```
**********************************************************
   51775 Sun Apr  7 23:57:51 2013
new/usr/src/uts/common/disp/thread.c
3625 we only need one thread_create_intr
**********************************************************
______unchanged_portion_omitted_

 313 /*
 314  * Create a thread.
 315  *
 316  * thread_create() blocks for memory if necessary.  It never fails.
 317  *
 318  * If stk is NULL, the thread is created at the base of the stack
 319  * and cannot be swapped.
 320  */
 321 kthread_t *
 322 thread_create(caddr_t stk, size_t stksize, void (*proc)(), void *arg,
 323     size_t len, proc_t *pp, int state, pri_t pri)
 322 thread_create(
 323         caddr_t stk,
 324         size_t  stksize,
 325         void    (*proc)(),
 326         void    *arg,
 327         size_t  len,
 328         proc_t   *pp,
 329         int     state,
 330         pri_t   pri)
 324 {
 325         kthread_t *t;
 326         extern struct classfuncs sys_classfuncs;
 327         turnstile_t *ts;

 329         /*
 330          * Every thread keeps a turnstile around in case it needs to block.
 331          * The only reason the turnstile is not simply part of the thread
 332          * structure is that we may have to break the association whenever
 333          * more than one thread blocks on a given synchronization object.
 334          * From a memory-management standpoint, turnstiles are like the
 335          * "attached mblks" that hang off dblks in the streams allocator.
 336          */
 337         ts = kmem_cache_alloc(turnstile_cache, KM_SLEEP);

 339         if (stk == NULL) {
 340                 /*
 341                  * alloc both thread and stack in segkp chunk
 342                  */

 344                 if (stksize < default_stksize)
 345                         stksize = default_stksize;

 347                 if (stksize == default_stksize) {
 348                         stk = (caddr_t)segkp_cache_get(segkp_thread);
 349                 } else {
 350                         stksize = roundup(stksize, PAGESIZE);
 351                         stk = (caddr_t)segkp_get(segkp, stksize,
 352                             (KPD_HASREDZONE | KPD_NO_ANON | KPD_LOCKED));
 353                 }

 355                 ASSERT(stk != NULL);

 357                 /*
 358                  * The machine-dependent mutex code may require that
 359                  * thread pointers (since they may be used for mutex owner
 360                  * fields) have certain alignment requirements.
 361                  * PTR24_ALIGN is the size of the alignment quanta.
 362                  * XXX - assumes stack grows toward low addresses.
```



```
 363                  */
 364                 if (stksize <= sizeof (kthread_t) + PTR24_ALIGN)
 365                         cmn_err(CE_PANIC, "thread_create: proposed stack size"
 366                             " too small to hold thread.");
 367 #ifdef STACK_GROWTH_DOWN
 368                 stksize -= SA(sizeof (kthread_t) + PTR24_ALIGN - 1);
 369                 stksize &= -PTR24_ALIGN;        /* make thread aligned */
 370                 t = (kthread_t *)(stk + stksize);
 371                 bzero(t, sizeof (kthread_t));
 372                 if (audit_active)
 373                         audit_thread_create(t);
 374                 t->t_stk = stk + stksize;
 375                 t->t_stkbase = stk;
 376 #else   /* stack grows to larger addresses */
 377                 stksize -= SA(sizeof (kthread_t));
 378                 t = (kthread_t *)(stk);
 379                 bzero(t, sizeof (kthread_t));
 380                 t->t_stk = stk + sizeof (kthread_t);
 381                 t->t_stkbase = stk + stksize + sizeof (kthread_t);
 382 #endif  /* STACK_GROWTH_DOWN */
 383                 t->t_flag |= T_TALLOCSTK;
 384                 t->t_swap = stk;
 385         } else {
 386                 t = kmem_cache_alloc(thread_cache, KM_SLEEP);
 387                 bzero(t, sizeof (kthread_t));
 388                 ASSERT(((uintptr_t)t & (PTR24_ALIGN - 1)) == 0);
 389                 if (audit_active)
 390                         audit_thread_create(t);
 391                 /*
 392                  * Initialize t_stk to the kernel stack pointer to use
 393                  * upon entry to the kernel
 394                  */
 395 #ifdef STACK_GROWTH_DOWN
 396                 t->t_stk = stk + stksize;
 397                 t->t_stkbase = stk;
 398 #else
 399                 t->t_stk = stk;                 /* 3b2-like */
 400                 t->t_stkbase = stk + stksize;
 401 #endif /* STACK_GROWTH_DOWN */
 402         }

 404         if (kmem_stackinfo != 0) {
 405                 stkinfo_begin(t);
 406         }

 408         t->t_ts = ts;

 410         /*
 411          * p_cred could be NULL if it thread_create is called before cred_init
 412          * is called in main.
 413          */
 414         mutex_enter(&pp->p_crlock);
 415         if (pp->p_cred)
 416                 crhold(t->t_cred = pp->p_cred);
 417         mutex_exit(&pp->p_crlock);
 418         t->t_start = gethrestime_sec();
 419         t->t_startpc = proc;
 420         t->t_procp = pp;
 421         t->t_clfuncs = &sys_classfuncs.thread;
 422         t->t_cid = syscid;
 423         t->t_pri = pri;
 424         t->t_stime = ddi_get_lbolt();
 425         t->t_schedflag = TS_LOAD | TS_DONT_SWAP;
 426         t->t_bind_cpu = PBIND_NONE;
 427         t->t_bindflag = (uchar_t)default_binding_mode;
 428         t->t_bind_pset = PS_NONE;
```

```
429         t->t_plockp = &pp->p_lock;
430         t->t_copyops = NULL;
431         t->t_taskq = NULL;
432         t->t_anttime = 0;
433         t->t_hatdepth = 0;

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

437         CPU_STATS_ADDQ(CPU, sys, nthreads, 1);
438 #ifndef NPROBE
439         /* Kernel probe */
440         tnf_thread_create(t);
441 #endif /* NPROBE */
442         LOCK_INIT_CLEAR(&t->t_lock);

444         /*
445          * Callers who give us a NULL proc must do their own
446          * stack initialization.  e.g. lwp_create()
447          */
448         if (proc != NULL) {
449                 t->t_stk = thread_stk_init(t->t_stk);
450                 thread_load(t, proc, arg, len);
451         }

453         /*
454          * Put a hold on project0. If this thread is actually in a
455          * different project, then t_proj will be changed later in
456          * lwp_create().  All kernel-only threads must be in project 0.
457          */
458         t->t_proj = project_hold(proj0p);

460         lgrp_affinity_init(&t->t_lgrp_affinity);

462         mutex_enter(&pidlock);
463         nthread++;
464         t->t_did = next_t_id++;
465         t->t_prev = curthread->t_prev;
466         t->t_next = curthread;

468         /*
469          * Add the thread to the list of all threads, and initialize
470          * its t_cpu pointer.  We need to block preemption since
471          * cpu_offline walks the thread list looking for threads
472          * with t_cpu pointing to the CPU being offlined.  We want
473          * to make sure that the list is consistent and that if t_cpu
474          * is set, the thread is on the list.
475          */
476         kpreempt_disable();
477         curthread->t_prev->t_next = t;
478         curthread->t_prev = t;

480         /*
481          * Threads should never have a NULL t_cpu pointer so assign it
482          * here.  If the thread is being created with state TS_RUN a
483          * better CPU may be chosen when it is placed on the run queue.
484          *
485          * We need to keep kernel preemption disabled when setting all
486          * three fields to keep them in sync.  Also, always create in
487          * the default partition since that's where kernel threads go
488          * (if this isn't a kernel thread, t_cpupart will be changed
489          * in lwp_create before setting the thread runnable).
490          */
491         t->t_cpupart = &cp_default;

493         /*
494          * For now, affiliate this thread with the root lgroup.
```



```
495          * Since the kernel does not (presently) allocate its memory
496          * in a locality aware fashion, the root is an appropriate home.
497          * If this thread is later associated with an lwp, it will have
498          * it's lgroup re-assigned at that time.
499          */
500         lgrp_move_thread(t, &cp_default.cp_lgrploads[LGRP_ROOTID], 1);

502         /*
503          * Inherit the current cpu.  If this cpu isn't part of the chosen
504          * lgroup, a new cpu will be chosen by cpu_choose when the thread
505          * is ready to run.
506          */
507         if (CPU->cpu_part == &cp_default)
508                 t->t_cpu = CPU;
509         else
510                 t->t_cpu = disp_lowpri_cpu(cp_default.cp_cpulist, t->t_lpl,
511                     t->t_pri, NULL);

513         t->t_disp_queue = t->t_cpu->cpu_disp;
514         kpreempt_enable();

516         /*
517          * Initialize thread state and the dispatcher lock pointer.
518          * Need to hold onto pidlock to block allthreads walkers until
519          * the state is set.
520          */
521         switch (state) {
522         case TS_RUN:
523                 curthread->t_oldspl = splhigh();        /* get dispatcher spl */
524                 THREAD_SET_STATE(t, TS_STOPPED, &transition_lock);
525                 CL_SETRUN(t);
526                 thread_unlock(t);
527                 break;

529         case TS_ONPROC:
530                 THREAD_ONPROC(t, t->t_cpu);
531                 break;

533         case TS_FREE:
534                 /*
535                  * Free state will be used for intr threads.
536                  * The interrupt routine must set the thread dispatcher
537                  * lock pointer (t_lockp) if starting on a CPU
538                  * other than the current one.
539                  */
540                 THREAD_FREEINTR(t, CPU);
541                 break;

543         case TS_STOPPED:
544                 THREAD_SET_STATE(t, TS_STOPPED, &stop_lock);
545                 break;

547         default:                        /* TS_SLEEP, TS_ZOMB or TS_TRANS */
548                 cmn_err(CE_PANIC, "thread_create: invalid state %d", state);
549         }
550         mutex_exit(&pidlock);
551         return (t);
552 }
```

_____***unchanged_portion_omitted_***

```
883 /*
884  * cleanup zombie threads that are on deathrow.
885  */
886 void
887 thread_reaper(void)
894 thread_reaper()
```

```
 888 {
 889         kthread_t *t, *l;
 890         callb_cpr_t cprinfo;

 892         /*
 893          * Register callback to clean up threads when zone is destroyed.
 894          */
 895         zone_key_create(&zone_thread_key, NULL, NULL, thread_zone_destroy);

 897         CALLB_CPR_INIT(&cprinfo, &reaplock, callb_generic_cpr, "t_reaper");
 898         for (;;) {
 899                 mutex_enter(&reaplock);
 900                 while (thread_deathrow == NULL && lwp_deathrow == NULL) {
 901                         CALLB_CPR_SAFE_BEGIN(&cprinfo);
 902                         cv_wait(&reaper_cv, &reaplock);
 903                         CALLB_CPR_SAFE_END(&cprinfo, &reaplock);
 904                 }
 905                 /*
 906                  * mutex_sync() needs to be called when reaping, but
 907                  * not too often.  We limit reaping rate to once
 908                  * per second.  Reaplimit is max rate at which threads can
 909                  * be freed. Does not impact thread destruction/creation.
 910                  */
 911                 t = thread_deathrow;
 912                 l = lwp_deathrow;
 913                 thread_deathrow = NULL;
 914                 lwp_deathrow = NULL;
 915                 thread_reapcnt = 0;
 916                 lwp_reapcnt = 0;
 917                 mutex_exit(&reaplock);

 919                 /*
 920                  * Guard against race condition in mutex_owner_running:
 921                  *      thread=owner(mutex)
 922                  *      <interrupt>
 923                  *                                  thread exits mutex
 924                  *                                  thread exits
 925                  *                                  thread reaped
 926                  *                                  thread struct freed
 927                  * cpu = thread->t_cpu <- BAD POINTER DEREFERENCE.
 928                  * A cross call to all cpus will cause the interrupt handler
 929                  * to reset the PC if it is in mutex_owner_running, refreshing
 930                  * stale thread pointers.
 931                  */
 932                 mutex_sync();   /* sync with mutex code */
 933                 /*
 934                  * Reap threads
 935                  */
 936                 thread_reap_list(t);

 938                 /*
 939                  * Reap lwps
 940                  */
 941                 thread_reap_list(l);
 942                 delay(hz);
 943         }
 944 }
_____unchanged_portion_omitted_

1020 /*
1021  * Install thread context ops for the current thread.
1022  */
1023 void
1024 installctx(kthread_t *t, void *arg, void (*save)(void *),
1025     void (*restore)(void *), void (*fork)(void *, void *),
1026     void (*lwp_create)(void *, void *), void (*exit)(void *),
```



```
1031 installctx(
1032         kthread_t *t,
1033         void    *arg,
1034         void    (*save)(void *),
1035         void    (*restore)(void *),
1036         void    (*fork)(void *, void *),
1037         void    (*lwp_create)(void *, void *),
1038         void    (*exit)(void *),
1027     void (*free)(void *, int))
1028 {
1029         struct ctxop *ctx;

1031         ctx = kmem_alloc(sizeof (struct ctxop), KM_SLEEP);
1032         ctx->save_op = save;
1033         ctx->restore_op = restore;
1034         ctx->fork_op = fork;
1035         ctx->lwp_create_op = lwp_create;
1036         ctx->exit_op = exit;
1037         ctx->free_op = free;
1038         ctx->arg = arg;
1039         ctx->next = t->t_ctx;
1040         t->t_ctx = ctx;
1041 }

1043 /*
1044  * Remove the thread context ops from a thread.
1045  */
1046 int
1047 removectx(kthread_t *t, void *arg, void (*save)(void *),
1048     void (*restore)(void *), void (*fork)(void *, void *),
1049     void (*lwp_create)(void *, void *), void (*exit)(void *),
1059 removectx(
1060         kthread_t *t,
1061         void    *arg,
1062         void    (*save)(void *),
1063         void    (*restore)(void *),
1064         void    (*fork)(void *, void *),
1065         void    (*lwp_create)(void *, void *),
1066         void    (*exit)(void *),
1050     void (*free)(void *, int))
1051 {
1052         struct ctxop *ctx, *prev_ctx;

1054         /*
1055          * The incoming kthread_t (which is the thread for which the
1056          * context ops will be removed) should be one of the following:
1057          *
1058          * a) the current thread,
1059          *
1060          * b) a thread of a process that's being forked (SIDL),
1061          *
1062          * c) a thread that belongs to the same process as the current
1063          *    thread and for which the current thread is the agent thread,
1064          *
1065          * d) a thread that is TS_STOPPED which is indicative of it
1066          *    being (if curthread is not an agent) a thread being created
1067          *    as part of an lwp creation.
1068          */
1069         ASSERT(t == curthread || ttoproc(t)->p_stat == SIDL ||
1070             ttoproc(t)->p_agenttp == curthread || t->t_state == TS_STOPPED);

1072         /*
1073          * Serialize modifications to t->t_ctx to prevent the agent thread
1074          * and the target thread from racing with each other during lwp exit.
1075          */
1076         mutex_enter(&t->t_ctx_lock);
```

```
1077         prev_ctx = NULL;
1078         for (ctx = t->t_ctx; ctx != NULL; ctx = ctx->next) {
1079                 if (ctx->save_op == save && ctx->restore_op == restore &&
1080                     ctx->fork_op == fork && ctx->lwp_create_op == lwp_create &&
1081                     ctx->exit_op == exit && ctx->free_op == free &&
1082                     ctx->arg == arg) {
1083                         if (prev_ctx)
1084                                 prev_ctx->next = ctx->next;
1085                         else
1086                                 t->t_ctx = ctx->next;
1087                         mutex_exit(&t->t_ctx_lock);
1088                         if (ctx->free_op != NULL)
1089                                 (ctx->free_op)(ctx->arg, 0);
1090                         kmem_free(ctx, sizeof (struct ctxop));
1091                         return (1);
1092                 }
1093                 prev_ctx = ctx;
1094         }
1095         mutex_exit(&t->t_ctx_lock);

1097         return (0);
1098 }
```

*_____unchanged_portion_omitted_*

```
1273 /*
1274  * Unpin an interrupted thread.
1275  *      When an interrupt occurs, the interrupt is handled on the stack
1276  *      of an interrupt thread, taken from a pool linked to the CPU structure.
1277  *
1278  *      When swtch() is switching away from an interrupt thread because it
1279  *      blocked or was preempted, this routine is called to complete the
1280  *      saving of the interrupted thread state, and returns the interrupted
1281  *      thread pointer so it may be resumed.
1282  *
1283  *      Called by swtch() only at high spl.
1284  */
1285 kthread_t *
1286 thread_unpin(void)
1303 thread_unpin()
1287 {
1288         kthread_t       *t = curthread; /* current thread */
1289         kthread_t       *itp;           /* interrupted thread */
1290         int             i;              /* interrupt level */
1291         extern int      intr_passivate();

1293         ASSERT(t->t_intr != NULL);

1295         itp = t->t_intr;                /* interrupted thread */
1296         t->t_intr = NULL;               /* clear interrupt ptr */

1298         /*
1299          * Get state from interrupt thread for the one
1300          * it interrupted.
1301          */

1303         i = intr_passivate(t, itp);

1305         TRACE_5(TR_FAC_INTR, TR_INTR_PASSIVATE,
1306             "intr_passivate:level %d curthread %p (%T) ithread %p (%T)",
1307             i, t, t, itp, itp);

1309         /*
1310          * Dissociate the current thread from the interrupted thread's LWP.
1311          */
1312         t->t_lwp = NULL;
```



```
1314         /*
1315          * Interrupt handlers above the level that spinlocks block must
1316          * not block.
1317          */
1318 #if DEBUG
1319         if (i < 0 || i > LOCK_LEVEL)
1320                 cmn_err(CE_PANIC, "thread_unpin: ipl out of range %x", i);
1321 #endif

1323         /*
1324          * Compute the CPU's base interrupt level based on the active
1325          * interrupts.
1326          */
1327         ASSERT(CPU->cpu_intr_actv & (1 << i));
1328         set_base_spl();

1330         return (itp);
1331 }

1333 /*
1351  * Create and initialize an interrupt thread.
1352  *      Returns non-zero on error.
1353  *      Called at spl7() or better.
1354  */
1355 void
1356 thread_create_intr(struct cpu *cp)
1357 {
1358         kthread_t *tp;

1360         tp = thread_create(NULL, 0,
1361             (void (*)())thread_create_intr, NULL, 0, &p0, TS_ONPROC, 0);

1363         /*
1364          * Set the thread in the TS_FREE state.  The state will change
1365          * to TS_ONPROC only while the interrupt is active.  Think of these
1366          * as being on a private free list for the CPU.  Being TS_FREE keeps
1367          * inactive interrupt threads out of debugger thread lists.
1368          *
1369          * We cannot call thread_create with TS_FREE because of the current
1370          * checks there for ONPROC.  Fix this when thread_create takes flags.
1371          */
1372         THREAD_FREEINTR(tp, cp);

1374         /*
1375          * Nobody should ever reference the credentials of an interrupt
1376          * thread so make it NULL to catch any such references.
1377          */
1378         tp->t_cred = NULL;
1379         tp->t_flag |= T_INTR_THREAD;
1380         tp->t_cpu = cp;
1381         tp->t_bound_cpu = cp;
1382         tp->t_disp_queue = cp->cpu_disp;
1383         tp->t_affinitycnt = 1;
1384         tp->t_preempt = 1;

1386         /*
1387          * Don't make a user-requested binding on this thread so that
1388          * the processor can be offlined.
1389          */
1390         tp->t_bind_cpu = PBIND_NONE;    /* no USER-requested binding */
1391         tp->t_bind_pset = PS_NONE;

1393 #if defined(__i386) || defined(__amd64)
1394         tp->t_stk -= STACK_ALIGN;
1395         *(tp->t_stk) = 0;               /* terminate intr thread stack */
1396 #endif
```

```
1398         /*
1399          * Link onto CPU's interrupt pool.
1400          */
1401         tp->t_link = cp->cpu_intr_thread;
1402         cp->cpu_intr_thread = tp;
1403 }

1405 /*
1334  * TSD -- THREAD SPECIFIC DATA
1335  */
1336 static kmutex_t         tsd_mutex;       /* linked list spin lock */
1337 static uint_t           tsd_nkeys;       /* size of destructor array */
1338 /* per-key destructor funcs */
1339 static void             (**tsd_destructor)(void *);
1340 /* list of tsd_thread's */
1341 static struct tsd_thread        *tsd_list;

1343 /*
1344  * Default destructor
1345  *      Needed because NULL destructor means that the key is unused
1346  */
1347 /* ARGSUSED */
1348 void
1349 tsd_defaultdestructor(void *value)
1350 {}

1352 /*
1353  * Create a key (index into per thread array)
1354  *      Locks out tsd_create, tsd_destroy, and tsd_exit
1355  *      May allocate memory with lock held
1356  */
1357 void
1358 tsd_create(uint_t *keyp, void (*destructor)(void *))
1359 {
1360         int     i;
1361         uint_t  nkeys;

1363         /*
1364          * if key is allocated, do nothing
1365          */
1366         mutex_enter(&tsd_mutex);
1367         if (*keyp) {
1368                 mutex_exit(&tsd_mutex);
1369                 return;
1370         }
1371         /*
1372          * find an unused key
1373          */
1374         if (destructor == NULL)
1375                 destructor = tsd_defaultdestructor;

1377         for (i = 0; i < tsd_nkeys; ++i)
1378                 if (tsd_destructor[i] == NULL)
1379                         break;

1381         /*
1382          * if no unused keys, increase the size of the destructor array
1383          */
1384         if (i == tsd_nkeys) {
1385                 if ((nkeys = (tsd_nkeys << 1)) == 0)
1386                         nkeys = 1;
1387                 tsd_destructor =
1388                     (void (**)(void *))tsd_realloc((void *)tsd_destructor,
1389                     (size_t)(tsd_nkeys * sizeof (void (*)(void *))),
1390                     (size_t)(nkeys * sizeof (void (*)(void *))));
```



```
1391                 tsd_nkeys = nkeys;
1392         }

1394         /*
1395          * allocate the next available unused key
1396          */
1397         tsd_destructor[i] = destructor;
1398         *keyp = i + 1;
1399         mutex_exit(&tsd_mutex);
1400 }
```

_____***unchanged_portion_omitted_***

****
2882 Sun Apr 7 23:57:52 2013
new/usr/src/uts/common/disp/thread_intr.c
3625 we only need one thread_create_intr
****

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

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

/*
 * FILE NOTICE BEGIN
 *
 * This file should not be modified.  If you wish to modify it or have it
 * modified, please contact Sun Microsystems at <LFI149367@-sun-.-com->
 * (without anti-spam dashes)
 *
 * FILE NOTICE END
 */

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

#include <sys/cpuvar.h>
#include <sys/stack.h>
#include <vm/seg_kp.h>
#include <sys/proc.h>
#include <sys/pset.h>
#include <sys/sysmacros.h>

/*
 * Create and initialize an interrupt thread.
 */
static void
thread_create_intr(cpu_t *cp)
{
        kthread_t *tp;

        tp = thread_create(NULL, 0,
            (void (*)())thread_create_intr, NULL, 0, &p0, TS_ONPROC, 0);

        /*
         * Set the thread in the TS_FREE state.  The state will change
         * to TS_ONPROC only while the interrupt is active.  Think of these
         * as being on a private free list for the CPU.  Being TS_FREE keeps
         * inactive interrupt threads out of debugger thread lists.
```



```
         *
         * We cannot call thread_create with TS_FREE because of the current
         * checks there for ONPROC.  Fix this when thread_create takes flags.
         */
        THREAD_FREEINTR(tp, cp);

        /*
         * Nobody should ever reference the credentials of an interrupt
         * thread so make it NULL to catch any such references.
         */
        tp->t_cred = NULL;
        tp->t_flag |= T_INTR_THREAD;
        tp->t_cpu = cp;
        tp->t_bound_cpu = cp;
        tp->t_disp_queue = cp->cpu_disp;
        tp->t_affinitycnt = 1;
        tp->t_preempt = 1;

        /*
         * Don't make a user-requested binding on this thread so that
         * the processor can be offlined.
         */
        tp->t_bind_cpu = PBIND_NONE;    /* no USER-requested binding */
        tp->t_bind_pset = PS_NONE;

#if defined(__i386) || defined(__amd64)
        tp->t_stk -= STACK_ALIGN;
        *(tp->t_stk) = 0;               /* terminate intr thread stack */
#endif

        /*
         * Link onto CPU's interrupt pool.
         */
        tp->t_link = cp->cpu_intr_thread;
        cp->cpu_intr_thread = tp;
}
```

_____unchanged_portion_omitted_

```
**********************************************************
   29252 Sun Apr  7 23:57:52 2013
new/usr/src/uts/common/sys/proc.h
3625 we only need one thread_create_intr
**********************************************************
______unchanged_portion_omitted_

 577 #ifdef _KERNEL

 579 /* user profiling functions */

 581 extern void profil_tick(uintptr_t);

 583 /* process management functions */

 585 extern int newproc(void (*)(), caddr_t, id_t, int, struct contract **, pid_t);
 586 extern void vfwait(pid_t);
 587 extern void proc_detach(proc_t *);
 588 extern void freeproc(proc_t *);
 589 extern void setrun(kthread_t *);
 590 extern void setrun_locked(kthread_t *);
 591 extern void exit(int, int);
 592 extern int proc_exit(int, int);
 593 extern void proc_is_exiting(proc_t *);
 594 extern void relvm(void);
 595 extern void add_ns(proc_t *, proc_t *);
 596 extern void delete_ns(proc_t *, proc_t *);
 597 extern void upcount_inc(uid_t, zoneid_t);
 598 extern void upcount_dec(uid_t, zoneid_t);
 599 extern int upcount_get(uid_t, zoneid_t);
 600 #if defined(__x86)
 601 extern selector_t setup_thrptr(proc_t *, uintptr_t);
 602 extern void deferred_singlestep_trap(caddr_t);
 603 #endif

 605 extern void sigcld(proc_t *, sigqueue_t *);
 606 extern void sigcld_delete(k_siginfo_t *);
 607 extern void sigcld_repost(void);
 608 extern int fsig(k_sigset_t *, kthread_t *);
 609 extern void psig(void);
 610 extern void stop(int, int);
 611 extern int stop_on_fault(uint_t, k_siginfo_t *);
 612 extern int issig(int);
 613 extern int jobstopped(proc_t *);
 614 extern void psignal(proc_t *, int);
 615 extern void tsignal(kthread_t *, int);
 616 extern void sigtoproc(proc_t *, kthread_t *, int);
 617 extern void trapsig(k_siginfo_t *, int);
 618 extern void realsigprof(int, int, int);
 619 extern int eat_signal(kthread_t *, int);
 620 extern int signal_is_blocked(kthread_t *, int);
 621 extern int sigcheck(proc_t *, kthread_t *);
 622 extern void sigdefault(proc_t *);

 624 extern void pid_setmin(void);
 625 extern pid_t pid_allocate(proc_t *, pid_t, int);
 626 extern int pid_rele(struct pid *);
 627 extern void pid_exit(proc_t *, struct task *);
 628 extern void proc_entry_free(struct pid *);
 629 extern proc_t *prfind(pid_t);
 630 extern proc_t *prfind_zone(pid_t, zoneid_t);
 631 extern proc_t *pgfind(pid_t);
 632 extern proc_t *pgfind_zone(pid_t, zoneid_t);
 633 extern proc_t *sprlock(pid_t);
 634 extern proc_t *sprlock_zone(pid_t, zoneid_t);
 635 extern int sprtrylock_proc(proc_t *);
```



```
 636 extern void sprwaitlock_proc(proc_t *);
 637 extern void sprlock_proc(proc_t *);
 638 extern void sprunlock(proc_t *);
 639 extern void pid_init(void);
 640 extern proc_t *pid_entry(int);
 641 extern int pid_slot(proc_t *);
 642 extern void signal(pid_t, int);
 643 extern void prsignal(struct pid *, int);
 644 extern int uread(proc_t *, void *, size_t, uintptr_t);
 645 extern int uwrite(proc_t *, void *, size_t, uintptr_t);

 647 extern void pgsignal(struct pid *, int);
 648 extern void pgjoin(proc_t *, struct pid *);
 649 extern void pgcreate(proc_t *);
 650 extern void pgexit(proc_t *);
 651 extern void pgdetach(proc_t *);
 652 extern int pgmembers(pid_t);

 654 extern  void    init_mstate(kthread_t *, int);
 655 extern  int     new_mstate(kthread_t *, int);
 656 extern  void    restore_mstate(kthread_t *);
 657 extern  void    term_mstate(kthread_t *);
 658 extern  void    estimate_msacct(kthread_t *, hrtime_t);
 659 extern  void    disable_msacct(proc_t *);
 660 extern  hrtime_t mstate_aggr_state(proc_t *, int);
 661 extern  hrtime_t mstate_thread_onproc_time(kthread_t *);
 662 extern  void    mstate_systhread_times(kthread_t *, hrtime_t *, hrtime_t *);
 663 extern  void    syscall_mstate(int, int);

 665 extern  uint_t  cpu_update_pct(kthread_t *, hrtime_t);

 667 extern void     set_proc_pre_sys(proc_t *p);
 668 extern void     set_proc_post_sys(proc_t *p);
 669 extern void     set_proc_sys(proc_t *p);
 670 extern void     set_proc_ast(proc_t *p);
 671 extern void     set_all_proc_sys(void);
 672 extern void     set_all_zone_usr_proc_sys(zoneid_t);

 674 /* thread function prototypes */

 676 extern  kthread_t *thread_create(caddr_t, size_t, void (*)(), void *,
 677     size_t, proc_t *, int, pri_t);
 676 extern  kthread_t       *thread_create(
 677         caddr_t         stk,
 678         size_t          stksize,
 679         void            (*proc)(),
 680         void            *arg,
 681         size_t          len,
 682         proc_t          *pp,
 683         int             state,
 684         pri_t           pri);
 678 extern  void    thread_exit(void) __NORETURN;
 679 extern  void    thread_free(kthread_t *);
 680 extern  void    thread_rele(kthread_t *);
 681 extern  void    thread_join(kt_did_t);
 682 extern  int     reaper(void);
 683 extern  void    installctx(kthread_t *, void *, void (*)(), void (*)(),
 684     void (*)(), void (*)(), void (*)());
 685 extern  int     removectx(kthread_t *, void *, void (*)(), void (*)(),
 686     void (*)(), void (*)(), void (*)());
 687 extern  void    savectx(kthread_t *);
 688 extern  void    restorectx(kthread_t *);
 689 extern  void    forkctx(kthread_t *, kthread_t *);
 690 extern  void    lwp_createctx(kthread_t *, kthread_t *);
 691 extern  void    exitctx(kthread_t *);
 692 extern  void    freectx(kthread_t *, int);
```

```
693 extern  void    installpctx(proc_t *, void *, void (*)(), void (*)(),
694     void (*)(), void (*)(), void (*)());
695 extern  int     removepctx(proc_t *, void *, void (*)(), void (*)(),
696     void (*)(), void (*)(), void (*)());
697 extern  void    savepctx(proc_t *);
698 extern  void    restorepctx(proc_t *);
699 extern  void    forkpctx(proc_t *, proc_t *);
700 extern  void    exitpctx(proc_t *);
701 extern  void    freepctx(proc_t *, int);
702 extern  kthread_t *thread_unpin(void);
703 extern  void    thread_init(void);
704 extern  void    thread_load(kthread_t *, void (*)(), caddr_t, size_t);

706 extern  void    tsd_create(uint_t *, void (*)(void *));
707 extern  void    tsd_destroy(uint_t *);
708 extern  void    *tsd_getcreate(uint_t *, void (*)(void *), void *(*)(void));
709 extern  void    *tsd_get(uint_t);
710 extern  int     tsd_set(uint_t, void *);
711 extern  void    tsd_exit(void);
712 extern  void    *tsd_agent_get(kthread_t *, uint_t);
713 extern  int     tsd_agent_set(kthread_t *, uint_t, void *);

715 /* lwp function prototypes */

717 extern  kthread_t *lwp_kernel_create(proc_t *, void (*)(), void *, int, pri_t);
718 extern  klwp_t  *lwp_create(void (*)(), caddr_t, size_t, proc_t *, int, int,
719     const k_sigset_t *, int, id_t);
725 extern  klwp_t          *lwp_create(
726         void            (*proc)(),
727         caddr_t         arg,
728         size_t          len,
729         proc_t          *p,
730         int             state,
731         int             pri,
732         const k_sigset_t *smask,
733         int             cid,
734         id_t            lwpid);
720 extern  kthread_t *idtot(proc_t *, id_t);
721 extern  void    lwp_hash_in(proc_t *, lwpent_t *, tidhash_t *, uint_t, int);
722 extern  void    lwp_hash_out(proc_t *, id_t);
723 extern  lwpdir_t *lwp_hash_lookup(proc_t *, id_t);
724 extern  lwpdir_t *lwp_hash_lookup_and_lock(proc_t *, id_t, kmutex_t **);
725 extern  void    lwp_create_done(kthread_t *);
726 extern  void    lwp_exit(void);
727 extern  void    lwp_pcb_exit(void);
728 extern  void    lwp_cleanup(void);
729 extern  int     lwp_suspend(kthread_t *);
730 extern  void    lwp_continue(kthread_t *);
731 extern  void    holdlwp(void);
732 extern  void    stoplwp(void);
733 extern  int     holdlwps(int);
734 extern  int     holdwatch(void);
735 extern  void    pokelwps(proc_t *);
736 extern  void    continuelwps(proc_t *);
737 extern  int     exitlwps(int);
738 extern  void    lwp_ctmpl_copy(klwp_t *, klwp_t *);
739 extern  void    lwp_ctmpl_clear(klwp_t *);
740 extern  klwp_t  *forklwp(klwp_t *, proc_t *, id_t);
741 extern  void    lwp_load(klwp_t *, gregset_t, uintptr_t);
742 extern  void    lwp_setrval(klwp_t *, int, int);
743 extern  void    lwp_forkregs(klwp_t *, klwp_t *);
744 extern  void    lwp_freeregs(klwp_t *, int);
745 extern  caddr_t lwp_stk_init(klwp_t *, caddr_t);
746 extern  void    lwp_stk_cache_init(void);
747 extern  void    lwp_stk_fini(klwp_t *);
748 extern  void    lwp_installctx(klwp_t *);
```



```
749 extern  void    lwp_rtt(void);
750 extern  void    lwp_rtt_initial(void);
751 extern  int     lwp_setprivate(klwp_t *, int, uintptr_t);
752 extern  void    lwp_stat_update(lwp_stat_id_t, long);
753 extern  void    lwp_attach_brand_hdlrs(klwp_t *);
754 extern  void    lwp_detach_brand_hdlrs(klwp_t *);

756 #if defined(__sparcv9)
757 extern  void    lwp_mmodel_newlwp(void);
758 extern  void    lwp_mmodel_shared_as(caddr_t, size_t);
759 #define LWP_MMODEL_NEWLWP()             lwp_mmodel_newlwp()
760 #define LWP_MMODEL_SHARED_AS(addr, sz)  lwp_mmodel_shared_as((addr), (sz))
761 #else
762 #define LWP_MMODEL_NEWLWP()
763 #define LWP_MMODEL_SHARED_AS(addr, sz)
764 #endif

766 /*
767  * Signal queue function prototypes. Must be here due to header ordering
768  * dependencies.
769  */
770 extern void sigqfree(proc_t *);
771 extern void siginfofree(sigqueue_t *);
772 extern void sigdeq(proc_t *, kthread_t *, int, sigqueue_t **);
773 extern void sigdelq(proc_t *, kthread_t *, int);
774 extern void sigaddq(proc_t *, kthread_t *, k_siginfo_t *, int);
775 extern void sigaddqa(proc_t *, kthread_t *, sigqueue_t *);
776 extern void sigqsend(int, proc_t *, kthread_t *, sigqueue_t *);
777 extern void sigdupq(proc_t *, proc_t *);
778 extern int sigwillqueue(int, int);
779 extern sigqhdr_t *sigqhdralloc(size_t, uint_t);
780 extern sigqueue_t *sigqalloc(sigqhdr_t *);
781 extern void sigqhdrfree(sigqhdr_t *);
782 extern sigqueue_t *sigappend(k_sigset_t *, sigqueue_t *,
783         k_sigset_t *, sigqueue_t *);
784 extern sigqueue_t *sigprepend(k_sigset_t *, sigqueue_t *,
785         k_sigset_t *, sigqueue_t *);
786 extern void winfo(proc_t *, k_siginfo_t *, int);
787 extern int wstat(int, int);
788 extern int sendsig(int, k_siginfo_t *, void (*)());
789 #if defined(_SYSCALL32_IMPL)
790 extern int sendsig32(int, k_siginfo_t *, void (*)());
791 #endif

793 #endif  /* _KERNEL */

795 #ifdef  __cplusplus
796 }
```

*unchanged_portion_omitted_*