segnf_nomap, /* fault */
93     (faultcode_t (*)(struct seg *, caddr_t))
94     segnf_nomap, /* faulta */
95     segnf_setprot,
96     segnf_checkprot,
97     (int (*)( )) segnf_badop, /* kluster */
98     (size_t (*)(struct seg *)) NULL, /* swapout */
99     (int (*)(struct seg *, caddr_t, size_t, int, uint_t))
100     segnf_nop, /* sync */
101     (size_t (*)(struct seg *, caddr_t, size_t, char *))
102     segnf_nop, /* incore */
103     (int (*)(struct seg *, caddr_t, size_t, int, int, ulong_t *, size_t))
104     segnf_nop, /* lockop */
105     segnf_getprot,
106     segnf_getoffset,
107     segnf_gettype,
108     segnf_getvp,
109     (int (*)(struct seg *, caddr_t, size_t, uint_t))
110     segnf_nop, /* advise */
111     segnf_dump,
112     segnf_pagelock,
113     segnf_setpagesize,
114     segnf_getmemid,
115     segnf_getpolicy,
116 };

```

unchanged portion omitted

```
*****
```

```
14116 Thu Feb 18 08:40:45 2016
```

```
new/usr/src/uts/sun4/os/mlsetup.c
```

```
6583 remove whole-process swapping
```

```
*****
```

```
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 #include <sys/types.h>
27 #include <sys/system.h>
28 #include <sys/archsystem.h>
29 #include <sys/machsystem.h>
30 #include <sys/disp.h>
31 #include <sys/autoconf.h>
32 #include <sys/promif.h>
33 #include <sys/prom_plat.h>
34 #include <sys/promimpl.h>
35 #include <sys/platform_module.h>
36 #include <sys/clock.h>
37 #include <sys/pte.h>
38 #include <sys/scb.h>
39 #include <sys/cpu.h>
40 #include <sys/stack.h>
41 #include <sys/intreg.h>
42 #include <sys/ivintr.h>
43 #include <vm/as.h>
44 #include <vm/hat_sfmmu.h>
45 #include <sys/reboot.h>
46 #include <sys/sysmacros.h>
47 #include <sys/vtrace.h>
48 #include <sys/trap.h>
49 #include <sys/machtrap.h>
50 #include <sys/privregs.h>
51 #include <sys/machpcb.h>
52 #include <sys/proc.h>
53 #include <sys/cpupart.h>
54 #include <sys/pset.h>
55 #include <sys/cpu_module.h>
56 #include <sys/copyops.h>
57 #include <sys/panic.h>
58 #include <sys/bootconf.h> /* for bootops */
59 #include <sys/pg.h>
60 #include <sys/kdi.h>
61 #include <sys/fpras.h>
```

```
63 #include <sys/prom_debug.h>
64 #include <sys/debug.h>

66 #include <sys/sunddi.h>
67 #include <sys/lgrp.h>
68 #include <sys/traptrace.h>

70 #include <sys/kobj_impl.h>
71 #include <sys/kdi_machimpl.h>

73 /*
74  * External Routines:
75  */
76 extern void map_wellknown_devices(void);
77 extern void hsvc_setup(void);
78 extern void mach_descrip_startup_init(void);
79 extern void mach_soft_state_init(void);

81 int     dcache_size;
82 int     dcache_linesize;
83 int     icache_size;
84 int     icache_linesize;
85 int     ecache_size;
86 int     ecache_alignsize;
87 int     ecache_associativity;
88 int     ecache_setsize; /* max possible e$ setsize */
89 int     cpu_setsize; /* max e$ setsize of configured cpus */
90 int     dcache_line_mask; /* spitfire only */
91 int     vac_size; /* cache size in bytes */
92 uint_t  vac_mask; /* VAC alignment consistency mask */
93 int     vac_shift; /* log2(vac_size) for ppmapout() */
94 int     vac = 0; /* virtual address cache type (none == 0) */

96 /*
97  * fpras. An individual sun4* machine class (or perhaps subclass,
98  * eg sun4u/cheetah) must set fpras_implemented to indicate that it implements
99  * the fpRAS feature. The feature can be suppressed by setting fpras_disable
100 * or the mechanism can be disabled for individual copy operations with
101 * fpras_disableids. All these are checked in post_startup() code so
102 * fpras_disable and fpras_disableids can be set in /etc/system.
103 * If/when fpRAS is implemented on non-sun4 architectures these
104 * definitions will need to move up to the common level.
105 */
106 int     fpras_implemented;
107 int     fpras_disable;
108 int     fpras_disableids;

110 /*
111  * Static Routines:
112  */
113 static void kern_splr_preprom(void);
114 static void kern_splx_postprom(void);

116 /*
117  * Setup routine called right before main(). Interposing this function
118  * before main() allows us to call it in a machine-independent fashion.
119  */

121 void
122 mlsetup(struct regs *rp, kfpu_t *fp)
123 {
124     struct machpcb *mpcb;

126     extern char t0stack[];
127     extern struct classfuncs sys_classfuncs;
```

```

128     extern disp_t cpu0_disp;
129     unsigned long long pa;

131 #ifdef TRAPTRACE
132     TRAP_TRACE_CTL *ctlp;
133 #endif /* TRAPTRACE */

135     /* drop into kmdb on boot -d */
136     if (boothowto & RB_DEBUGENTER)
137         kmdb_enter();

139     /*
140      * initialize cpu_self
141      */
142     cpu0.cpu_self = &cpu0;

144     /*
145      * initialize t0
146      */
147     t0.t_stk = (caddr_t)rp - REGOFF;
148     /* Can't use va_to_pa here - wait until prom_ initialized */
149     t0.t_stkbase = t0stack;
150     t0.t_pri = maxclsyspri - 3;
151     t0.t_schedflag = 0;
152     t0.t_schedflag = TS_LOAD | TS_DONT_SWAP;
153     t0.t_procp = &p0;
154     t0.t_plockp = &p0lock.pl_lock;
155     t0.t_lwp = &lwp0;
156     t0.t_forw = &t0;
157     t0.t_back = &t0;
158     t0.t_next = &t0;
159     t0.t_prev = &t0;
160     t0.t_cpu = &cpu0;          /* loaded by _start */
161     t0.t_disp_queue = &cpu0_disp;
162     t0.t_bind_cpu = PBIND_NONE;
163     t0.t_bind_pset = PS_NONE;
164     t0.t_bindflag = (uchar_t)default_binding_mode;
165     t0.t_cpupart = &cp_default;
166     t0.t_clfuncs = &sys_classfuncs.thread;
167     t0.t_copyops = NULL;
168     THREAD_ONPROC(&t0, CPU);

169     lwp0.lwp_thread = &t0;
170     lwp0.lwp_procp = &p0;
171     lwp0.lwp_regs = (void *)rp;
172     t0.t_tid = p0.p_lwpcnt = p0.p_lwprcnt = p0.p_lwpid = 1;

174     mpcb = lwptompcb(&lwp0);
175     mpcb->mpcb_fpu = fp;
176     mpcb->mpcb_fpu->fpu_q = mpcb->mpcb_fpu_q;
177     mpcb->mpcb_thread = &t0;
178     lwp0.lwp_fpu = (void *)mpcb->mpcb_fpu;

180     p0.p_exec = NULL;
181     p0.p_stat = SRUN;
182     p0.p_flag = SSYS;
183     p0.p_tlist = &t0;
184     p0.p_stksize = 2*PAGESIZE;
185     p0.p_stkpageszc = 0;
186     p0.p_as = &kas;
187     p0.p_lockp = &p0lock;
188     p0.p_utrap = NULL;
189     p0.p_brkpageszc = 0;
190     p0.p_tl_lgrp_id = LGRP_NONE;
191     p0.p_tr_lgrp_id = LGRP_NONE;
192     sigorset(&p0.p_ignore, &ignoredefault);

```

```

194     CPU->cpu_thread = &t0;
195     CPU->cpu_dispthread = &t0;
196     bzero(&cpu0_disp, sizeof (disp_t));
197     CPU->cpu_disp = &cpu0_disp;
198     CPU->cpu_disp->disp_cpu = CPU;
199     CPU->cpu_idle_thread = &t0;
200     CPU->cpu_flags = CPU_RUNNING;
201     CPU->cpu_id = getprocessorid();
202     CPU->cpu_dispatch_pri = t0.t_pri;

204     /*
205      * Initialize thread/cpu microstate accounting
206      */
207     init_mstate(&t0, LMS_SYSTEM);
208     init_cpu_mstate(CPU, CMS_SYSTEM);

210     /*
211      * Initialize lists of available and active CPUs.
212      */
213     cpu_list_init(CPU);

215     cpu_vm_data_init(CPU);

217     pg_cpu_bootstrap(CPU);

219     (void) prom_set_preprom(kern_splr_preprom);
220     (void) prom_set_postprom(kern_splx_postprom);
221     PRM_INFO("mlsetup: now ok to call prom_printf");

223     mpcb->mpcb_pa = va_to_pa(t0.t_stk);

225     /*
226      * Claim the physical and virtual resources used by panicbuf,
227      * then map panicbuf. This operation removes the phys and
228      * virtual addresses from the free lists.
229      */
230     if (prom_claim_virt(PANICBUFSIZE, panicbuf) != panicbuf)
231         prom_panic("Can't claim panicbuf virtual address");

233     if (prom_retain("panicbuf", PANICBUFSIZE, MMU_PAGESIZE, &pa) != 0)
234         prom_panic("Can't allocate retained panicbuf physical address");

236     if (prom_map_phys(-1, PANICBUFSIZE, panicbuf, pa) != 0)
237         prom_panic("Can't map panicbuf");

239     PRM_DEBUG(panicbuf);
240     PRM_DEBUG(pa);

242     /*
243      * Negotiate hypervisor services, if any
244      */
245     hsvc_setup();
246     mach_soft_state_init();

248 #ifdef TRAPTRACE
249     /*
250      * initialize the trap trace buffer for the boot cpu
251      * XXX todo, dynamically allocate this buffer too
252      */
253     ctlp = &trap_trace_ctl[CPU->cpu_id];
254     ctlp->d.vaddr_base = trap_tr0;
255     ctlp->d.offset = ctlp->d.last_offset = 0;
256     ctlp->d.limit = TRAP_TSIZE;          /* XXX dynamic someday */
257     ctlp->d.paddr_base = va_to_pa(trap_tr0);
258 #endif /* TRAPTRACE */

```

```
260     /*
261     * Initialize the Machine Description kernel framework
262     */
264     mach_descrip_startup_init();
266     /*
267     * initialize HV trap trace buffer for the boot cpu
268     */
269     mach_htraptrace_setup(CPU->cpu_id);
270     mach_htraptrace_configure(CPU->cpu_id);
272     /*
273     * lgroup framework initialization. This must be done prior
274     * to devices being mapped.
275     */
276     lgrp_init(LGRP_INIT_STAGE1);
278     cpu_setup();
280     if (boothowto & RB_HALT) {
281         prom_printf("unix: kernel halted by -h flag\n");
282         prom_enter_mon();
283     }
285     setcputype();
286     map_wellknown_devices();
287     setcpudelay();
288 }
unchanged_portion_omitted
```

```

*****
51352 Thu Feb 18 08:40:45 2016
new/usr/src/uts/sun4/os/trap.c
6583 remove whole-process swapping
*****
_____unchanged_portion_omitted_____

121 #if defined(SF_ERRATA_23) || defined(SF_ERRATA_30) /* call ... illegal-insn */
122 int ill_calls;
123 #endif

125 /*
126 * Currently, the only PREFETCH/PREFETCHA instructions which cause traps
127 * are the "strong" prefetches (fcn=20-23). But we check for all flavors of
128 * PREFETCH, in case some future variant also causes a DATA_MMU_MISS.
129 */
130 #define IS_PREFETCH(i) (((i) & 0xc1780000) == 0xc1680000)

132 #define IS_FLUSH(i) (((i) & 0xc1f80000) == 0x81d80000)
133 #define IS_SWAP(i) (((i) & 0xc1f80000) == 0xc0780000)
134 #define IS_LDSTUB(i) (((i) & 0xc1f80000) == 0xc0680000)
135 #define IS_FLOAT(i) (((i) & 0x1000000) != 0)
136 #define IS_STORE(i) (((i) >> 21) & 1)

138 /*
139 * Called from the trap handler when a processor trap occurs.
140 */
141 /*VARARGS2*/
142 void
143 trap(struct regs *rp, caddr_t addr, uint32_t type, uint32_t mmu_fsr)
144 {
145     proc_t *p = ttproc(curthread);
146     klwp_id_t lwp = ttolwp(curthread);
147     struct machpcb *mpcb = NULL;
148     k_siginfo_t siginfo;
149     uint_t op3, fault = 0;
150     int stepped = 0;
151     greg_t oldpc;
152     int mstate;
153     char *badaddr;
154     faultcode_t res;
155     enum fault_type fault_type;
156     enum seg_rw rw;
157     uintptr_t lofault;
158     label_t *onfault;
159     int instr;
160     int iskernel;
161     int watchcode;
162     int watchpage;
163     extern faultcode_t pagefault(caddr_t, enum fault_type,
164     enum seg_rw, int);
165 #ifdef sun4v
166     extern boolean_t tick_stick_emulation_active;
167 #endif /* sun4v */

169     CPU_STATS_ADDQ(CPU, sys, trap, 1);

171 #ifndef SF_ERRATA_23 /* call causes illegal-insn */
172     ASSERT((curthread->t_schedflag & TS_DONT_SWAP) ||
173     (type == T_UNIMP_INSTR));
174 #else
175     ASSERT(curthread->t_schedflag & TS_DONT_SWAP);
176 #endif /* SF_ERRATA_23 */

171     if (USERMODE(rp->r_tstate) || (type & T_USER)) {
172         /*

```

```

173         * Set lwp_state before trying to acquire any
174         * adaptive lock
175         */
176     ASSERT(lwp != NULL);
177     lwp->lwp_state = LWP_SYS;
178     /*
179     * Set up the current cred to use during this trap. u_cred
180     * no longer exists. t_cred is used instead.
181     * The current process credential applies to the thread for
182     * the entire trap. If trapping from the kernel, this
183     * should already be set up.
184     */
185     if (curthread->t_cred != p->p_cred) {
186         cred_t *oldcred = curthread->t_cred;
187         /*
188         * DTrace accesses t_cred in probe context. t_cred
189         * must always be either NULL, or point to a valid,
190         * allocated cred structure.
191         */
192         curthread->t_cred = crgetcred();
193         crfree(oldcred);
194     }
195     type |= T_USER;
196     ASSERT((type == (T_SYS_RTT_PAGE | T_USER)) ||
197     (type == (T_SYS_RTT_ALIGN | T_USER)) ||
198     lwp->lwp_regs == rp);
199     mpcb = lwptombpcb(lwp);
200     switch (type) {
201     case T_WIN_OVERFLOW + T_USER:
202     case T_WIN_UNDERFLOW + T_USER:
203     case T_SYS_RTT_PAGE + T_USER:
204     case T_DATA_MMU_MISS + T_USER:
205         mstate = LMS_DFAULT;
206         break;
207     case T_INSTR_MMU_MISS + T_USER:
208         mstate = LMS_TFAULT;
209         break;
210     default:
211         mstate = LMS_TRAP;
212         break;
213     }
214     /* Kernel probe */
215     TNF_PROBE_1(thread_state, "thread", /* CSTYLEL */ ,
216     tnf_microstate, state, (char)mstate);
217     mstate = new_mstate(curthread, mstate);
218     siginfo.si_signo = 0;
219     stepped =
220     lwp->lwp_pcb.pcb_step != STEP_NONE &&
221     ((oldpc = rp->r_pc), prundostep()) &&
222     mmu_btop((uintptr_t)addr) == mmu_btop((uintptr_t)oldpc);
223     /* this assignment must not precede call to prundostep() */
224     oldpc = rp->r_pc;
225 }

227     TRACE_1(TR_FAC_TRAP, TR_C_TRAP_HANDLER_ENTER,
228     "C_trap_handler_enter:type %x", type);

230 #ifdef F_DEFERRED
231     /*
232     * Take any pending floating point exceptions now.
233     * If the floating point unit has an exception to handle,
234     * just return to user-level to let the signal handler run.
235     * The instruction that got us to trap() will be reexecuted on
236     * return from the signal handler and we will trap to here again.
237     * This is necessary to disambiguate simultaneous traps which
238     * happen when a floating-point exception is pending and a

```

```

239     * machine fault is incurred.
240     */
241     if (type & USER) {
242         /*
243          * FP_TRAPPED is set only by sendsig() when it copies
244          * out the floating-point queue for the signal handler.
245          * It is set there so we can test it here and in syscall().
246          */
247         mpcb->mpcb_flags &= ~FP_TRAPPED;
248         syncfpu();
249         if (mpcb->mpcb_flags & FP_TRAPPED) {
250             /*
251              * trap() has have been called recursively and may
252              * have stopped the process, so do single step
253              * support for /proc.
254              */
255             mpcb->mpcb_flags &= ~FP_TRAPPED;
256             goto out;
257         }
258     }
259 #endif
260     switch (type) {
261     case T_DATA_MMU_MISS:
262     case T_INSTR_MMU_MISS + T_USER:
263     case T_DATA_MMU_MISS + T_USER:
264     case T_DATA_PROT + T_USER:
265     case T_AST + T_USER:
266     case T_SYS_RTT_PAGE + T_USER:
267     case T_FLUSH_PCB + T_USER:
268     case T_FLUSHW + T_USER:
269         break;
270
271     default:
272         FTRACE_3("trap(): type=0x%lx, regs=0x%lx, addr=0x%lx",
273             (ulong_t)type, (ulong_t)rp, (ulong_t)addr);
274         break;
275     }
276
277     switch (type) {
278
279     default:
280         /*
281          * Check for user software trap.
282          */
283         if (type & T_USER) {
284             if (tudebug)
285                 showregs(type, rp, (caddr_t)0, 0);
286             if ((type & ~T_USER) >= T_SOFTWARE_TRAP) {
287                 bzero(&siginfo, sizeof (siginfo));
288                 siginfo.si_signo = SIGILL;
289                 siginfo.si_code = ILL_ILTRP;
290                 siginfo.si_addr = (caddr_t)rp->r_pc;
291                 siginfo.si_trapno = type &~ T_USER;
292                 fault = FLTILL;
293                 break;
294             }
295         }
296         addr = (caddr_t)rp->r_pc;
297         (void) die(type, rp, addr, 0);
298         /*NOTREACHED*/
299
300     case T_ALIGNMENT: /* supv alignment error */
301         if (nload(rp, NULL))
302             goto cleanup;
303
304         if (curthread->t_lofault) {

```

```

305         if (lodebug) {
306             showregs(type, rp, addr, 0);
307             traceback((caddr_t)rp->r_sp);
308         }
309         rp->r_g1 = EFAULT;
310         rp->r_pc = curthread->t_lofault;
311         rp->r_npc = rp->r_pc + 4;
312         goto cleanup;
313     }
314     (void) die(type, rp, addr, 0);
315     /*NOTREACHED*/
316
317     case T_INSTR_EXCEPTION: /* sys instruction access exception */
318         addr = (caddr_t)rp->r_pc;
319         (void) die(type, rp, addr, mmu_fsr);
320         /*NOTREACHED*/
321
322     case T_INSTR_MMU_MISS: /* sys instruction mmu miss */
323         addr = (caddr_t)rp->r_pc;
324         (void) die(type, rp, addr, 0);
325         /*NOTREACHED*/
326
327     case T_DATA_EXCEPTION: /* system data access exception */
328         switch (X_FAULT_TYPE(mmu_fsr)) {
329         case FT_RANGE:
330             /*
331              * This happens when we attempt to dereference an
332              * address in the address hole. If t_ontrap is set,
333              * then break and fall through to T_DATA_MMU_MISS /
334              * T_DATA_PROT case below. If lofault is set, then
335              * honour it (perhaps the user gave us a bogus
336              * address in the hole to copyin from or copyout to?)
337              */
338
339             if (curthread->t_ontrap != NULL)
340                 break;
341
342             addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
343             if (curthread->t_lofault) {
344                 if (lodebug) {
345                     showregs(type, rp, addr, 0);
346                     traceback((caddr_t)rp->r_sp);
347                 }
348                 rp->r_g1 = EFAULT;
349                 rp->r_pc = curthread->t_lofault;
350                 rp->r_npc = rp->r_pc + 4;
351                 goto cleanup;
352             }
353             (void) die(type, rp, addr, mmu_fsr);
354             /*NOTREACHED*/
355
356         case FT_PRIV:
357             /*
358              * This can happen if we access ASI_USER from a kernel
359              * thread. To support pxfs, we need to honor lofault if
360              * we're doing a copyin/copyout from a kernel thread.
361              */
362
363             if (nload(rp, NULL))
364                 goto cleanup;
365             addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
366             if (curthread->t_lofault) {
367                 if (lodebug) {
368                     showregs(type, rp, addr, 0);
369                     traceback((caddr_t)rp->r_sp);
370                 }

```

```

371         rp->r_g1 = EFAULT;
372         rp->r_pc = curthread->t_lofault;
373         rp->r_npc = rp->r_pc + 4;
374         goto cleanup;
375     }
376     (void) die(type, rp, addr, mmu_fsr);
377     /*NOTREACHED*/

379     default:
380         if (nload(rp, NULL))
381             goto cleanup;
382         addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
383         (void) die(type, rp, addr, mmu_fsr);
384         /*NOTREACHED*/

386     case FT_NFO:
387         break;
388     }
389     /* fall into ... */

391     case T_DATA_MMU_MISS:           /* system data mmu miss */
392     case T_DATA_PROT:              /* system data protection fault */
393         if (nload(rp, &instr))
394             goto cleanup;

396     /*
397     * If we're under on_trap() protection (see <sys/ontrap.h>),
398     * set ot_trap and return from the trap to the trampoline.
399     */
400     if (curthread->t_ontrap != NULL) {
401         on_trap_data_t *otp = curthread->t_ontrap;

403         TRACE_0(TR_FAC_TRAP, TR_C_TRAP_HANDLER_EXIT,
404                "C_trap_handler_exit");
405         TRACE_0(TR_FAC_TRAP, TR_TRAP_END, "trap_end");

407         if (otp->ot_prot & OT_DATA_ACCESS) {
408             otp->ot_trap |= OT_DATA_ACCESS;
409             rp->r_pc = otp->ot_trampoline;
410             rp->r_npc = rp->r_pc + 4;
411             goto cleanup;
412         }
413     }
414     lofault = curthread->t_lofault;
415     onfault = curthread->t_onfault;
416     curthread->t_lofault = 0;

418     mstate = new_mstate(curthread, LMS_KFAULT);

420     switch (type) {
421     case T_DATA_PROT:
422         fault_type = F_PROT;
423         rw = S_WRITE;
424         break;
425     case T_INSTR_MMU_MISS:
426         fault_type = F_INVALID;
427         rw = S_EXEC;
428         break;
429     case T_DATA_MMU_MISS:
430     case T_DATA_EXCEPTION:
431         /*
432         * The hardware doesn't update the fsr on mmu
433         * misses so it is not easy to find out whether
434         * the access was a read or a write so we need
435         * to decode the actual instruction.
436         */

```

```

437         fault_type = F_INVALID;
438         rw = get_accesstype(rp);
439         break;
440     default:
441         cmn_err(CE_PANIC, "trap: unknown type %x", type);
442         break;
443     }
444     /*
445     * We determine if access was done to kernel or user
446     * address space. The addr passed into trap is really the
447     * tag access register.
448     */
449     iskernl = (((uintptr_t)addr & TAGACC_CTX_MASK) == KCONTEXT);
450     addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);

452     res = pagefault(addr, fault_type, rw, iskernl);
453     if (!iskernl && res == FC_NOMAP &&
454         addr < p->p_usrstack && grow(addr))
455         res = 0;

457     (void) new_mstate(curthread, mstate);

459     /*
460     * Restore lofault and onfault. If we resolved the fault, exit.
461     * If we didn't and lofault wasn't set, die.
462     */
463     curthread->t_lofault = lofault;
464     curthread->t_onfault = onfault;

466     if (res == 0)
467         goto cleanup;

469     if (IS_PREFETCH(instr)) {
470         /* skip prefetch instructions in kernel-land */
471         rp->r_pc = rp->r_npc;
472         rp->r_npc += 4;
473         goto cleanup;
474     }

476     if ((lofault == 0 || lodebug) &&
477         (calc_memaddr(rp, &badaddr) == SIMU_SUCCESS))
478         addr = badaddr;
479     if (lofault == 0)
480         (void) die(type, rp, addr, 0);
481     /*
482     * Cannot resolve fault. Return to lofault.
483     */
484     if (lodebug) {
485         showregs(type, rp, addr, 0);
486         traceback((caddr_t)rp->r_sp);
487     }
488     if (FC_CODE(res) == FC_OBJERR)
489         res = FC_ERRNO(res);
490     else
491         res = EFAULT;
492     rp->r_g1 = res;
493     rp->r_pc = curthread->t_lofault;
494     rp->r_npc = curthread->t_lofault + 4;
495     goto cleanup;

497     case T_INSTR_EXCEPTION + T_USER: /* user insn access exception */
498         bzero(&siginfo, sizeof (siginfo));
499         siginfo.si_addr = (caddr_t)rp->r_pc;
500         siginfo.si_signo = SIGSEGV;
501         siginfo.si_code = X_FAULT_TYPE(mmu_fsr) == FT_PRIV ?
502             SEGV_ACCERR : SEGV_MAPERR;

```

```

503         fault = FLTBOUNDS;
504         break;

506     case T_WIN_OVERFLOW + T_USER: /* window overflow in ??? */
507     case T_WIN_UNDERFLOW + T_USER: /* window underflow in ??? */
508     case T_SYS_RTT_PAGE + T_USER: /* window underflow in user_rtt */
509     case T_INSTR_MMU_MISS + T_USER: /* user instruction mmu miss */
510     case T_DATA_MMU_MISS + T_USER: /* user data mmu miss */
511     case T_DATA_PROT + T_USER: /* user data protection fault */
512         switch (type) {
513             case T_INSTR_MMU_MISS + T_USER:
514                 addr = (caddr_t)rp->r_pc;
515                 fault_type = F_INVAL;
516                 rw = S_EXEC;
517                 break;

519             case T_DATA_MMU_MISS + T_USER:
520                 addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
521                 fault_type = F_INVAL;
522                 /*
523                  * The hardware doesn't update the sfsr on mmu misses
524                  * so it is not easy to find out whether the access
525                  * was a read or a write so we need to decode the
526                  * actual instruction. XXX BUGLY HW
527                  */
528                 rw = get_accesstype(rp);
529                 break;

531             case T_DATA_PROT + T_USER:
532                 addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
533                 fault_type = F_PROT;
534                 rw = S_WRITE;
535                 break;

537             case T_WIN_OVERFLOW + T_USER:
538                 addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
539                 fault_type = F_INVAL;
540                 rw = S_WRITE;
541                 break;

543             case T_WIN_UNDERFLOW + T_USER:
544             case T_SYS_RTT_PAGE + T_USER:
545                 addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
546                 fault_type = F_INVAL;
547                 rw = S_READ;
548                 break;

550         default:
551             cmn_err(CE_PANIC, "trap: unknown type %x", type);
552             break;
553     }

555     /*
556     * If we are single stepping do not call pagefault
557     */
558     if (stepped) {
559         res = FC_NOMAP;
560     } else {
561         caddr_t vaddr = addr;
562         size_t sz;
563         int ta;

565         ASSERT(!(curthread->t_flag & T_WATCHPT));
566         watchpage = (pr_watch_active(p) &&
567             type != T_WIN_OVERFLOW + T_USER &&
568             type != T_WIN_UNDERFLOW + T_USER &&

```

```

569         type != T_SYS_RTT_PAGE + T_USER &&
570         pr_is_watchpage(addr, rw));

572     if (!watchpage ||
573         (sz = instr_size(rp, &vaddr, rw)) <= 0)
574         /* EMPTY */;
575     else if ((watchcode = pr_is_watchpoint(&vaddr, &ta,
576         sz, NULL, rw)) != 0) {
577         if (ta) {
578             do_watch_step(vaddr, sz, rw,
579                 watchcode, rp->r_pc);
580             fault_type = F_INVAL;
581         } else {
582             bzero(&siginfo, sizeof (siginfo));
583             siginfo.si_signo = SIGTRAP;
584             siginfo.si_code = watchcode;
585             siginfo.si_addr = vaddr;
586             siginfo.si_trapafter = 0;
587             siginfo.si_pc = (caddr_t)rp->r_pc;
588             fault = FLTWATCH;
589             break;
590         }
591     } else {
592         if (rw != S_EXEC &&
593             pr_watch_emul(rp, vaddr, rw))
594             goto out;
595         do_watch_step(vaddr, sz, rw, 0, 0);
596         fault_type = F_INVAL;
597     }

599     if (pr_watch_active(p) &&
600         (type == T_WIN_OVERFLOW + T_USER ||
601          type == T_WIN_UNDERFLOW + T_USER ||
602          type == T_SYS_RTT_PAGE + T_USER)) {
603         int dotwo = (type == T_WIN_UNDERFLOW + T_USER);
604         if (copy_return_window(dotwo))
605             goto out;
606         fault_type = F_INVAL;
607     }

609     res = pagefault(addr, fault_type, rw, 0);

611     /*
612     * If pagefault succeed, ok.
613     * Otherwise grow the stack automatically.
614     */
615     if (res == 0 ||
616         (res == FC_NOMAP &&
617          type != T_INSTR_MMU_MISS + T_USER &&
618          addr < p->p_usrstack &&
619          grow(addr))) {
620         int ismem = prismember(&p->p_fltmask, FLTPAGE);

622         /*
623         * instr_size() is used to get the exact
624         * address of the fault, instead of the
625         * page of the fault. Unfortunately it is
626         * very slow, and this is an important
627         * code path. Don't call it unless
628         * correctness is needed. ie. if FLTPAGE
629         * is set, or we're profiling.
630         */

632         if (curthread->t_rprof != NULL || ismem)
633             (void) instr_size(rp, &addr, rw);

```



```

635     lwp->lwp_lastfault = FLTPAGE;
636     lwp->lwp_lastfaddr = addr;

638     if (ismem) {
639         bzero(&siginfo, sizeof (siginfo));
640         siginfo.si_addr = addr;
641         (void) stop_on_fault(FLTPAGE, &siginfo);
642     }
643     goto out;
644 }

646     if (type != (T_INSTR_MMU_MISS + T_USER)) {
647         /*
648          * check for non-faulting loads, also
649          * fetch the instruction to check for
650          * flush
651          */
652         if (nflod(rp, &instr))
653             goto out;

655         /* skip userland prefetch instructions */
656         if (IS_PREFETCH(instr)) {
657             rp->r_pc = rp->r_npc;
658             rp->r_npc += 4;
659             goto out;
660             /*NOTREACHED*/
661         }

663         /*
664          * check if the instruction was a
665          * flush.  ABI allows users to specify
666          * an illegal address on the flush
667          * instruction so we simply return in
668          * this case.
669          *
670          * NB: the hardware should set a bit
671          * indicating this trap was caused by
672          * a flush instruction.  Instruction
673          * decoding is buggy!
674          */
675         if (IS_FLUSH(instr)) {
676             /* skip the flush instruction */
677             rp->r_pc = rp->r_npc;
678             rp->r_npc += 4;
679             goto out;
680             /*NOTREACHED*/
681         }
682     } else if (res == FC_PROT) {
683         report_stack_exec(p, addr);
684     }

686     if (tudebug)
687         showregs(type, rp, addr, 0);
688 }

690 /*
691  * In the case where both pagefault and grow fail,
692  * set the code to the value provided by pagefault.
693  */
694 (void) instr_size(rp, &addr, rw);
695 bzero(&siginfo, sizeof (siginfo));
696 siginfo.si_addr = addr;
697 if (FC_CODE(res) == FC_OBJERR) {
698     siginfo.si_errno = FC_ERRNO(res);
699     if (siginfo.si_errno != EINTR) {
700         siginfo.si_signo = SIGBUS;

```

```

701         siginfo.si_code = BUS_OBJERR;
702         fault = FLTACCESS;
703     }
704 } else { /* FC_NOMAP || FC_PROT */
705     siginfo.si_signo = SIGSEGV;
706     siginfo.si_code = (res == FC_NOMAP) ?
707         SEGV_MAPERR : SEGV_ACCERR;
708     fault = FLTBOUNDS;
709 }
710 /*
711  * If this is the culmination of a single-step,
712  * reset the addr, code, signal and fault to
713  * indicate a hardware trace trap.
714  */
715 if (stepped) {
716     pcb_t *pcb = &lwp->lwp_pcb;

718     siginfo.si_signo = 0;
719     fault = 0;
720     if (pcb->pcb_step == STEP_WASACTIVE) {
721         pcb->pcb_step = STEP_NONE;
722         pcb->pcb_tracepc = NULL;
723         oldpc = rp->r_pc - 4;
724     }
725     /*
726      * If both NORMAL_STEP and WATCH_STEP are in
727      * effect, give precedence to WATCH_STEP.
728      * One or the other must be set at this point.
729      */
730     ASSERT(pcb->pcb_flags & (NORMAL_STEP|WATCH_STEP));
731     if ((fault = undo_watch_step(&siginfo)) == 0 &&
732         (pcb->pcb_flags & NORMAL_STEP)) {
733         siginfo.si_signo = SIGTRAP;
734         siginfo.si_code = TRAP_TRACE;
735         siginfo.si_addr = (caddr_t)rp->r_pc;
736         fault = FLITRACE;
737     }
738     pcb->pcb_flags &= ~(NORMAL_STEP|WATCH_STEP);
739 }
740 break;

742 case T_DATA_EXCEPTION + T_USER: /* user data access exception */

744     if (&visl_partial_support != NULL) {
745         bzero(&siginfo, sizeof (siginfo));
746         if (visl_partial_support(rp,
747             &siginfo, &fault) == 0)
748             goto out;
749     }

751     if (nflod(rp, &instr))
752         goto out;
753     if (IS_FLUSH(instr)) {
754         /* skip the flush instruction */
755         rp->r_pc = rp->r_npc;
756         rp->r_npc += 4;
757         goto out;
758         /*NOTREACHED*/
759     }
760     bzero(&siginfo, sizeof (siginfo));
761     siginfo.si_addr = addr;
762     switch (X_FAULT_TYPE(mmu_fsr)) {
763     case FT_ATOMIC_NC:
764         if ((IS_SWAP(instr) && swap_nc(rp, instr)) ||
765             (IS_LDSTUB(instr) && ldstub_nc(rp, instr))) {
766             /* skip the atomic */

```

```

767         rp->r_pc = rp->r_npc;
768         rp->r_npc += 4;
769         goto out;
770     }
771     /* fall into ... */
772 case FT_PRIV:
773     siginfo.si_signo = SIGSEGV;
774     siginfo.si_code = SEGV_ACCERR;
775     fault = FLTBOUNDS;
776     break;
777 case FT_SPEC_LD:
778 case FT_ILL_ALT:
779     siginfo.si_signo = SIGILL;
780     siginfo.si_code = ILL_ILLADR;
781     fault = FLTILL;
782     break;
783 default:
784     siginfo.si_signo = SIGSEGV;
785     siginfo.si_code = SEGV_MAPERR;
786     fault = FLTBOUNDS;
787     break;
788 }
789 break;

791 case T_SYS_RTT_ALIGN + T_USER: /* user alignment error */
792 case T_ALIGNMENT + T_USER:   /* user alignment error */
793     if (tudebug)
794         showregs(type, rp, addr, 0);
795     /*
796      * If the user has to do unaligned references
797      * the ugly stuff gets done here.
798      */
799     alignfaults++;
800     if (&visl_partial_support != NULL) {
801         bzero(&siginfo, sizeof (siginfo));
802         if (visl_partial_support(rp,
803             &siginfo, &fault) == 0)
804             goto out;
805     }

807     bzero(&siginfo, sizeof (siginfo));
808     if (type == T_SYS_RTT_ALIGN + T_USER) {
809         if (nload(rp, NULL))
810             goto out;
811         /*
812          * Can't do unaligned stack access
813          */
814         siginfo.si_signo = SIGBUS;
815         siginfo.si_code = BUS_ADRALN;
816         siginfo.si_addr = addr;
817         fault = FLTACCESS;
818         break;
819     }

821     /*
822     * Try to fix alignment before non-faulting load test.
823     */
824     if (p->p_fixalignment) {
825         if (do_unaligned(rp, &badaddr) == SIMU_SUCCESS) {
826             rp->r_pc = rp->r_npc;
827             rp->r_npc += 4;
828             goto out;
829         }
830         if (nload(rp, NULL))
831             goto out;
832         siginfo.si_signo = SIGSEGV;

```

```

833         siginfo.si_code = SEGV_MAPERR;
834         siginfo.si_addr = badaddr;
835         fault = FLTBOUNDS;
836     } else {
837         if (nload(rp, NULL))
838             goto out;
839         siginfo.si_signo = SIGBUS;
840         siginfo.si_code = BUS_ADRALN;
841         if (rp->r_pc & 3) { /* offending address, if pc */
842             siginfo.si_addr = (caddr_t)rp->r_pc;
843         } else {
844             if (calc_memaddr(rp, &badaddr) == SIMU_UNALIGN)
845                 siginfo.si_addr = badaddr;
846             else
847                 siginfo.si_addr = (caddr_t)rp->r_pc;
848         }
849         fault = FLTACCESS;
850     }
851     break;

853 case T_PRIV_INSTR + T_USER: /* privileged instruction fault */
854     if (tudebug)
855         showregs(type, rp, (caddr_t)0, 0);

857     bzero(&siginfo, sizeof (siginfo));
858 #ifdef sun4v
859     /*
860      * If this instruction fault is a non-privileged %tick
861      * or %stick trap, and %tick/%stick user emulation is
862      * enabled as a result of an OS suspend, then simulate
863      * the register read. We rely on simulate_rdtick to fail
864      * if the instruction is not a %tick or %stick read,
865      * causing us to fall through to the normal privileged
866      * instruction handling.
867      */
868     if (tick_stick_emulation_active &&
869         (X_FAULT_TYPE(mmu_fsr) == FT_NEW_PRIVACT) &&
870         simulate_rdtick(rp) == SIMU_SUCCESS) {
871         /* skip the successfully simulated instruction */
872         rp->r_pc = rp->r_npc;
873         rp->r_npc += 4;
874         goto out;
875     }
876 #endif

877     siginfo.si_signo = SIGILL;
878     siginfo.si_code = ILL_PRIVOPC;
879     siginfo.si_addr = (caddr_t)rp->r_pc;
880     fault = FLTILL;
881     break;

883 case T_UNIMP_INSTR: /* priv illegal instruction fault */
884     if (fpras_implemented) {
885         /*
886          * Call fpras_chktrap indicating that
887          * we've come from a trap handler and pass
888          * the regs. That function may choose to panic
889          * (in which case it won't return) or it may
890          * determine that a reboot is desired. In the
891          * latter case it must alter pc/npc to skip
892          * the illegal instruction and continue at
893          * a controlled address.
894          */
895         if (&fpras_chktrap) {
896             if (fpras_chktrap(rp))
897                 goto cleanup;
898         }

```

```

899     }
900 #if defined(SF_ERRATA_23) || defined(SF_ERRATA_30) /* call ... illegal-insn */
901     instr = *(int *)rp->r_pc;
902     if ((instr & 0xc0000000) == 0x40000000) {
903         long pc;
904
905         rp->r_o7 = (long long)rp->r_pc;
906         pc = rp->r_pc + ((instr & 0x3fffffff) << 2);
907         rp->r_pc = rp->r_npc;
908         rp->r_npc = pc;
909         ill_calls++;
910         goto cleanup;
911     }
912 #endif /* SF_ERRATA_23 || SF_ERRATA_30 */
913     /*
914     * It's not an fpras failure and it's not SF_ERRATA_23 - die
915     */
916     addr = (caddr_t)rp->r_pc;
917     (void) die(type, rp, addr, 0);
918     /*NOTREACHED*/
919
920     case T_UNIMP_INSTR + T_USER: /* illegal instruction fault */
921 #if defined(SF_ERRATA_23) || defined(SF_ERRATA_30) /* call ... illegal-insn */
922     instr = fetch_user_instr((caddr_t)rp->r_pc);
923     if ((instr & 0xc0000000) == 0x40000000) {
924         long pc;
925
926         rp->r_o7 = (long long)rp->r_pc;
927         pc = rp->r_pc + ((instr & 0x3fffffff) << 2);
928         rp->r_pc = rp->r_npc;
929         rp->r_npc = pc;
930         ill_calls++;
931         goto out;
932     }
933 #endif /* SF_ERRATA_23 || SF_ERRATA_30 */
934     if (tudebug)
935         showregs(type, rp, (caddr_t)0, 0);
936     bzero(&siginfo, sizeof (siginfo));
937     /*
938     * Try to simulate the instruction.
939     */
940     switch (simulate_unimp(rp, &badaddr)) {
941     case SIMU_RETRY:
942         goto out; /* regs are already set up */
943         /*NOTREACHED*/
944
945     case SIMU_SUCCESS:
946         /* skip the successfully simulated instruction */
947         rp->r_pc = rp->r_npc;
948         rp->r_npc += 4;
949         goto out;
950         /*NOTREACHED*/
951
952     case SIMU_FAULT:
953         siginfo.si_signo = SIGSEGV;
954         siginfo.si_code = SEGV_MAPERR;
955         siginfo.si_addr = badaddr;
956         fault = FLTBOUNDS;
957         break;
958
959     case SIMU_DZERO:
960         siginfo.si_signo = SIGFPE;
961         siginfo.si_code = FPE_INTDIV;
962         siginfo.si_addr = (caddr_t)rp->r_pc;
963         fault = FLTIZDIV;
964         break;

```

```

966     case SIMU_UNALIGN:
967         siginfo.si_signo = SIGBUS;
968         siginfo.si_code = BUS_ADRALN;
969         siginfo.si_addr = badaddr;
970         fault = FLTACCESS;
971         break;
972
973     case SIMU_ILLEGAL:
974     default:
975         siginfo.si_signo = SIGILL;
976         op3 = (instr >> 19) & 0x3F;
977         if ((IS_FLOAT(instr) && (op3 == IOP_V8_STQFA) ||
978             (op3 == IOP_V8_STDFA)))
979             siginfo.si_code = ILL_ILLADR;
980         else
981             siginfo.si_code = ILL_ILLOPC;
982         siginfo.si_addr = (caddr_t)rp->r_pc;
983         fault = FLTILL;
984         break;
985     }
986     break;
987
988     case T_UNIMP_LDD + T_USER:
989     case T_UNIMP_STD + T_USER:
990         if (tudebug)
991             showregs(type, rp, (caddr_t)0, 0);
992         switch (simulate_lddstd(rp, &badaddr)) {
993         case SIMU_SUCCESS:
994             /* skip the successfully simulated instruction */
995             rp->r_pc = rp->r_npc;
996             rp->r_npc += 4;
997             goto out;
998             /*NOTREACHED*/
999
1000         case SIMU_FAULT:
1001             if (nload(rp, NULL))
1002                 goto out;
1003             siginfo.si_signo = SIGSEGV;
1004             siginfo.si_code = SEGV_MAPERR;
1005             siginfo.si_addr = badaddr;
1006             fault = FLTBOUNDS;
1007             break;
1008
1009         case SIMU_UNALIGN:
1010             if (nload(rp, NULL))
1011                 goto out;
1012             siginfo.si_signo = SIGBUS;
1013             siginfo.si_code = BUS_ADRALN;
1014             siginfo.si_addr = badaddr;
1015             fault = FLTACCESS;
1016             break;
1017
1018         case SIMU_ILLEGAL:
1019         default:
1020             siginfo.si_signo = SIGILL;
1021             siginfo.si_code = ILL_ILLOPC;
1022             siginfo.si_addr = (caddr_t)rp->r_pc;
1023             fault = FLTILL;
1024             break;
1025         }
1026         break;
1027
1028     case T_UNIMP_LDD:
1029     case T_UNIMP_STD:
1030         if (simulate_lddstd(rp, &badaddr) == SIMU_SUCCESS) {

```

```

1031         /* skip the successfully simulated instruction */
1032         rp->r_pc = rp->r_npc;
1033         rp->r_npc += 4;
1034         goto cleanup;
1035         /*NOTREACHED*/
1036     }
1037     /*
1038     * A third party driver executed an {LDD,STD,LDDA,STDA}
1039     * that we couldn't simulate.
1040     */
1041     if (nflod(rp, NULL))
1042         goto cleanup;
1043
1044     if (curthread->t_lofault) {
1045         if (lodebug) {
1046             showregs(type, rp, addr, 0);
1047             traceback((caddr_t)rp->r_sp);
1048         }
1049         rp->r_g1 = EFAULT;
1050         rp->r_pc = curthread->t_lofault;
1051         rp->r_npc = rp->r_pc + 4;
1052         goto cleanup;
1053     }
1054     (void) die(type, rp, addr, 0);
1055     /*NOTREACHED*/
1056
1057     case T_IDIV0 + T_USER:          /* integer divide by zero */
1058     case T_DIV0 + T_USER:          /* integer divide by zero */
1059         if (tudebug && tudebugfpe)
1060             showregs(type, rp, (caddr_t)0, 0);
1061         bzero(&siginfo, sizeof (siginfo));
1062         siginfo.si_signo = SIGFPE;
1063         siginfo.si_code = FPE_INTDIV;
1064         siginfo.si_addr = (caddr_t)rp->r_pc;
1065         fault = FLTIZDIV;
1066         break;
1067
1068     case T_INT_OVERFLOW + T_USER:  /* integer overflow */
1069     if (tudebug && tudebugfpe)
1070         showregs(type, rp, (caddr_t)0, 0);
1071     bzero(&siginfo, sizeof (siginfo));
1072     siginfo.si_signo = SIGFPE;
1073     siginfo.si_code = FPE_INTOVF;
1074     siginfo.si_addr = (caddr_t)rp->r_pc;
1075     fault = FLTIOVF;
1076     break;
1077
1078     case T_BREAKPOINT + T_USER:    /* breakpoint trap (t 1) */
1079     if (tudebug && tudebugbpt)
1080         showregs(type, rp, (caddr_t)0, 0);
1081     bzero(&siginfo, sizeof (siginfo));
1082     siginfo.si_signo = SIGTRAP;
1083     siginfo.si_code = TRAP_BRKPT;
1084     siginfo.si_addr = (caddr_t)rp->r_pc;
1085     fault = FLTBPT;
1086     break;
1087
1088     case T_TAG_OVERFLOW + T_USER:  /* tag overflow (taddcctv, tsubcctv) */
1089     if (tudebug)
1090         showregs(type, rp, (caddr_t)0, 0);
1091     bzero(&siginfo, sizeof (siginfo));
1092     siginfo.si_signo = SIGEMT;
1093     siginfo.si_code = EMT_TAGOVF;
1094     siginfo.si_addr = (caddr_t)rp->r_pc;
1095     fault = FLTACCESS;
1096     break;

```

```

1098     case T_FLUSH_PCB + T_USER:    /* finish user window overflow */
1099     case T_FLUSHW + T_USER:       /* finish user window flush */
1100         /*
1101         * This trap is entered from sys_rtt in locore.s when,
1102         * upon return to user is is found that there are user
1103         * windows in pcb_wbuf. This happens because they could
1104         * not be saved on the user stack, either because it
1105         * wasn't resident or because it was misaligned.
1106         */
1107     {
1108         int error;
1109         caddr_t sp;
1110
1111         error = flush_user_windows_to_stack(&sp);
1112         /*
1113         * Possible errors:
1114         *   error copying out
1115         *   unaligned stack pointer
1116         * The first is given to us as the return value
1117         * from flush_user_windows_to_stack(). The second
1118         * results in residual windows in the pcb.
1119         */
1120         if (error != 0) {
1121             /*
1122             * EINTR comes from a signal during copyout;
1123             * we should not post another signal.
1124             */
1125             if (error != EINTR) {
1126                 /*
1127                 * Zap the process with a SIGSEGV - process
1128                 * may be managing its own stack growth by
1129                 * taking SIGSEGVs on a different signal stack.
1130                 */
1131                 bzero(&siginfo, sizeof (siginfo));
1132                 siginfo.si_signo = SIGSEGV;
1133                 siginfo.si_code = SEGV_MAPERR;
1134                 siginfo.si_addr = sp;
1135                 fault = FLTBOUNDS;
1136             }
1137             break;
1138         } else if (mpcb->mpcb_wbcnt) {
1139             bzero(&siginfo, sizeof (siginfo));
1140             siginfo.si_signo = SIGILL;
1141             siginfo.si_code = ILL_BADSTK;
1142             siginfo.si_addr = (caddr_t)rp->r_pc;
1143             fault = FLTILL;
1144             break;
1145         }
1146     }
1147
1148     /*
1149     * T_FLUSHW is used when handling a ta 0x3 -- the old flush
1150     * window trap -- which is implemented by executing the
1151     * flushw instruction. The flushw can trap if any of the
1152     * stack pages are not writable for whatever reason. In this
1153     * case only, we advance the pc to the next instruction so
1154     * that the user thread doesn't needlessly execute the trap
1155     * again. Normally this wouldn't be a problem -- we'll
1156     * usually only end up here if this is the first touch to a
1157     * stack page -- since the second execution won't trap, but
1158     * if there's a watchpoint on the stack page the user thread
1159     * would spin, continuously executing the trap instruction.
1160     */
1161     if (type == T_FLUSHW + T_USER) {
1162         rp->r_pc = rp->r_npc;

```

```

1163         rp->r_npc += 4;
1164     }
1165     goto out;

1167 case T_AST + T_USER:      /* profiling or resched pseudo trap */
1168     if (lwp->lwp_pcb.pcb_flags & CPC_OVERFLOW) {
1169         lwp->lwp_pcb.pcb_flags &= ~CPC_OVERFLOW;
1170         if (kcpic_overflow_ast()) {
1171             /*
1172              * Signal performance counter overflow
1173              */
1174             if (tudebug)
1175                 showregs(type, rp, (caddr_t)0, 0);
1176             bzero(&siginfo, sizeof (siginfo));
1177             siginfo.si_signo = SIGEMT;
1178             siginfo.si_code = EMT_CPCOVF;
1179             siginfo.si_addr = (caddr_t)rp->r_pc;
1180             /* for trap_cleanup(), below */
1181             oldpc = rp->r_pc - 4;
1182             fault = FLT_CPCOVF;
1183         }
1184     }

1186     /*
1187     * The CPC_OVERFLOW check above may already have populated
1188     * siginfo and set fault, so the checks below must not
1189     * touch these and the functions they call must use
1190     * trapsig() directly.
1191     */

1193     if (lwp->lwp_pcb.pcb_flags & ASYNC_HWERR) {
1194         lwp->lwp_pcb.pcb_flags &= ~ASYNC_HWERR;
1195         trap_async_hwerr();
1196     }

1198     if (lwp->lwp_pcb.pcb_flags & ASYNC_BERR) {
1199         lwp->lwp_pcb.pcb_flags &= ~ASYNC_BERR;
1200         trap_async_berr_bto(ASYNC_BERR, rp);
1201     }

1203     if (lwp->lwp_pcb.pcb_flags & ASYNC_BTO) {
1204         lwp->lwp_pcb.pcb_flags &= ~ASYNC_BTO;
1205         trap_async_berr_bto(ASYNC_BTO, rp);
1206     }

1208     break;
1209 }

1211 if (fault) {
1212     /* We took a fault so abort single step. */
1213     lwp->lwp_pcb.pcb_flags &= ~(NORMAL_STEP|WATCH_STEP);
1214 }
1215 trap_cleanup(rp, fault, &siginfo, oldpc == rp->r_pc);

1217 out:    /* We can't get here from a system trap */
1218 ASSERT(type & T_USER);
1219 trap_rtt();
1220 (void) new_mstate(curthread, mstate);
1221 /* Kernel probe */
1222 TNF_PROBE_1(thread_state, "thread", /* CSTYLED */,
1223             tnf_microstate, state, LMS_USER);

1225 TRACE_0(TR_FAC_TRAP, TR_C_TRAP_HANDLER_EXIT, "C_trap_handler_exit");
1226 return;

1228 cleanup:    /* system traps end up here */

```

```

1229     ASSERT(!(type & T_USER));

1231     TRACE_0(TR_FAC_TRAP, TR_C_TRAP_HANDLER_EXIT, "C_trap_handler_exit");
1232 }
_____ unchanged_portion_omitted _____

1343 /*
1344 * Called from fp_traps when a floating point trap occurs.
1345 * Note that the T_DATA_EXCEPTION case does not use X_FAULT_TYPE(mmu_fsr),
1346 * because mmu_fsr (now changed to code) is always 0.
1347 * Note that the T_UNIMP_INSTR case does not call simulate_unimp(),
1348 * because the simulator only simulates multiply and divide instructions,
1349 * which would not cause floating point traps in the first place.
1350 * XXX - Supervisor mode floating point traps?
1351 */
1352 void
1353 fpu_trap(struct regs *rp, caddr_t addr, uint32_t type, uint32_t code)
1354 {
1355     proc_t *p = ttoproc(curthread);
1356     klpw_id_t lwp = ttolwp(curthread);
1357     k_siginfo_t siginfo;
1358     uint_t op3, fault = 0;
1359     int mstate;
1360     char *badaddr;
1361     kfpw_t *fp;
1362     struct_fpq *fpq;
1363     uint32_t inst;
1364     utrap_handler_t *utrapp;

1366     CPU_STATS_ADDQ(CPU, sys, trap, 1);

1375     ASSERT(curthread->t_schedflag & TS_DONT_SWAP);

1368     if (USERMODE(rp->r_tstate)) {
1369         /*
1370          * Set lwp_state before trying to acquire any
1371          * adaptive lock
1372          */
1373         ASSERT(lwp != NULL);
1374         lwp->lwp_state = LWP_SYS;
1375         /*
1376          * Set up the current cred to use during this trap. u_cred
1377          * no longer exists. t_cred is used instead.
1378          * The current process credential applies to the thread for
1379          * the entire trap. If trapping from the kernel, this
1380          * should already be set up.
1381          */
1382         if (curthread->t_cred != p->p_cred) {
1383             cred_t *oldcred = curthread->t_cred;
1384             /*
1385              * DTrace accesses t_cred in probe context. t_cred
1386              * must always be either NULL, or point to a valid,
1387              * allocated cred structure.
1388              */
1389             curthread->t_cred = crgetcred();
1390             crfree(oldcred);
1391         }
1392         ASSERT(lwp->lwp_regs == rp);
1393         mstate = new_mstate(curthread, LMS_TRAP);
1394         siginfo.si_signo = 0;
1395         type |= T_USER;
1396     }

1398     TRACE_1(TR_FAC_TRAP, TR_C_TRAP_HANDLER_ENTER,
1399            "C_fpu_trap_handler_enter: type %x", type);

```

```

1401     if (tudebug && tudebugfpe)
1402         showregs(type, rp, addr, 0);

1404     bzero(&siginfo, sizeof (siginfo));
1405     siginfo.si_code = code;
1406     siginfo.si_addr = addr;

1408     switch (type) {

1410     case T_FP_EXCEPTION_IEEE + T_USER:      /* FPU arithmetic exception */
1411         /*
1412          * FPU arithmetic exception - fake up a fpq if we
1413          * came here directly from _fp_ieee_exception,
1414          * which is indicated by a zero fpu_qcnt.
1415          */
1416         fp = lwptofpu(curthread->t_lwp);
1417         utrapp = curthread->t_procp->p_utraps;
1418         if (fp->fpu_qcnt == 0) {
1419             inst = fetch_user_instr((caddr_t)rp->r_pc);
1420             lwp->lwp_state = LWP_SYS;
1421             pfpq = &fp->fpu_q->FQu.fpq;
1422             pfpq->fpq_addr = (uint32_t *)rp->r_pc;
1423             pfpq->fpq_instr = inst;
1424             fp->fpu_qcnt = 1;
1425             fp->fpu_q_entrysize = sizeof (struct _fpq);
1426 #ifdef SF_V9_TABLE_28
1427             /*
1428              * Spitfire and blackbird followed the SPARC V9 manual
1429              * paragraph 3 of section 5.1.7.9 FSR_current_exception
1430              * (cexc) for setting fsr.cexc bits on underflow and
1431              * overflow traps when the fsr.tem.inexact bit is set,
1432              * instead of following Table 28. Bugid 1263234.
1433              */
1434             {
1435                 extern int spitfire_bb_fsr_bug;

1437                 if (spitfire_bb_fsr_bug &&
1438                     (fp->fpu_fsr & FSR_TEM_NX)) {
1439                     if (((fp->fpu_fsr & FSR_TEM_OF) == 0) &&
1440                         (fp->fpu_fsr & FSR_CEXC_OF)) {
1441                         fp->fpu_fsr &= ~FSR_CEXC_OF;
1442                         fp->fpu_fsr |= FSR_CEXC_NX;
1443                         _fp_write_pfsr(&fp->fpu_fsr);
1444                         siginfo.si_code = FPE_FLTRES;
1445                     }
1446                     if (((fp->fpu_fsr & FSR_TEM_UF) == 0) &&
1447                         (fp->fpu_fsr & FSR_CEXC_UF)) {
1448                         fp->fpu_fsr &= ~FSR_CEXC_UF;
1449                         fp->fpu_fsr |= FSR_CEXC_NX;
1450                         _fp_write_pfsr(&fp->fpu_fsr);
1451                         siginfo.si_code = FPE_FLTRES;
1452                     }
1453                 }
1454             }
1455 #endif /* SF_V9_TABLE_28 */
1456             rp->r_pc = rp->r_npc;
1457             rp->r_npc += 4;
1458         } else if (utrapp && utrapp[UT_FP_EXCEPTION_IEEE_754]) {
1459             /*
1460              * The user had a trap handler installed. Jump to
1461              * the trap handler instead of signalling the process.
1462              */
1463             rp->r_pc = (long)utrapp[UT_FP_EXCEPTION_IEEE_754];
1464             rp->r_npc = rp->r_pc + 4;
1465             break;
1466         }

```

```

1467         siginfo.si_signo = SIGFPE;
1468         fault = FLTFPE;
1469         break;

1471     case T_DATA_EXCEPTION + T_USER:      /* user data access exception */
1472         siginfo.si_signo = SIGSEGV;
1473         fault = FLTBOUNDS;
1474         break;

1476     case T_LDDF_ALIGN + T_USER: /* 64 bit user lddfa alignment error */
1477     case T_STDF_ALIGN + T_USER: /* 64 bit user stdfa alignment error */
1478         alignfaults++;
1479         lwp->lwp_state = LWP_SYS;
1480         if (&visl_partial_support != NULL) {
1481             bzero(&siginfo, sizeof (siginfo));
1482             if (visl_partial_support(rp,
1483                 &siginfo, &fault) == 0)
1484                 goto out;
1485         }
1486         if (do_unaligned(rp, &badaddr) == SIMU_SUCCESS) {
1487             rp->r_pc = rp->r_npc;
1488             rp->r_npc += 4;
1489             goto out;
1490         }
1491         fp = lwptofpu(curthread->t_lwp);
1492         fp->fpu_qcnt = 0;
1493         siginfo.si_signo = SIGSEGV;
1494         siginfo.si_code = SEGV_MAPERR;
1495         siginfo.si_addr = badaddr;
1496         fault = FLTBOUNDS;
1497         break;

1499     case T_ALIGNMENT + T_USER:          /* user alignment error */
1500         /*
1501          * If the user has to do unaligned references
1502          * the ugly stuff gets done here.
1503          * Only handles vanilla loads and stores.
1504          */
1505         alignfaults++;
1506         if (p->p_fixalignment) {
1507             if (do_unaligned(rp, &badaddr) == SIMU_SUCCESS) {
1508                 rp->r_pc = rp->r_npc;
1509                 rp->r_npc += 4;
1510                 goto out;
1511             }
1512             siginfo.si_signo = SIGSEGV;
1513             siginfo.si_code = SEGV_MAPERR;
1514             siginfo.si_addr = badaddr;
1515             fault = FLTBOUNDS;
1516         } else {
1517             siginfo.si_signo = SIGBUS;
1518             siginfo.si_code = BUS_ADRALN;
1519             if (rp->r_pc & 3) { /* offending address, if pc */
1520                 siginfo.si_addr = (caddr_t)rp->r_pc;
1521             } else {
1522                 if (calc_memaddr(rp, &badaddr) == SIMU_UNALIGN)
1523                     siginfo.si_addr = badaddr;
1524                 else
1525                     siginfo.si_addr = (caddr_t)rp->r_pc;
1526             }
1527             fault = FLTACCESS;
1528         }
1529         break;

1531     case T_UNIMP_INSTR + T_USER:        /* illegal instruction fault */
1532         siginfo.si_signo = SIGILL;

```

```
1533         inst = fetch_user_instr((caddr_t)rp->r_pc);
1534         op3 = (inst >> 19) & 0x3F;
1535         if ((op3 == IOP_V8_STQFA) || (op3 == IOP_V8_STDFA))
1536             siginfo.si_code = ILL_ILLADR;
1537         else
1538             siginfo.si_code = ILL_ILLTRP;
1539         fault = FLTILL;
1540         break;
1541
1542     default:
1543         (void) die(type, rp, addr, 0);
1544         /*NOTREACHED*/
1545     }
1546
1547     /*
1548     * We can't get here from a system trap
1549     * Never restart any instruction which got here from an fp trap.
1550     */
1551     ASSERT(type & T_USER);
1552
1553     trap_cleanup(rp, fault, &siginfo, 0);
1554 out:
1555     trap_rtt();
1556     (void) new_mstate(curthread, mstate);
1557 }
1558
1559 unchanged_portion_omitted
```