```
**********************************************************
   66807 Fri Mar 28 23:33:13 2014
new/usr/src/uts/common/disp/disp.c
patch delete-swapped_lock
patch remove-dead-disp-code
patch remove-useless-var2
patch remove-load-flag
patch remove-on-swapq-flag
patch remove-dont-swap-flag
**********************************************************
______unchanged_portion_omitted_
  75 static void     disp_dq_alloc(struct disp_queue_info *dptr, int numpris,
  76     disp_t *dp);
  77 static void     disp_dq_assign(struct disp_queue_info *dptr, int numpris);
  78 static void     disp_dq_free(struct disp_queue_info *dptr);

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

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

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

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

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

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

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

 123 /*
 124  * Parameter that determines how long (in nanoseconds) a thread must
 125  * be sitting on a run queue before it can be stolen by another CPU
 126  * to reduce migrations.  The interval is in nanoseconds.
 127  *
```



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

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

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

 143 static void     cpu_dispqalloc(int numpris);

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

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

 156 void    swtch_to(kthread_t *);

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

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

 171         ASSERT(MUTEX_HELD(&cpu_lock));

 173         newnglobpris = maxglobpri + 1 + LOCK_LEVEL;

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

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

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

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

694 extern kthread_t *thread_unpin();

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

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

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

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

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

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



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

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

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

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

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

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

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

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

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

818         /*
819          * Set TS_DONT_SWAP flag to prevent another processor from swapping
820          * out this thread before we have a chance to run it.
821          * While running, it is protected against swapping by t_lock.
```

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

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

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

 828         return (tp);
 829 }
```

unchanged_portion_omitted_

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

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

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

1194         /*
1195          * If thread is "swapped" or on the swap queue don't
1196          * queue it, but wake sched.
1197          */
```



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

1185         self = (tp == curthread);

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

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

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

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

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

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

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

1243                                 if (RUNQ_LEN(newcp, tpri) < qlen) {
1244                                         DTRACE_PROBE3(runq__balance,
1245                                             kthread_t *, tp,
```

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

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

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

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

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

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

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

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

1311    if (dq->dq_sruncnt++ != 0) {
```



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

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

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

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

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

1365         if (tp->t_bound_cpu || tp->t_weakbound_cpu)
1366                 bound = 1;
1367         else
1368                 bound = 0;
```

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

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

1430                 curtime = gethrtime_unscaled();
1431                 (void) cpu_update_pct(tp, curtime);
1432                 tp->t_waitrq = curtime;
1433         } else {
1434                 (void) cpu_update_pct(tp, gethrtime_unscaled());
```



```
1435         }

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

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

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

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

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

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

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

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

unchanged_portion_omitted_

```
1573 /*
1574  * Remove a thread from the dispatcher queue if it is on it.
1575  * It is not an error if it is not found but we return whether
1576  * or not it was found in case the caller wants to check.
1577  */
```

```
1578 int
1579 dispdeq(kthread_t *tp)
1580 {
1581         disp_t          *dp;
1582         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_PRIO(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_sruninc 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_sruninc(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 maxclsyspri.
1770                  */
1771                 if (DISP_PRIO(tp) > maxclsyspri)
1772                         wake_sched = 1;
1773                 else
1774                         wake_sched_sec = 1;
1775                 THREAD_RUN(tp, &swapped_lock); /* set TS_RUN state and lock */
1776                 break;
1777         case TS_RUN:                            /* called from ts_update */
1778                 break;
1779         default:
1780                 panic("disp_swapped_setrun: tp: %p bad t_state", (void *)tp);
1781         }
1782 }

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



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

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

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

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

1692         DTRACE_SCHED1(surrender, kthread_t *, tp);

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

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

2015         ASSERT(THREAD_LOCK_HELD(tp));

2017         /*
2018          * Don't do anything if the thread is not bound, or
2019          * currently not runnable.
2157          * currently not runnable or swapped out.
2020          */
2021         if (tp->t_bound_cpu == NULL ||
2022             tp->t_state != TS_RUN)
2160             tp->t_state != TS_RUN ||
```

```
2161             tp->t_schedflag & TS_ON_SWAPQ)
2023                 return;

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

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

2052         disp_lock_enter(&dp->disp_lock);

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

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


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

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

2086                 /*
2087                  * We've got some unbound threads on this queue, so turn
```



```
2088                  * the allbound flag off now.
2089                  */
2090                 allbound = B_FALSE;

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

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

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

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

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

```
2154                          break;

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

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

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

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

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

2199  #ifdef DEBUG
2200          {
2201                  int     thread_was_on_queue;

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

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

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

2356          tp->t_schedflag |= TS_DONT_SWAP;

2217          /*
```



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

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

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

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

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

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

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

2258          return (tp);
2259  }
```

*unchanged_portion_omitted_*

```
**********************************************************
   66902 Fri Mar 28 23:33:15 2014
new/usr/src/uts/common/disp/fss.c
patch delete-t_stime
patch remove-swapenq-flag
patch remove-dont-swap-flag
patch remove-swapinout-class-ops
**********************************************************
______unchanged_portion_omitted_

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

 152 /*
 153  * Decay rate percentages are based on n/128 rather than n/100 so  that
 154  * calculations can avoid having to do an integer divide by 100 (divide
 155  * by FSS_DECAY_BASE == 128 optimizes to an arithmetic shift).
 156  *
 157  * FSS_DECAY_MIN        =  83/128 ~= 65%
 158  * FSS_DECAY_MAX        = 108/128 ~= 85%
 159  * FSS_DECAY_USG        =  96/128 ~= 75%
 160  */
 161 #define FSS_DECAY_MIN   83      /* fsspri decay pct for threads w/ nice -20 */
 162 #define FSS_DECAY_MAX   108     /* fsspri decay pct for threads w/ nice +19 */
 163 #define FSS_DECAY_USG   96      /* fssusage decay pct for projects */
 164 #define FSS_DECAY_BASE  128     /* base for decay percentages above */

 166 #define FSS_NICE_MIN    0
 167 #define FSS_NICE_MAX    (2 * NZERO - 1)
 168 #define FSS_NICE_RANGE  (FSS_NICE_MAX - FSS_NICE_MIN + 1)

 170 static int      fss_nice_tick[FSS_NICE_RANGE];
 171 static int      fss_nice_decay[FSS_NICE_RANGE];

 173 static pri_t    fss_maxupri = FSS_MAXUPRI; /* maximum FSS user priority */
 174 static pri_t    fss_maxumdpri; /* maximum user mode fss priority */
 175 static pri_t    fss_maxglobpri; /* maximum global priority used by fss class */
 176 static pri_t    fss_minglobpri; /* minimum global priority */

 178 static fssproc_t fss_listhead[FSS_LISTS];
 179 static kmutex_t fss_listlock[FSS_LISTS];

 181 static fsspset_t *fsspsets;
 182 static kmutex_t fsspsets_lock;  /* protects fsspsets */

 184 static id_t     fss_cid;

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

 190 static void     fss_newpri(fssproc_t *);
 191 static void     fss_update(void *);
 192 static int      fss_update_list(int);
 193 static void     fss_change_priority(kthread_t *, fssproc_t *);

 195 static int      fss_admin(caddr_t, cred_t *);
 196 static int      fss_getclinfo(void *);
 197 static int      fss_parmsin(void *);
 198 static int      fss_parmsout(void *, pc_vaparms_t *);
 199 static int      fss_vaparmsin(void *, pc_vaparms_t *);
 200 static int      fss_vaparmsout(void *, pc_vaparms_t *);
 201 static int      fss_getclpri(pcpri_t *);
 202 static int      fss_alloc(void **, int);
 203 static void     fss_free(void *);

 205 static int      fss_enterclass(kthread_t *, id_t, void *, cred_t *, void *);
```



```
 206 static void     fss_exitclass(void *);
 207 static int      fss_canexit(kthread_t *, cred_t *);
 208 static int      fss_fork(kthread_t *, kthread_t *, void *);
 209 static void     fss_forkret(kthread_t *, kthread_t *);
 210 static void     fss_parmsget(kthread_t *, void *);
 211 static int      fss_parmsset(kthread_t *, void *, id_t, cred_t *);
 212 static void     fss_stop(kthread_t *, int, int);
 213 static void     fss_exit(kthread_t *);
 214 static void     fss_active(kthread_t *);
 215 static void     fss_inactive(kthread_t *);
 216 static pri_t    fss_swapin(kthread_t *, int);
 217 static pri_t    fss_swapout(kthread_t *, int);
 216 static void     fss_trapret(kthread_t *);
 217 static void     fss_preempt(kthread_t *);
 218 static void     fss_setrun(kthread_t *);
 219 static void     fss_sleep(kthread_t *);
 220 static void     fss_tick(kthread_t *);
 221 static void     fss_wakeup(kthread_t *);
 222 static int      fss_donice(kthread_t *, cred_t *, int, int *);
 223 static int      fss_doprio(kthread_t *, cred_t *, int, int *);
 224 static pri_t    fss_globpri(kthread_t *);
 225 static void     fss_yield(kthread_t *);
 226 static void     fss_nullsys();

 228 static struct classfuncs fss_classfuncs = {
 229         /* class functions */
 230         fss_admin,
 231         fss_getclinfo,
 232         fss_parmsin,
 233         fss_parmsout,
 234         fss_vaparmsin,
 235         fss_vaparmsout,
 236         fss_getclpri,
 237         fss_alloc,
 238         fss_free,

 240         /* thread functions */
 241         fss_enterclass,
 242         fss_exitclass,
 243         fss_canexit,
 244         fss_fork,
 245         fss_forkret,
 246         fss_parmsget,
 247         fss_parmsset,
 248         fss_stop,
 249         fss_exit,
 250         fss_active,
 251         fss_inactive,
 254         fss_swapin,
 255         fss_swapout,
 252         fss_trapret,
 253         fss_preempt,
 254         fss_setrun,
 255         fss_sleep,
 256         fss_tick,
 257         fss_wakeup,
 258         fss_donice,
 259         fss_globpri,
 260         fss_nullsys,    /* set_process_group */
 261         fss_yield,
 262         fss_doprio,
 263 };
______unchanged_portion_omitted_

1827 /*
1832  * fss_swapin() returns -1 if the thread is loaded or is not eligible to be
```

```
1833  * swapped in. Otherwise, it returns the thread's effective priority based
1834  * on swapout time and size of process (0 <= epri <= 0 SHRT_MAX).
1835  */
1836 /*ARGSUSED*/
1837 static pri_t
1838 fss_swapin(kthread_t *t, int flags)
1839 {
1840         fssproc_t *fssproc = FSSPROC(t);
1841         long epri = -1;
1842         proc_t *pp = ttoproc(t);

1844         ASSERT(THREAD_LOCK_HELD(t));

1846         if (t->t_state == TS_RUN && (t->t_schedflag & TS_LOAD) == 0) {
1847                 time_t swapout_time;

1849                 swapout_time = (ddi_get_lbolt() - t->t_stime) / hz;
1850                 if (INHERITED(t) || (fssproc->fss_flags & FSSKPRI)) {
1851                         epri = (long)DISP_PRIO(t) + swapout_time;
1852                 } else {
1853                         /*
1854                          * Threads which have been out for a long time,
1855                          * have high user mode priority and are associated
1856                          * with a small address space are more deserving.
1857                          */
1858                         epri = fssproc->fss_umdpri;
1859                         ASSERT(epri >= 0 && epri <= fss_maxumdpri);
1860                         epri += swapout_time - pp->p_swrss / nz(maxpgio)/2;
1861                 }
1862                 /*
1863                  * Scale epri so that SHRT_MAX / 2 represents zero priority.
1864                  */
1865                 epri += SHRT_MAX / 2;
1866                 if (epri < 0)
1867                         epri = 0;
1868                 else if (epri > SHRT_MAX)
1869                         epri = SHRT_MAX;
1870         }
1871         return ((pri_t)epri);
1872 }

1874 /*
1875  * fss_swapout() returns -1 if the thread isn't loaded or is not eligible to
1876  * be swapped out. Otherwise, it returns the thread's effective priority
1877  * based on if the swapper is in softswap or hardswap mode.
1878  */
1879 static pri_t
1880 fss_swapout(kthread_t *t, int flags)
1881 {
1882         fssproc_t *fssproc = FSSPROC(t);
1883         long epri = -1;
1884         proc_t *pp = ttoproc(t);
1885         time_t swapin_time;

1887         ASSERT(THREAD_LOCK_HELD(t));

1889         if (INHERITED(t) ||
1890             (fssproc->fss_flags & FSSKPRI) ||
1891             (t->t_proc_flag & TP_LWPEXIT) ||
1892             (t->t_state & (TS_ZOMB|TS_FREE|TS_STOPPED|TS_ONPROC|TS_WAIT)) ||
1893             !(t->t_schedflag & TS_LOAD) ||
1894             !(SWAP_OK(t)))
1895                 return (-1);

1897         ASSERT(t->t_state & (TS_SLEEP | TS_RUN));
```



```
1899         swapin_time = (ddi_get_lbolt() - t->t_stime) / hz;

1901         if (flags == SOFTSWAP) {
1902                 if (t->t_state == TS_SLEEP && swapin_time > maxslp) {
1903                         epri = 0;
1904                 } else {
1905                         return ((pri_t)epri);
1906                 }
1907         } else {
1908                 pri_t pri;

1910                 if ((t->t_state == TS_SLEEP && swapin_time > fss_minslp) ||
1911                     (t->t_state == TS_RUN && swapin_time > fss_minrun)) {
1912                         pri = fss_maxumdpri;
1913                         epri = swapin_time -
1914                             (rm_asrss(pp->p_as) / nz(maxpgio)/2) - (long)pri;
1915                 } else {
1916                         return ((pri_t)epri);
1917                 }
1918         }

1920         /*
1921          * Scale epri so that SHRT_MAX / 2 represents zero priority.
1922          */
1923         epri += SHRT_MAX / 2;
1924         if (epri < 0)
1925                 epri = 0;
1926         else if (epri > SHRT_MAX)
1927                 epri = SHRT_MAX;

1929         return ((pri_t)epri);
1930 }

1932 /*
1828  * If thread is currently at a kernel mode priority (has slept) and is
1829  * returning to the userland we assign it the appropriate user mode priority
1830  * and time quantum here.  If we're lowering the thread's priority below that
1831  * of other runnable threads then we will set runrun via cpu_surrender() to
1832  * cause preemption.
1833  */
1834 static void
1835 fss_trapret(kthread_t *t)
1836 {
1837         fssproc_t *fssproc = FSSPROC(t);
1838         cpu_t *cp = CPU;

1840         ASSERT(THREAD_LOCK_HELD(t));
1841         ASSERT(t == curthread);
1842         ASSERT(cp->cpu_dispthread == t);
1843         ASSERT(t->t_state == TS_ONPROC);

1845         t->t_kpri_req = 0;
1846         if (fssproc->fss_flags & FSSKPRI) {
1847                 /*
1848                  * If thread has blocked in the kernel
1849                  */
1850                 THREAD_CHANGE_PRI(t, fssproc->fss_umdpri);
1851                 cp->cpu_dispatch_pri = DISP_PRIO(t);
1852                 ASSERT(t->t_pri >= 0 && t->t_pri <= fss_maxglobpri);
1853                 fssproc->fss_flags &= ~FSSKPRI;

1855                 if (DISP_MUST_SURRENDER(t))
1856                         cpu_surrender(t);
1857         }

1964         /*
```

```
1965          * Swapout lwp if the swapper is waiting for this thread to reach
1966          * a safe point.
1967          */
1968         if (t->t_schedflag & TS_SWAPENQ) {
1969                 thread_unlock(t);
1970                 swapout_lwp(ttolwp(t));
1971                 thread_lock(t);
1972         }
1858 }

1860 /*
1861  * Arrange for thread to be placed in appropriate location on dispatcher queue.
1862  * This is called with the current thread in TS_ONPROC and locked.
1863  */
1864 static void
1865 fss_preempt(kthread_t *t)
1866 {
1867         fssproc_t *fssproc = FSSPROC(t);
1868         klwp_t *lwp;
1869         uint_t flags;

1871         ASSERT(t == curthread);
1872         ASSERT(THREAD_LOCK_HELD(curthread));
1873         ASSERT(t->t_state == TS_ONPROC);

1875         /*
1876          * If preempted in the kernel, make sure the thread has a kernel
1877          * priority if needed.
1878          */
1879         lwp = curthread->t_lwp;
1880         if (!(fssproc->fss_flags & FSSKPRI) && lwp != NULL && t->t_kpri_req) {
1881                 fssproc->fss_flags |= FSSKPRI;
1882                 THREAD_CHANGE_PRI(t, minclsyspri);
1883                 ASSERT(t->t_pri >= 0 && t->t_pri <= fss_maxglobpri);
1884                 t->t_trapret = 1;       /* so that fss_trapret will run */
1885                 aston(t);
1886         }

1888         /*
1889          * This thread may be placed on wait queue by CPU Caps. In this case we
1890          * do not need to do anything until it is removed from the wait queue.
1891          * Do not enforce CPU caps on threads running at a kernel priority
1892          */
1893         if (CPUCAPS_ON()) {
1894                 (void) cpucaps_charge(t, &fssproc->fss_caps,
1895                     CPUCAPS_CHARGE_ENFORCE);

1897                 if (!(fssproc->fss_flags & FSSKPRI) && CPUCAPS_ENFORCE(t))
1898                         return;
1899         }

1901         /*
2017          * If preempted in user-land mark the thread as swappable because it
2018          * cannot be holding any kernel locks.
2019          */
2020         ASSERT(t->t_schedflag & TS_DONT_SWAP);
2021         if (lwp != NULL && lwp->lwp_state == LWP_USER)
2022                 t->t_schedflag &= ~TS_DONT_SWAP;

2024         /*
1902          * Check to see if we're doing "preemption control" here.  If
1903          * we are, and if the user has requested that this thread not
1904          * be preempted, and if preemptions haven't been put off for
1905          * too long, let the preemption happen here but try to make
1906          * sure the thread is rescheduled as soon as possible.  We do
1907          * this by putting it on the front of the highest priority run
```



```
1908          * queue in the FSS class.  If the preemption has been put off
1909          * for too long, clear the "nopreempt" bit and let the thread
1910          * be preempted.
1911          */
1912         if (t->t_schedctl && schedctl_get_nopreempt(t)) {
1913                 if (fssproc->fss_timeleft > -SC_MAX_TICKS) {
1914                         DTRACE_SCHED1(schedctl__nopreempt, kthread_t *, t);
1915                         if (!(fssproc->fss_flags & FSSKPRI)) {
1916                                 /*
1917                                  * If not already remembered, remember current
1918                                  * priority for restoration in fss_yield().
1919                                  */
1920                                 if (!(fssproc->fss_flags & FSSRESTORE)) {
1921                                         fssproc->fss_scpri = t->t_pri;
1922                                         fssproc->fss_flags |= FSSRESTORE;
1923                                 }
1924                                 THREAD_CHANGE_PRI(t, fss_maxumdpri);
2048                                 t->t_schedflag |= TS_DONT_SWAP;
1925                         }
1926                         schedctl_set_yield(t, 1);
1927                         setfrontdq(t);
1928                         return;
1929                 } else {
1930                         if (fssproc->fss_flags & FSSRESTORE) {
1931                                 THREAD_CHANGE_PRI(t, fssproc->fss_scpri);
1932                                 fssproc->fss_flags &= ~FSSRESTORE;
1933                         }
1934                         schedctl_set_nopreempt(t, 0);
1935                         DTRACE_SCHED1(schedctl__preempt, kthread_t *, t);
1936                         /*
1937                          * Fall through and be preempted below.
1938                          */
1939                 }
1940         }

1942         flags = fssproc->fss_flags & (FSSBACKQ | FSSKPRI);

1944         if (flags == FSSBACKQ) {
1945                 fssproc->fss_timeleft = fss_quantum;
1946                 fssproc->fss_flags &= ~FSSBACKQ;
1947                 setbackdq(t);
1948         } else if (flags == (FSSBACKQ | FSSKPRI)) {
1949                 fssproc->fss_flags &= ~FSSBACKQ;
1950                 setbackdq(t);
1951         } else {
1952                 setfrontdq(t);
1953         }
1954 }
______unchanged_portion_omitted_

1985 /*
1986  * Prepare thread for sleep. We reset the thread priority so it will run at the
1987  * kernel priority level when it wakes up.
1988  */
1989 static void
1990 fss_sleep(kthread_t *t)
1991 {
1992         fssproc_t *fssproc = FSSPROC(t);

1994         ASSERT(t == curthread);
1995         ASSERT(THREAD_LOCK_HELD(t));

1997         ASSERT(t->t_state == TS_ONPROC);

1999         /*
2000          * Account for time spent on CPU before going to sleep.
```

```
2001          */
2002         (void) CPUCAPS_CHARGE(t, &fssproc->fss_caps, CPUCAPS_CHARGE_ENFORCE);

2004         fss_inactive(t);

2006         /*
2007          * Assign a system priority to the thread and arrange for it to be
2008          * retained when the thread is next placed on the run queue (i.e.,
2009          * when it wakes up) instead of being given a new pri.  Also arrange
2010          * for trapret processing as the thread leaves the system call so it
2011          * will drop back to normal priority range.
2012          */
2013         if (t->t_kpri_req) {
2014                 THREAD_CHANGE_PRI(t, minclsyspri);
2015                 fssproc->fss_flags |= FSSKPRI;
2016                 t->t_trapret = 1;       /* so that fss_trapret will run */
2017                 aston(t);
2018         } else if (fssproc->fss_flags & FSSKPRI) {
2019                 /*
2020                  * The thread has done a THREAD_KPRI_REQUEST(), slept, then
2021                  * done THREAD_KPRI_RELEASE() (so no t_kpri_req is 0 again),
2022                  * then slept again all without finishing the current system
2023                  * call so trapret won't have cleared FSSKPRI
2024                  */
2025                 fssproc->fss_flags &= ~FSSKPRI;
2026                 THREAD_CHANGE_PRI(t, fssproc->fss_umdpri);
2027                 if (DISP_MUST_SURRENDER(curthread))
2028                         cpu_surrender(t);
2029         }
2154         t->t_stime = ddi_get_lbolt();   /* time stamp for the swapper */
2030 }

2032 /*
2033  * A tick interrupt has ocurrend on a running thread. Check to see if our
2034  * time slice has expired.
2159  * time slice has expired.  We must also clear the TS_DONT_SWAP flag in
2160  * t_schedflag if the thread is eligible to be swapped out.
2035  */
2036 static void
2037 fss_tick(kthread_t *t)
2038 {
2039         fssproc_t *fssproc;
2040         fssproj_t *fssproj;
2041         klwp_t *lwp;
2042         boolean_t call_cpu_surrender = B_FALSE;
2043         boolean_t cpucaps_enforce = B_FALSE;

2045         ASSERT(MUTEX_HELD(&(ttoproc(t))->p_lock));

2047         /*
2048          * It's safe to access fsspset and fssproj structures because we're
2049          * holding our p_lock here.
2050          */
2051         thread_lock(t);
2052         fssproc = FSSPROC(t);
2053         fssproj = FSSPROC2FSSPROJ(fssproc);
2054         if (fssproj != NULL) {
2055                 fsspset_t *fsspset = FSSPROJ2FSSPSET(fssproj);
2056                 disp_lock_enter_high(&fsspset->fssps_displock);
2057                 fssproj->fssp_ticks += fss_nice_tick[fssproc->fss_nice];
2058                 fssproc->fss_ticks++;
2059                 disp_lock_exit_high(&fsspset->fssps_displock);
2060         }

2062         /*
2063          * Keep track of thread's project CPU usage.  Note that projects
```



```
2064          * get charged even when threads are running in the kernel.
2065          * Do not surrender CPU if running in the SYS class.
2066          */
2067         if (CPUCAPS_ON()) {
2068                 cpucaps_enforce = cpucaps_charge(t,
2069                     &fssproc->fss_caps, CPUCAPS_CHARGE_ENFORCE) &&
2070                     !(fssproc->fss_flags & FSSKPRI);
2071         }

2073         /*
2074          * A thread's execution time for threads running in the SYS class
2075          * is not tracked.
2076          */
2077         if ((fssproc->fss_flags & FSSKPRI) == 0) {
2078                 /*
2079                  * If thread is not in kernel mode, decrement its fss_timeleft
2080                  */
2081                 if (--fssproc->fss_timeleft <= 0) {
2082                         pri_t new_pri;

2084                         /*
2085                          * If we're doing preemption control and trying to
2086                          * avoid preempting this thread, just note that the
2087                          * thread should yield soon and let it keep running
2088                          * (unless it's been a while).
2089                          */
2090                         if (t->t_schedctl && schedctl_get_nopreempt(t)) {
2091                                 if (fssproc->fss_timeleft > -SC_MAX_TICKS) {
2092                                         DTRACE_SCHED1(schedctl__nopreempt,
2093                                             kthread_t *, t);
2094                                         schedctl_set_yield(t, 1);
2095                                         thread_unlock_nopreempt(t);
2096                                         return;
2097                                 }
2098                         }
2099                         fssproc->fss_flags &= ~FSSRESTORE;

2101                         fss_newpri(fssproc);
2102                         new_pri = fssproc->fss_umdpri;
2103                         ASSERT(new_pri >= 0 && new_pri <= fss_maxglobpri);

2105                         /*
2106                          * When the priority of a thread is changed, it may
2107                          * be necessary to adjust its position on a sleep queue
2108                          * or dispatch queue. The function thread_change_pri
2109                          * accomplishes this.
2110                          */
2111                         if (thread_change_pri(t, new_pri, 0)) {
2238                                 if ((t->t_schedflag & TS_LOAD) &&
2239                                     (lwp = t->t_lwp) &&
2240                                     lwp->lwp_state == LWP_USER)
2241                                         t->t_schedflag &= ~TS_DONT_SWAP;
2112                                 fssproc->fss_timeleft = fss_quantum;
2113                         } else {
2114                                 call_cpu_surrender = B_TRUE;
2115                         }
2116                 } else if (t->t_state == TS_ONPROC &&
2117                     t->t_pri < t->t_disp_queue->disp_maxrunpri) {
2118                         /*
2119                          * If there is a higher-priority thread which is
2120                          * waiting for a processor, then thread surrenders
2121                          * the processor.
2122                          */
2123                         call_cpu_surrender = B_TRUE;
2124                 }
2125         }
```

```
2127         if (cpucaps_enforce && 2 * fssproc->fss_timeleft > fss_quantum) {
2128                 /*
2129                  * The thread used more than half of its quantum, so assume that
2130                  * it used the whole quantum.
2131                  *
2132                  * Update thread's priority just before putting it on the wait
2133                  * queue so that it gets charged for the CPU time from its
2134                  * quantum even before that quantum expires.
2135                  */
2136                 fss_newpri(fssproc);
2137                 if (t->t_pri != fssproc->fss_umdpri)
2138                         fss_change_priority(t, fssproc);

2140                 /*
2141                  * We need to call cpu_surrender for this thread due to cpucaps
2142                  * enforcement, but fss_change_priority may have already done
2143                  * so. In this case FSSBACKQ is set and there is no need to call
2144                  * cpu-surrender again.
2145                  */
2146                 if (!(fssproc->fss_flags & FSSBACKQ))
2147                         call_cpu_surrender = B_TRUE;
2148         }

2150         if (call_cpu_surrender) {
2151                 fssproc->fss_flags |= FSSBACKQ;
2152                 cpu_surrender(t);
2153         }

2155         thread_unlock_nopreempt(t);     /* clock thread can't be preempted */
2156 }

2158 /*
2159  * Processes waking up go to the back of their queue.  We don't need to assign
2160  * a time quantum here because thread is still at a kernel mode priority and
2161  * the time slicing is not done for threads running in the kernel after
2162  * sleeping.  The proper time quantum will be assigned by fss_trapret before the
2163  * thread returns to user mode.
2164  */
2165 static void
2166 fss_wakeup(kthread_t *t)
2167 {
2168         fssproc_t *fssproc;

2170         ASSERT(THREAD_LOCK_HELD(t));
2171         ASSERT(t->t_state == TS_SLEEP);

2173         fss_active(t);

2305         t->t_stime = ddi_get_lbolt();           /* time stamp for the swapper */
2175         fssproc = FSSPROC(t);
2176         fssproc->fss_flags &= ~FSSBACKQ;

2178         if (fssproc->fss_flags & FSSKPRI) {
2179                 /*
2180                  * If we already have a kernel priority assigned, then we
2181                  * just use it.
2182                  */
2183                 setbackdq(t);
2184         } else if (t->t_kpri_req) {
2185                 /*
2186                  * Give thread a priority boost if we were asked.
2187                  */
2188                 fssproc->fss_flags |= FSSKPRI;
2189                 THREAD_CHANGE_PRI(t, minclsyspri);
2190                 setbackdq(t);
```



```
2191                 t->t_trapret = 1;       /* so that fss_trapret will run */
2192                 aston(t);
2193         } else {
2194                 /*
2195                  * Otherwise, we recalculate the priority.
2196                  */
2197                 if (t->t_disp_time == ddi_get_lbolt()) {
2198                         setfrontdq(t);
2199                 } else {
2200                         fssproc->fss_timeleft = fss_quantum;
2201                         THREAD_CHANGE_PRI(t, fssproc->fss_umdpri);
2202                         setbackdq(t);
2203                 }
2204         }
2205 }
```

unchanged_portion_omitted_

```
**********************************************************
   42975 Fri Mar 28 23:33:18 2014
new/usr/src/uts/common/disp/fx.c
patch delete-t_stime
patch remove-swapinout-class-ops
**********************************************************
______unchanged_portion_omitted_

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


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

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

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


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

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

 196 extern fxdpent_t *fx_getdptbl(void);

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



```
 200 static void fx_list_release(fxproc_t *);


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

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


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

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

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

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

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

1249         ASSERT(THREAD_LOCK_HELD(t));

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

1255         return (tpri);
1256 }

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

1268         return (-1);

1223 }
```

_____unchanged_portion_omitted_

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

1350         ASSERT(THREAD_LOCK_HELD(t));

1399         t->t_stime = ddi_get_lbolt();           /* time stamp for the swapper */
1352         if (FX_HAS_CB(fxpp)) {
1353                 clock_t new_quantum =  (clock_t)fxpp->fx_pquantum;
1354                 pri_t   newpri = fxpp->fx_pri;
1355                 FX_CB_WAKEUP(FX_CALLB(fxpp), fxpp->fx_cookie,
1356                     &new_quantum, &newpri);
1357                 FX_ADJUST_QUANTUM(new_quantum);
1358                 if ((int)new_quantum != fxpp->fx_pquantum) {
1359                         fxpp->fx_pquantum = (int)new_quantum;
1360                         fxpp->fx_timeleft = fxpp->fx_pquantum;
1361                 }
```



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

1370         fxpp->fx_flags &= ~FXBACKQ;

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

_____unchanged_portion_omitted_

```
**********************************************************
   25930 Fri Mar 28 23:33:20 2014
new/usr/src/uts/common/disp/rt.c
patch remove-dont-swap-flag
patch remove-swapinout-class-ops
**********************************************************
______unchanged_portion_omitted_


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

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

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

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

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

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

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

 146 extern rtdpent_t *rt_getdptbl(void);

 148 static struct classfuncs rt_classfuncs = {
```



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


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

 902         ASSERT(THREAD_LOCK_HELD(t));

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

 911 }
______unchanged_portion_omitted_
```

```
 935 /*
 947  * Returns the priority of the thread, -1 if the thread is loaded or ineligible
 948  * for swapin.
 949  *
 950  * FX and RT threads are designed so that they don't swapout; however, it
 951  * is possible that while the thread is swapped out and in another class, it
 952  * can be changed to FX or RT.  Since these threads should be swapped in as
 953  * soon as they're runnable, rt_swapin returns SHRT_MAX, and fx_swapin
 954  * returns SHRT_MAX - 1, so that it gives deference to any swapped out RT
 955  * threads.
 956  */
 957 /* ARGSUSED */
 958 static pri_t
 959 rt_swapin(kthread_t *t, int flags)
 960 {
 961         pri_t   tpri = -1;

 963         ASSERT(THREAD_LOCK_HELD(t));

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

 969         return (tpri);
 970 }

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

 981         return (-1);
 982 }

 984 /*
 936  * Check for time slice expiration (unless thread has infinite time
 937  * slice).  If time slice has expired arrange for thread to be preempted
 938  * and placed on back of queue.
 939  */
 940 static void
 941 rt_tick(kthread_t *t)
 942 {
 943         rtproc_t *rtpp = (rtproc_t *)(t->t_cldata);

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

 947         thread_lock(t);
 948         if ((rtpp->rt_pquantum != RT_TQINF && --rtpp->rt_timeleft == 0) ||
 949             (t->t_state == TS_ONPROC && DISP_MUST_SURRENDER(t))) {
 950                 if (rtpp->rt_timeleft == 0 && rtpp->rt_tqsignal) {
 951                         thread_unlock(t);
 952                         sigtoproc(ttoproc(t), t, rtpp->rt_tqsignal);
 953                         thread_lock(t);
 954                 }
 955                 rtpp->rt_flags |= RTBACKQ;
 956                 cpu_surrender(t);
 957         }
 958         thread_unlock(t);
 959 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    4863 Fri Mar 28 23:33:21 2014
new/usr/src/uts/common/disp/sysclass.c
patch fix-compile
**********************************************************
```

```
   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 2008 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 #pragma ident   "%Z%%M% %I%     %E% SMI"        /* from SVr4.0 1.12 */

  32 #include <sys/types.h>
  33 #include <sys/param.h>
  34 #include <sys/sysmacros.h>
  35 #include <sys/signal.h>
  36 #include <sys/pcb.h>
  37 #include <sys/user.h>
  38 #include <sys/systm.h>
  39 #include <sys/sysinfo.h>
  40 #include <sys/var.h>
  41 #include <sys/errno.h>
  42 #include <sys/cmn_err.h>
  43 #include <sys/proc.h>
  44 #include <sys/debug.h>
  45 #include <sys/inline.h>
  46 #include <sys/disp.h>
  47 #include <sys/class.h>
  48 #include <sys/kmem.h>
  49 #include <sys/cpuvar.h>
  50 #include <sys/priocntl.h>

  52 /*
  53  * Class specific code for the sys class. There are no
  54  * class specific data structures associated with
  55  * the sys class and the scheduling policy is trivially
  56  * simple. There is no time slicing.
  57  */

  59 pri_t           sys_init(id_t, int, classfuncs_t **);
  60 static int      sys_getclpri(pcpri_t *);
  61 static int      sys_fork(kthread_t *, kthread_t *, void *);
```



```
  62 static int      sys_enterclass(kthread_t *, id_t, void *, cred_t *, void *);
  63 static int      sys_canexit(kthread_t *, cred_t *);
  64 static int      sys_nosys();
  65 static int      sys_donice(kthread_t *, cred_t *, int, int *);
  66 static int      sys_doprio(kthread_t *, cred_t *, int, int *);
  67 static void     sys_forkret(kthread_t *, kthread_t *);
  68 static void     sys_nullsys();
  69 static pri_t    sys_swappri(kthread_t *, int);
  69 static int      sys_alloc(void **, int);

  71 struct classfuncs sys_classfuncs = {
  72         /* messages to class manager */
  73         {
  74                 sys_nosys,      /* admin */
  75                 sys_nosys,      /* getclinfo */
  76                 sys_nosys,      /* parmsin */
  77                 sys_nosys,      /* parmsout */
  78                 sys_nosys,      /* vaparmsin */
  79                 sys_nosys,      /* vaparmsout */
  80                 sys_getclpri,   /* getclpri */
  81                 sys_alloc,
  82                 sys_nullsys,    /* free */
  83         },
  84         /* operations on threads */
  85         {
  86                 sys_enterclass, /* enterclass */
  87                 sys_nullsys,    /* exitclass */
  88                 sys_canexit,
  89                 sys_fork,
  90                 sys_forkret,    /* forkret */
  91                 sys_nullsys,    /* parmsget */
  92                 sys_nosys,      /* parmsset */
  93                 sys_nullsys,    /* stop */
  94                 sys_nullsys,    /* exit */
  95                 sys_nullsys,    /* active */
  96                 sys_nullsys,    /* inactive */
  98                 sys_swappri,    /* swapin */
  99                 sys_swappri,    /* swapout */
  97                 sys_nullsys,    /* trapret */
  98                 setfrontdq,     /* preempt */
  99                 setbackdq,      /* setrun */
 100                 sys_nullsys,    /* sleep */
 101                 sys_nullsys,    /* tick */
 102                 setbackdq,      /* wakeup */
 103                 sys_donice,
 104                 (pri_t (*)())sys_nosys, /* globpri */
 105                 sys_nullsys,    /* set_process_group */
 106                 sys_nullsys,    /* yield */
 107                 sys_doprio,
 108         }

 110 };
_____unchanged_portion_omitted_

 197 /* ARGSUSED */
 198 static pri_t
 199 sys_swappri(t, flags)
 200         kthread_t       *t;
 201         int             flags;
 202 {
 203         return (-1);
 204 }

 194 static int
 195 sys_nosys()
 196 {
```

```
 197         return (ENOSYS);
 198 }
```

unchanged_portion_omitted_

```
**********************************************************
   37694 Fri Mar 28 23:33:22 2014
new/usr/src/uts/common/disp/sysdc.c
patch remove-swapinout-class-ops
**********************************************************
______unchanged_portion_omitted_

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

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

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

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



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

```
**********************************************************
   53368 Fri Mar 28 23:33:24 2014
new/usr/src/uts/common/disp/thread.c
patch delete-t_stime
patch remove-load-flag
patch remove-dont-swap-flag
**********************************************************
______unchanged_portion_omitted_

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

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

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

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

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

 363                 ASSERT(stk != NULL);

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



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

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

 416         t->t_ts = ts;

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

```
435         t->t_bind_pset = PS_NONE;
436         t->t_plockp = &pp->p_lock;
437         t->t_copyops = NULL;
438         t->t_taskq = NULL;
439         t->t_anttime = 0;
440         t->t_hatdepth = 0;

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

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

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

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

467         lgrp_affinity_init(&t->t_lgrp_affinity);

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

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

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

500         /*
```



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

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

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

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

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

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

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

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

_____***unchanged_portion_omitted_***

```
**********************************************************
   57797 Fri Mar 28 23:33:25 2014
new/usr/src/uts/common/disp/ts.c
patch delete-t_stime
patch remove-swapenq-flag
patch remove-dont-swap-flag
patch remove-swapinout-class-ops
**********************************************************
______unchanged_portion_omitted_


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

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

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

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

 236 static gid_t    IA_gid = 0;
```



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

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

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

 293         /* thread functions */
 294         ts_enterclass,
 295         ts_exitclass,
 296         ts_canexit,
 297         ts_fork,
 298         ts_forkret,
 299         ia_parmsget,
 300         ia_parmsset,
```

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

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

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

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

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

1400         /*
1407          * If thread got preempted in the user-land then we know
1408          * it isn't holding any locks.  Mark it as swappable.
1409          */
1410         ASSERT(t->t_schedflag & TS_DONT_SWAP);
1411         if (lwp != NULL && lwp->lwp_state == LWP_USER)
```



```
1412                 t->t_schedflag &= ~TS_DONT_SWAP;

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

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

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

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

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

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

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

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

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

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


1571 /*
1572  * Return Values:
1573  *
1574  *      -1 if the thread is loaded or is not eligible to be swapped in.
1575  *
1576  *      effective priority of the specified thread based on swapout time
```



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

1587         ASSERT(THREAD_LOCK_HELD(t));

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

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

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

1641 static pri_t
1642 ts_swapout(kthread_t *t, int flags)
```

```
1643 {
1644         tsproc_t        *tspp = (tsproc_t *)(t->t_cldata);
1645         long            epri = -1;
1646         proc_t          *pp = ttoproc(t);
1647         time_t          swapin_time;

1649         ASSERT(THREAD_LOCK_HELD(t));

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

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

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

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

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

1694         return ((pri_t)epri);
1695 }

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



```
1567         ASSERT(MUTEX_HELD(&(ttoproc(t))->p_lock));

1569         thread_lock(t);

1571         /*
1572          * Keep track of thread's project CPU usage.  Note that projects
1573          * get charged even when threads are running in the kernel.
1574          */
1575         if (CPUCAPS_ON()) {
1576                 call_cpu_surrender = cpucaps_charge(t, &tspp->ts_caps,
1577                     CPUCAPS_CHARGE_ENFORCE) && !(tspp->ts_flags & TSKPRI);
1578         }

1580         if ((tspp->ts_flags & TSKPRI) == 0) {
1581                 if (--tspp->ts_timeleft <= 0) {
1582                         pri_t   new_pri;

1584                         /*
1585                          * If we're doing preemption control and trying to
1586                          * avoid preempting this thread, just note that
1587                          * the thread should yield soon and let it keep
1588                          * running (unless it's been a while).
1589                          */
1590                         if (t->t_schedctl && schedctl_get_nopreempt(t)) {
1591                                 if (tspp->ts_timeleft > -SC_MAX_TICKS) {
1592                                         DTRACE_SCHED1(schedctl__nopreempt,
1593                                             kthread_t *, t);
1594                                         schedctl_set_yield(t, 1);
1595                                         thread_unlock_nopreempt(t);
1596                                         return;
1597                                 }

1599                                 TNF_PROBE_2(schedctl_failsafe,
1600                                     "schedctl TS ts_tick", /* CSTYLED */,
1601                                     tnf_pid, pid, ttoproc(t)->p_pid,
1602                                     tnf_lwpid, lwpid, t->t_tid);
1603                         }
1604                         tspp->ts_flags &= ~TSRESTORE;
1605                         tspp->ts_cpupri = ts_dptbl[tspp->ts_cpupri].ts_tqexp;
1606                         TS_NEWUMDPRI(tspp);
1607                         tspp->ts_dispwait = 0;
1608                         new_pri = ts_dptbl[tspp->ts_umdpri].ts_globpri;
1609                         ASSERT(new_pri >= 0 && new_pri <= ts_maxglobpri);
1610                         /*
1611                          * When the priority of a thread is changed,
1612                          * it may be necessary to adjust its position
1613                          * on a sleep queue or dispatch queue.
1614                          * The function thread_change_pri accomplishes
1615                          * this.
1616                          */
1617                         if (thread_change_pri(t, new_pri, 0)) {
1761                                 if ((t->t_schedflag & TS_LOAD) &&
1762                                     (lwp = t->t_lwp) &&
1763                                     lwp->lwp_state == LWP_USER)
1764                                         t->t_schedflag &= ~TS_DONT_SWAP;
1618                                 tspp->ts_timeleft =
1619                                     ts_dptbl[tspp->ts_cpupri].ts_quantum;
1620                         } else {
1621                                 call_cpu_surrender = B_TRUE;
1622                         }
1623                         TRACE_2(TR_FAC_DISP, TR_TICK,
1624                             "tick:tid %p old pri %d", t, oldpri);
1625                 } else if (t->t_state == TS_ONPROC &&
1626                     t->t_pri < t->t_disp_queue->disp_maxrunpri) {
1627                         call_cpu_surrender = B_TRUE;
```

```
1628                 }
1629         }

1631         if (call_cpu_surrender) {
1632                 tspp->ts_flags |= TSBACKQ;
1633                 cpu_surrender(t);
1634         }

1636         thread_unlock_nopreempt(t);     /* clock thread can't be preempted */
1637 }


1640 /*
1641  * If thread is currently at a kernel mode priority (has slept)
1642  * we assign it the appropriate user mode priority and time quantum
1643  * here.  If we are lowering the thread's priority below that of
1644  * other runnable threads we will normally set runrun via cpu_surrender() to
1645  * cause preemption.
1646  */
1647 static void
1648 ts_trapret(kthread_t *t)
1649 {
1650         tsproc_t        *tspp = (tsproc_t *)t->t_cldata;
1651         cpu_t           *cp = CPU;
1652         pri_t           old_pri = curthread->t_pri;

1654         ASSERT(THREAD_LOCK_HELD(t));
1655         ASSERT(t == curthread);
1656         ASSERT(cp->cpu_dispthread == t);
1657         ASSERT(t->t_state == TS_ONPROC);

1659         t->t_kpri_req = 0;
1660         if (tspp->ts_dispwait > ts_dptbl[tspp->ts_umdpri].ts_maxwait) {
1661                 tspp->ts_cpupri = ts_dptbl[tspp->ts_cpupri].ts_slpret;
1662                 TS_NEWUMDPRI(tspp);
1663                 tspp->ts_timeleft = ts_dptbl[tspp->ts_cpupri].ts_quantum;
1664                 tspp->ts_dispwait = 0;

1666                 /*
1667                  * If thread has blocked in the kernel (as opposed to
1668                  * being merely preempted), recompute the user mode priority.
1669                  */
1670                 THREAD_CHANGE_PRI(t, ts_dptbl[tspp->ts_umdpri].ts_globpri);
1671                 cp->cpu_dispatch_pri = DISP_PRIO(t);
1672                 ASSERT(t->t_pri >= 0 && t->t_pri <= ts_maxglobpri);
1673                 tspp->ts_flags &= ~TSKPRI;

1675                 if (DISP_MUST_SURRENDER(t))
1676                         cpu_surrender(t);
1677         } else if (tspp->ts_flags & TSKPRI) {
1678                 /*
1679                  * If thread has blocked in the kernel (as opposed to
1680                  * being merely preempted), recompute the user mode priority.
1681                  */
1682                 THREAD_CHANGE_PRI(t, ts_dptbl[tspp->ts_umdpri].ts_globpri);
1683                 cp->cpu_dispatch_pri = DISP_PRIO(t);
1684                 ASSERT(t->t_pri >= 0 && t->t_pri <= ts_maxglobpri);
1685                 tspp->ts_flags &= ~TSKPRI;

1687                 if (DISP_MUST_SURRENDER(t))
1688                         cpu_surrender(t);
1689         }

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         }

1691         TRACE_2(TR_FAC_DISP, TR_TRAPRET,
1692             "trapret:tid %p old pri %d", t, old_pri);
1693 }
______unchanged_portion_omitted_

1813 /*
1814  * Processes waking up go to the back of their queue.  We don't
1815  * need to assign a time quantum here because thread is still
1816  * at a kernel mode priority and the time slicing is not done
1817  * for threads running in the kernel after sleeping.  The proper
1818  * time quantum will be assigned by ts_trapret before the thread
1819  * returns to user mode.
1820  */
1821 static void
1822 ts_wakeup(kthread_t *t)
1823 {
1824         tsproc_t        *tspp = (tsproc_t *)(t->t_cldata);

1826         ASSERT(THREAD_LOCK_HELD(t));

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

1828         if (tspp->ts_flags & TSKPRI) {
1829                 tspp->ts_flags &= ~TSBACKQ;
1830                 if (tspp->ts_flags & TSIASET)
1831                         setfrontdq(t);
1832                 else
1833                         setbackdq(t);
1834         } else if (t->t_kpri_req) {
1835                 /*
1836                  * Give thread a priority boost if we were asked.
1837                  */
1838                 tspp->ts_flags |= TSKPRI;
1839                 THREAD_CHANGE_PRI(t, ts_kmdpris[0]);
1840                 setbackdq(t);
1841                 t->t_trapret = 1;       /* so that ts_trapret will run */
1842                 aston(t);
1843         } else {
1844                 if (tspp->ts_dispwait > ts_dptbl[tspp->ts_umdpri].ts_maxwait) {
1845                         tspp->ts_cpupri = ts_dptbl[tspp->ts_cpupri].ts_slpret;
1846                         TS_NEWUMDPRI(tspp);
1847                         tspp->ts_timeleft =
1848                             ts_dptbl[tspp->ts_cpupri].ts_quantum;
1849                         tspp->ts_dispwait = 0;
1850                         THREAD_CHANGE_PRI(t,
1851                             ts_dptbl[tspp->ts_umdpri].ts_globpri);
1852                         ASSERT(t->t_pri >= 0 && t->t_pri <= ts_maxglobpri);
1853                 }

1855                 tspp->ts_flags &= ~TSBACKQ;

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

```
1865                            else
1866                                    setfrontdq(t);
1867                    }
1868            }
1869 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   67434 Fri Mar 28 23:33:25 2014
new/usr/src/uts/common/fs/nfs/nfs_srv.c
patch remove-dont-swap-flag
**********************************************************
______unchanged_portion_omitted_

1148 static struct rfs_async_write_list *rfs_async_write_head = NULL;
1149 static kmutex_t rfs_async_write_lock;
1150 static int rfs_write_async = 1; /* enables write clustering if == 1 */

1152 #define MAXCLIOVECS     42
1153 #define RFSWRITE_INITVAL (enum nfsstat) -1

1155 #ifdef DEBUG
1156 static int rfs_write_hits = 0;
1157 static int rfs_write_misses = 0;
1158 #endif

1160 /*
1161  * Write data to file.
1162  * Returns attributes of a file after writing some data to it.
1163  */
1164 void
1165 rfs_write(struct nfswriteargs *wa, struct nfsattrstat *ns,
1166         struct exportinfo *exi, struct svc_req *req, cred_t *cr)
1167 {
1168         int error;
1169         vnode_t *vp;
1170         rlim64_t rlimit;
1171         struct vattr va;
1172         struct uio uio;
1173         struct rfs_async_write_list *lp;
1174         struct rfs_async_write_list *nlp;
1175         struct rfs_async_write *rp;
1176         struct rfs_async_write *nrp;
1177         struct rfs_async_write *trp;
1178         struct rfs_async_write *lrp;
1179         int data_written;
1180         int iovcnt;
1181         mblk_t *m;
1182         struct iovec *iovp;
1183         struct iovec *niovp;
1184         struct iovec iov[MAXCLIOVECS];
1185         int count;
1186         int rcount;
1187         uint_t off;
1188         uint_t len;
1189         struct rfs_async_write nrpsp;
1190         struct rfs_async_write_list nlpsp;
1191         ushort_t t_flag;
1192         cred_t *savecred;
1193         int in_crit = 0;
1194         caller_context_t ct;

1196         if (!rfs_write_async) {
1197                 rfs_write_sync(wa, ns, exi, req, cr);
1198                 return;
1199         }

1201         /*
1202          * Initialize status to RFSWRITE_INITVAL instead of 0, since value of 0
1203          * is considered an OK.
1204          */
1205         ns->ns_status = RFSWRITE_INITVAL;
```



```
1207         nrp = &nrpsp;
1208         nrp->wa = wa;
1209         nrp->ns = ns;
1210         nrp->req = req;
1211         nrp->cr = cr;
1212         nrp->thread = curthread;

1214         ASSERT(curthread->t_schedflag & TS_DONT_SWAP);

1214         /*
1215          * Look to see if there is already a cluster started
1216          * for this file.
1217          */
1218         mutex_enter(&rfs_async_write_lock);
1219         for (lp = rfs_async_write_head; lp != NULL; lp = lp->next) {
1220                 if (bcmp(&wa->wa_fhandle, lp->fhp,
1221                     sizeof (fhandle_t)) == 0)
1222                         break;
1223         }

1225         /*
1226          * If lp is non-NULL, then there is already a cluster
1227          * started.  We need to place ourselves in the cluster
1228          * list in the right place as determined by starting
1229          * offset.  Conflicts with non-blocking mandatory locked
1230          * regions will be checked when the cluster is processed.
1231          */
1232         if (lp != NULL) {
1233                 rp = lp->list;
1234                 trp = NULL;
1235                 while (rp != NULL && rp->wa->wa_offset < wa->wa_offset) {
1236                         trp = rp;
1237                         rp = rp->list;
1238                 }
1239                 nrp->list = rp;
1240                 if (trp == NULL)
1241                         lp->list = nrp;
1242                 else
1243                         trp->list = nrp;
1244                 while (nrp->ns->ns_status == RFSWRITE_INITVAL)
1245                         cv_wait(&lp->cv, &rfs_async_write_lock);
1246                 mutex_exit(&rfs_async_write_lock);

1248                 return;
1249         }

1251         /*
1252          * No cluster started yet, start one and add ourselves
1253          * to the list of clusters.
1254          */
1255         nrp->list = NULL;

1257         nlp = &nlpsp;
1258         nlp->fhp = &wa->wa_fhandle;
1259         cv_init(&nlp->cv, NULL, CV_DEFAULT, NULL);
1260         nlp->list = nrp;
1261         nlp->next = NULL;

1263         if (rfs_async_write_head == NULL) {
1264                 rfs_async_write_head = nlp;
1265         } else {
1266                 lp = rfs_async_write_head;
1267                 while (lp->next != NULL)
1268                         lp = lp->next;
1269                 lp->next = nlp;
1270         }
```

```
1271         mutex_exit(&rfs_async_write_lock);

1273         /*
1274          * Convert the file handle common to all of the requests
1275          * in this cluster to a vnode.
1276          */
1277         vp = nfs_fhtovp(&wa->wa_fhandle, exi);
1278         if (vp == NULL) {
1279                 mutex_enter(&rfs_async_write_lock);
1280                 if (rfs_async_write_head == nlp)
1281                         rfs_async_write_head = nlp->next;
1282                 else {
1283                         lp = rfs_async_write_head;
1284                         while (lp->next != nlp)
1285                                 lp = lp->next;
1286                         lp->next = nlp->next;
1287                 }
1288                 t_flag = curthread->t_flag & T_WOULDBLOCK;
1289                 for (rp = nlp->list; rp != NULL; rp = rp->list) {
1290                         rp->ns->ns_status = NFSERR_STALE;
1291                         rp->thread->t_flag |= t_flag;
1292                 }
1293                 cv_broadcast(&nlp->cv);
1294                 mutex_exit(&rfs_async_write_lock);

1296                 return;
1297         }

1299         /*
1300          * Can only write regular files.  Attempts to write any
1301          * other file types fail with EISDIR.
1302          */
1303         if (vp->v_type != VREG) {
1304                 VN_RELE(vp);
1305                 mutex_enter(&rfs_async_write_lock);
1306                 if (rfs_async_write_head == nlp)
1307                         rfs_async_write_head = nlp->next;
1308                 else {
1309                         lp = rfs_async_write_head;
1310                         while (lp->next != nlp)
1311                                 lp = lp->next;
1312                         lp->next = nlp->next;
1313                 }
1314                 t_flag = curthread->t_flag & T_WOULDBLOCK;
1315                 for (rp = nlp->list; rp != NULL; rp = rp->list) {
1316                         rp->ns->ns_status = NFSERR_ISDIR;
1317                         rp->thread->t_flag |= t_flag;
1318                 }
1319                 cv_broadcast(&nlp->cv);
1320                 mutex_exit(&rfs_async_write_lock);

1322                 return;
1323         }

1325         /*
1326          * Enter the critical region before calling VOP_RWLOCK, to avoid a
1327          * deadlock with ufs.
1328          */
1329         if (nbl_need_check(vp)) {
1330                 nbl_start_crit(vp, RW_READER);
1331                 in_crit = 1;
1332         }

1334         ct.cc_sysid = 0;
1335         ct.cc_pid = 0;
1336         ct.cc_caller_id = nfs2_srv_caller_id;
```



```
1337         ct.cc_flags = CC_DONTBLOCK;

1339         /*
1340          * Lock the file for writing.  This operation provides
1341          * the delay which allows clusters to grow.
1342          */
1343         error = VOP_RWLOCK(vp, V_WRITELOCK_TRUE, &ct);

1345         /* check if a monitor detected a delegation conflict */
1346         if (error == EAGAIN && (ct.cc_flags & CC_WOULDBLOCK)) {
1347                 if (in_crit)
1348                         nbl_end_crit(vp);
1349                 VN_RELE(vp);
1350                 /* mark as wouldblock so response is dropped */
1351                 curthread->t_flag |= T_WOULDBLOCK;
1352                 mutex_enter(&rfs_async_write_lock);
1353                 if (rfs_async_write_head == nlp)
1354                         rfs_async_write_head = nlp->next;
1355                 else {
1356                         lp = rfs_async_write_head;
1357                         while (lp->next != nlp)
1358                                 lp = lp->next;
1359                         lp->next = nlp->next;
1360                 }
1361                 for (rp = nlp->list; rp != NULL; rp = rp->list) {
1362                         if (rp->ns->ns_status == RFSWRITE_INITVAL) {
1363                                 rp->ns->ns_status = puterrno(error);
1364                                 rp->thread->t_flag |= T_WOULDBLOCK;
1365                         }
1366                 }
1367                 cv_broadcast(&nlp->cv);
1368                 mutex_exit(&rfs_async_write_lock);

1370                 return;
1371         }

1373         /*
1374          * Disconnect this cluster from the list of clusters.
1375          * The cluster that is being dealt with must be fixed
1376          * in size after this point, so there is no reason
1377          * to leave it on the list so that new requests can
1378          * find it.
1379          *
1380          * The algorithm is that the first write request will
1381          * create a cluster, convert the file handle to a
1382          * vnode pointer, and then lock the file for writing.
1383          * This request is not likely to be clustered with
1384          * any others.  However, the next request will create
1385          * a new cluster and be blocked in VOP_RWLOCK while
1386          * the first request is being processed.  This delay
1387          * will allow more requests to be clustered in this
1388          * second cluster.
1389          */
1390         mutex_enter(&rfs_async_write_lock);
1391         if (rfs_async_write_head == nlp)
1392                 rfs_async_write_head = nlp->next;
1393         else {
1394                 lp = rfs_async_write_head;
1395                 while (lp->next != nlp)
1396                         lp = lp->next;
1397                 lp->next = nlp->next;
1398         }
1399         mutex_exit(&rfs_async_write_lock);

1401         /*
1402          * Step through the list of requests in this cluster.
```

```
1403          * We need to check permissions to make sure that all
1404          * of the requests have sufficient permission to write
1405          * the file.  A cluster can be composed of requests
1406          * from different clients and different users on each
1407          * client.
1408          *
1409          * As a side effect, we also calculate the size of the
1410          * byte range that this cluster encompasses.
1411          */
1412         rp = nlp->list;
1413         off = rp->wa->wa_offset;
1414         len = (uint_t)0;
1415         do {
1416                 if (rdonly(exi, vp, rp->req)) {
1417                         rp->ns->ns_status = NFSERR_ROFS;
1418                         t_flag = curthread->t_flag & T_WOULDBLOCK;
1419                         rp->thread->t_flag |= t_flag;
1420                         continue;
1421                 }

1423                 va.va_mask = AT_UID|AT_MODE;

1425                 error = VOP_GETATTR(vp, &va, 0, rp->cr, &ct);

1427                 if (!error) {
1428                         if (crgetuid(rp->cr) != va.va_uid) {
1429                                 /*
1430                                  * This is a kludge to allow writes of files
1431                                  * created with read only permission.  The
1432                                  * owner of the file is always allowed to
1433                                  * write it.
1434                                  */
1435                                 error = VOP_ACCESS(vp, VWRITE, 0, rp->cr, &ct);
1436                         }
1437                         if (!error && MANDLOCK(vp, va.va_mode))
1438                                 error = EACCES;
1439                 }

1441                 /*
1442                  * Check for a conflict with a nbmand-locked region.
1443                  */
1444                 if (in_crit && nbl_conflict(vp, NBL_WRITE, rp->wa->wa_offset,
1445                     rp->wa->wa_count, 0, NULL)) {
1446                         error = EACCES;
1447                 }

1449                 if (error) {
1450                         rp->ns->ns_status = puterrno(error);
1451                         t_flag = curthread->t_flag & T_WOULDBLOCK;
1452                         rp->thread->t_flag |= t_flag;
1453                         continue;
1454                 }
1455                 if (len < rp->wa->wa_offset + rp->wa->wa_count - off)
1456                         len = rp->wa->wa_offset + rp->wa->wa_count - off;
1457         } while ((rp = rp->list) != NULL);

1459         /*
1460          * Step through the cluster attempting to gather as many
1461          * requests which are contiguous as possible.  These
1462          * contiguous requests are handled via one call to VOP_WRITE
1463          * instead of different calls to VOP_WRITE.  We also keep
1464          * track of the fact that any data was written.
1465          */
1466         rp = nlp->list;
1467         data_written = 0;
1468         do {
```



```
1469                 /*
1470                  * Skip any requests which are already marked as having an
1471                  * error.
1472                  */
1473                 if (rp->ns->ns_status != RFSWRITE_INITVAL) {
1474                         rp = rp->list;
1475                         continue;
1476                 }

1478                 /*
1479                  * Count the number of iovec's which are required
1480                  * to handle this set of requests.  One iovec is
1481                  * needed for each data buffer, whether addressed
1482                  * by wa_data or by the b_rptr pointers in the
1483                  * mblk chains.
1484                  */
1485                 iovcnt = 0;
1486                 lrp = rp;
1487                 for (;;) {
1488                         if (lrp->wa->wa_data || lrp->wa->wa_rlist)
1489                                 iovcnt++;
1490                         else {
1491                                 m = lrp->wa->wa_mblk;
1492                                 while (m != NULL) {
1493                                         iovcnt++;
1494                                         m = m->b_cont;
1495                                 }
1496                         }
1497                         if (lrp->list == NULL ||
1498                             lrp->list->ns->ns_status != RFSWRITE_INITVAL ||
1499                             lrp->wa->wa_offset + lrp->wa->wa_count !=
1500                             lrp->list->wa->wa_offset) {
1501                                 lrp = lrp->list;
1502                                 break;
1503                         }
1504                         lrp = lrp->list;
1505                 }

1507                 if (iovcnt <= MAXCLIOVECS) {
1508 #ifdef DEBUG
1509                         rfs_write_hits++;
1510 #endif
1511                         niovp = iov;
1512                 } else {
1513 #ifdef DEBUG
1514                         rfs_write_misses++;
1515 #endif
1516                         niovp = kmem_alloc(sizeof (*niovp) * iovcnt, KM_SLEEP);
1517                 }
1518                 /*
1519                  * Put together the scatter/gather iovecs.
1520                  */
1521                 iovp = niovp;
1522                 trp = rp;
1523                 count = 0;
1524                 do {
1525                         if (trp->wa->wa_data || trp->wa->wa_rlist) {
1526                                 if (trp->wa->wa_rlist) {
1527                                         iovp->iov_base =
1528                                             (char *)((trp->wa->wa_rlist)->
1529                                             u.c_daddr3);
1530                                         iovp->iov_len = trp->wa->wa_count;
1531                                 } else  {
1532                                         iovp->iov_base = trp->wa->wa_data;
1533                                         iovp->iov_len = trp->wa->wa_count;
1534                                 }
```

```
1535                                    iovp++;
1536                            } else {
1537                                    m = trp->wa->wa_mblk;
1538                                    rcount = trp->wa->wa_count;
1539                                    while (m != NULL) {
1540                                            iovp->iov_base = (caddr_t)m->b_rptr;
1541                                            iovp->iov_len = (m->b_wptr - m->b_rptr);
1542                                            rcount -= iovp->iov_len;
1543                                            if (rcount < 0)
1544                                                    iovp->iov_len += rcount;
1545                                            iovp++;
1546                                            if (rcount <= 0)
1547                                                    break;
1548                                            m = m->b_cont;
1549                                    }
1550                            }
1551                            count += trp->wa->wa_count;
1552                            trp = trp->list;
1553                    } while (trp != lrp);

1555                    uio.uio_iov = niovp;
1556                    uio.uio_iovcnt = iovcnt;
1557                    uio.uio_segflg = UIO_SYSSPACE;
1558                    uio.uio_extflg = UIO_COPY_DEFAULT;
1559                    uio.uio_loffset = (offset_t)rp->wa->wa_offset;
1560                    uio.uio_resid = count;
1561                    /*
1562                     * The limit is checked on the client. We
1563                     * should allow any size writes here.
1564                     */
1565                    uio.uio_llimit = curproc->p_fsz_ctl;
1566                    rlimit = uio.uio_llimit - rp->wa->wa_offset;
1567                    if (rlimit < (rlim64_t)uio.uio_resid)
1568                            uio.uio_resid = (uint_t)rlimit;

1570                    /*
1571                     * For now we assume no append mode.
1572                     */

1574                    /*
1575                     * We're changing creds because VM may fault
1576                     * and we need the cred of the current
1577                     * thread to be used if quota * checking is
1578                     * enabled.
1579                     */
1580                    savecred = curthread->t_cred;
1581                    curthread->t_cred = cr;
1582                    error = VOP_WRITE(vp, &uio, 0, rp->cr, &ct);
1583                    curthread->t_cred = savecred;

1585                    /* check if a monitor detected a delegation conflict */
1586                    if (error == EAGAIN && (ct.cc_flags & CC_WOULDBLOCK))
1587                            /* mark as wouldblock so response is dropped */
1588                            curthread->t_flag |= T_WOULDBLOCK;

1590                    if (niovp != iov)
1591                            kmem_free(niovp, sizeof (*niovp) * iovcnt);

1593                    if (!error) {
1594                            data_written = 1;
1595                            /*
1596                             * Get attributes again so we send the latest mod
1597                             * time to the client side for his cache.
1598                             */
1599                            va.va_mask = AT_ALL;    /* now we want everything */
```



```
1601                            error = VOP_GETATTR(vp, &va, 0, rp->cr, &ct);

1603                            if (!error)
1604                                    acl_perm(vp, exi, &va, rp->cr);
1605                    }

1607                    /*
1608                     * Fill in the status responses for each request
1609                     * which was just handled.  Also, copy the latest
1610                     * attributes in to the attribute responses if
1611                     * appropriate.
1612                     */
1613                    t_flag = curthread->t_flag & T_WOULDBLOCK;
1614                    do {
1615                            rp->thread->t_flag |= t_flag;
1616                            /* check for overflows */
1617                            if (!error) {
1618                                    error  = vattr_to_nattr(&va, &rp->ns->ns_attr);
1619                            }
1620                            rp->ns->ns_status = puterrno(error);
1621                            rp = rp->list;
1622                    } while (rp != lrp);
1623            } while (rp != NULL);

1625            /*
1626             * If any data was written at all, then we need to flush
1627             * the data and metadata to stable storage.
1628             */
1629            if (data_written) {
1630                    error = VOP_PUTPAGE(vp, (u_offset_t)off, len, 0, cr, &ct);

1632                    if (!error) {
1633                            error = VOP_FSYNC(vp, FNODSYNC, cr, &ct);
1634                    }
1635            }

1637            VOP_RWUNLOCK(vp, V_WRITELOCK_TRUE, &ct);

1639            if (in_crit)
1640                    nbl_end_crit(vp);
1641            VN_RELE(vp);

1643            t_flag = curthread->t_flag & T_WOULDBLOCK;
1644            mutex_enter(&rfs_async_write_lock);
1645            for (rp = nlp->list; rp != NULL; rp = rp->list) {
1646                    if (rp->ns->ns_status == RFSWRITE_INITVAL) {
1647                            rp->ns->ns_status = puterrno(error);
1648                            rp->thread->t_flag |= t_flag;
1649                    }
1650            }
1651            cv_broadcast(&nlp->cv);
1652            mutex_exit(&rfs_async_write_lock);

1654 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   73799 Fri Mar 28 23:33:26 2014
new/usr/src/uts/common/os/clock.c
patch clock-wakeup-remove
**********************************************************
______unchanged_portion_omitted_

 370 /*
 371  * test hook for tod broken detection in tod_validate
 372  */
 373 int tod_unit_test = 0;
 374 time_t tod_test_injector;

 376 #define CLOCK_ADJ_HIST_SIZE     4

 378 static int      adj_hist_entry;

 380 int64_t clock_adj_hist[CLOCK_ADJ_HIST_SIZE];

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

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

 389 extern clock_t clock_tick_proc_max;

 391 static int64_t deadman_counter = 0;

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

 410         if (panicstr)
 411                 return;

 413         /*
 414          * Make sure that 'freemem' do not drift too far from the truth
 415          */
 416         set_freemem();


 419         /*
 420          * Before the section which is repeated is executed, we do
 421          * the time delta processing which occurs every clock tick
 422          *
 423          * There is additional processing which happens every time
 424          * the nanosecond counter rolls over which is described
 425          * below - see the section which begins with : if (one_sec)
 426          *
 427          * This section marks the beginning of the precision-kernel
 428          * code fragment.
```



```
 429          *
 430          * First, compute the phase adjustment. If the low-order bits
 431          * (time_phase) of the update overflow, bump the higher order
 432          * bits (time_update).
 433          */
 434         time_phase += time_adj;
 435         if (time_phase <= -FINEUSEC) {
 436                 ltemp = -time_phase / SCALE_PHASE;
 437                 time_phase += ltemp * SCALE_PHASE;
 438                 s = hr_clock_lock();
 439                 timedelta -= ltemp * (NANOSEC/MICROSEC);
 440                 hr_clock_unlock(s);
 441         } else if (time_phase >= FINEUSEC) {
 442                 ltemp = time_phase / SCALE_PHASE;
 443                 time_phase -= ltemp * SCALE_PHASE;
 444                 s = hr_clock_lock();
 445                 timedelta += ltemp * (NANOSEC/MICROSEC);
 446                 hr_clock_unlock(s);
 447         }

 449         /*
 450          * End of precision-kernel code fragment which is processed
 451          * every timer interrupt.
 452          *
 453          * Continue with the interrupt processing as scheduled.
 454          */
 455         /*
 456          * Count the number of runnable threads and the number waiting
 457          * for some form of I/O to complete -- gets added to
 458          * sysinfo.waiting.  To know the state of the system, must add
 459          * wait counts from all CPUs.  Also add up the per-partition
 460          * statistics.
 461          */
 462         w_io = 0;
 463         nrunnable = 0;

 465         /*
 466          * keep track of when to update lgrp/part loads
 467          */

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

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

 480         /*
 481          * First count the threads waiting on kpreempt queues in each
 482          * CPU partition.
 483          */

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

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

```
495                 }
496         } while ((cpupart = cpupart->cp_next) != cp_list_head);


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

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

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

528                         /*
529                          * Estimate interrupt load on this cpu each second.
530                          * Computes cpu_intrload as %utilization (0-99).
531                          */

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

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

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

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

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

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



```
561                 }

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

578                         /*
579                          * Account for the load average for this thread if
580                          * it isn't the idle thread or it is on the interrupt
581                          * stack and not the current CPU handling the clock
582                          * interrupt
583                          */
584                         if ((t && t != cp->cpu_idle_thread) || (CPU != cp &&
585                             CPU_ON_INTR(cp))) {
586                                 if (t->t_lpl == cp->cpu_lpl) {
587                                         /* local thread */
588                                         cpu_nrunnable++;
589                                 } else {
590                                         /*
591                                          * This is a remote thread, charge it
592                                          * against its home lgroup.  Note that
593                                          * we notice that a thread is remote
594                                          * only if it's currently executing.
595                                          * This is a reasonable approximation,
596                                          * since queued remote threads are rare.
597                                          * Note also that if we didn't charge
598                                          * it to its home lgroup, remote
599                                          * execution would often make a system
600                                          * appear balanced even though it was
601                                          * not, and thread placement/migration
602                                          * would often not be done correctly.
603                                          */
604                                         lgrp_loadavg(t->t_lpl,
605                                             LGRP_LOADAVG_IN_THREAD_MAX, 0);
606                                 }
607                         }
608                         lgrp_loadavg(cp->cpu_lpl,
609                             cpu_nrunnable * LGRP_LOADAVG_IN_THREAD_MAX, 1);
610                 }
611         } while ((cp = cp->cpu_next) != cpu_list);

613         clock_tick_schedule(one_sec);

615         /*
616          * Check for a callout that needs be called from the clock
617          * thread to support the membership protocol in a clustered
618          * system.  Copy the function pointer so that we can reset
619          * this to NULL if needed.
620          */
621         if ((funcp = cmm_clock_callout) != NULL)
622                 (*funcp)();

624         if ((funcp = cpucaps_clock_callout) != NULL)
625                 (*funcp)();
```

```
627         /*
628          * Wakeup the cageout thread waiters once per second.
629          */
630         if (one_sec)
631                 kcage_tick();

633         if (one_sec) {

635                 int drift, absdrift;
636                 timestruc_t tod;
637                 int s;

639                 /*
640                  * Beginning of precision-kernel code fragment executed
641                  * every second.
642                  *
643                  * On rollover of the second the phase adjustment to be
644                  * used for the next second is calculated.  Also, the
645                  * maximum error is increased by the tolerance.  If the
646                  * PPS frequency discipline code is present, the phase is
647                  * increased to compensate for the CPU clock oscillator
648                  * frequency error.
649                  *
650                  * On a 32-bit machine and given parameters in the timex.h
651                  * header file, the maximum phase adjustment is +-512 ms
652                  * and maximum frequency offset is (a tad less than)
653                  * +-512 ppm. On a 64-bit machine, you shouldn't need to ask.
654                  */
655                 time_maxerror += time_tolerance / SCALE_USEC;

657                 /*
658                  * Leap second processing. If in leap-insert state at
659                  * the end of the day, the system clock is set back one
660                  * second; if in leap-delete state, the system clock is
661                  * set ahead one second. The microtime() routine or
662                  * external clock driver will insure that reported time
663                  * is always monotonic. The ugly divides should be
664                  * replaced.
665                  */
666                 switch (time_state) {

668                 case TIME_OK:
669                         if (time_status & STA_INS)
670                                 time_state = TIME_INS;
671                         else if (time_status & STA_DEL)
672                                 time_state = TIME_DEL;
673                         break;

675                 case TIME_INS:
676                         if (hrestime.tv_sec % 86400 == 0) {
677                                 s = hr_clock_lock();
678                                 hrestime.tv_sec--;
679                                 hr_clock_unlock(s);
680                                 time_state = TIME_OOP;
681                         }
682                         break;

684                 case TIME_DEL:
685                         if ((hrestime.tv_sec + 1) % 86400 == 0) {
686                                 s = hr_clock_lock();
687                                 hrestime.tv_sec++;
688                                 hr_clock_unlock(s);
689                                 time_state = TIME_WAIT;
690                         }
691                         break;
```



```
693                 case TIME_OOP:
694                         time_state = TIME_WAIT;
695                         break;

697                 case TIME_WAIT:
698                         if (!(time_status & (STA_INS | STA_DEL)))
699                                 time_state = TIME_OK;
700                 default:
701                         break;
702                 }

704                 /*
705                  * Compute the phase adjustment for the next second. In
706                  * PLL mode, the offset is reduced by a fixed factor
707                  * times the time constant. In FLL mode the offset is
708                  * used directly. In either mode, the maximum phase
709                  * adjustment for each second is clamped so as to spread
710                  * the adjustment over not more than the number of
711                  * seconds between updates.
712                  */
713                 if (time_offset == 0)
714                         time_adj = 0;
715                 else if (time_offset < 0) {
716                         lltemp = -time_offset;
717                         if (!(time_status & STA_FLL)) {
718                                 if ((1 << time_constant) >= SCALE_KG)
719                                         lltemp *= (1 << time_constant) /
720                                             SCALE_KG;
721                                 else
722                                         lltemp = (lltemp / SCALE_KG) >>
723                                             time_constant;
724                         }
725                         if (lltemp > (MAXPHASE / MINSEC) * SCALE_UPDATE)
726                                 lltemp = (MAXPHASE / MINSEC) * SCALE_UPDATE;
727                         time_offset += lltemp;
728                         time_adj = -(lltemp * SCALE_PHASE) / hz / SCALE_UPDATE;
729                 } else {
730                         lltemp = time_offset;
731                         if (!(time_status & STA_FLL)) {
732                                 if ((1 << time_constant) >= SCALE_KG)
733                                         lltemp *= (1 << time_constant) /
734                                             SCALE_KG;
735                                 else
736                                         lltemp = (lltemp / SCALE_KG) >>
737                                             time_constant;
738                         }
739                         if (lltemp > (MAXPHASE / MINSEC) * SCALE_UPDATE)
740                                 lltemp = (MAXPHASE / MINSEC) * SCALE_UPDATE;
741                         time_offset -= lltemp;
742                         time_adj = (lltemp * SCALE_PHASE) / hz / SCALE_UPDATE;
743                 }

745                 /*
746                  * Compute the frequency estimate and additional phase
747                  * adjustment due to frequency error for the next
748                  * second. When the PPS signal is engaged, gnaw on the
749                  * watchdog counter and update the frequency computed by
750                  * the pll and the PPS signal.
751                  */
752                 pps_valid++;
753                 if (pps_valid == PPS_VALID) {
754                         pps_jitter = MAXTIME;
755                         pps_stabil = MAXFREQ;
756                         time_status &= ~(STA_PPSSIGNAL | STA_PPSJITTER |
757                             STA_PPSWANDER | STA_PPSERROR);
758                 }
```

```
759                 lltemp = time_freq + pps_freq;
760
761                 if (lltemp)
762                         time_adj += (lltemp * SCALE_PHASE) / (SCALE_USEC * hz);
763
764                 /*
765                  * End of precision kernel-code fragment
766                  *
767                  * The section below should be modified if we are planning
768                  * to use NTP for synchronization.
769                  *
770                  * Note: the clock synchronization code now assumes
771                  * the following:
772                  *   - if dosynctodr is 1, then compute the drift between
773                  *      the tod chip and software time and adjust one or
774                  *      the other depending on the circumstances
775                  *
776                  *   - if dosynctodr is 0, then the tod chip is independent
777                  *      of the software clock and should not be adjusted,
778                  *      but allowed to free run.  this allows NTP to sync.
779                  *      hrestime without any interference from the tod chip.
780                  */
781
782                 tod_validate_deferred = B_FALSE;
783                 mutex_enter(&tod_lock);
784                 tod = tod_get();
785                 drift = tod.tv_sec - hrestime.tv_sec;
786                 absdrift = (drift >= 0) ? drift : -drift;
787                 if (tod_needsync || absdrift > 1) {
788                         int s;
789                         if (absdrift > 2) {
790                                 if (!tod_broken && tod_faulted == TOD_NOFAULT) {
791                                         s = hr_clock_lock();
792                                         hrestime = tod;
793                                         membar_enter(); /* hrestime visible */
794                                         timedelta = 0;
795                                         timechanged++;
796                                         tod_needsync = 0;
797                                         hr_clock_unlock(s);
798                                         callout_hrestime();
799
800                                 }
801                         } else {
802                                 if (tod_needsync || !dosynctodr) {
803                                         gethrestime(&tod);
804                                         tod_set(tod);
805                                         s = hr_clock_lock();
806                                         if (timedelta == 0)
807                                                 tod_needsync = 0;
808                                         hr_clock_unlock(s);
809                                 } else {
810                                         /*
811                                          * If the drift is 2 seconds on the
812                                          * money, then the TOD is adjusting
813                                          * the clock;  record that.
814                                          */
815                                         clock_adj_hist[adj_hist_entry++ %
816                                             CLOCK_ADJ_HIST_SIZE] = now;
817                                         s = hr_clock_lock();
818                                         timedelta = (int64_t)drift*NANOSEC;
819                                         hr_clock_unlock(s);
820                                 }
821                         }
822                 }
823                 one_sec = 0;
824                 time = gethrestime_sec();  /* for crusty old kmem readers */
```



```
825                 mutex_exit(&tod_lock);
826
827                 /*
828                  * Some drivers still depend on this... XXX
829                  */
830                 cv_broadcast(&lbolt_cv);
831
832                 vminfo.freemem += freemem;
833                 {
834                         pgcnt_t maxswap, resv, free;
835                         pgcnt_t avail =
836                             MAX((spgcnt_t)(availrmem - swapfs_minfree), 0);
837
838                         maxswap = k_anoninfo.ani_mem_resv +
839                             k_anoninfo.ani_max +avail;
840                         /* Update ani_free */
841                         set_anoninfo();
842                         free = k_anoninfo.ani_free + avail;
843                         resv = k_anoninfo.ani_phys_resv +
844                             k_anoninfo.ani_mem_resv;
845
846                         vminfo.swap_resv += resv;
847                         /* number of reserved and allocated pages */
848 #ifdef  DEBUG
849                         if (maxswap < free)
850                                 cmn_err(CE_WARN, "clock: maxswap < free");
851                         if (maxswap < resv)
852                                 cmn_err(CE_WARN, "clock: maxswap < resv");
853 #endif
854                         vminfo.swap_alloc += maxswap - free;
855                         vminfo.swap_avail += maxswap - resv;
856                         vminfo.swap_free += free;
857                 }
858                 vminfo.updates++;
859                 if (nrunnable) {
860                         sysinfo.runque += nrunnable;
861                         sysinfo.runocc++;
862                 }
863                 if (nswapped) {
864                         sysinfo.swpque += nswapped;
865                         sysinfo.swpocc++;
866                 }
867                 sysinfo.waiting += w_io;
868                 sysinfo.updates++;
869
870                 /*
871                  * Wake up fsflush to write out DELWRI
872                  * buffers, dirty pages and other cached
873                  * administrative data, e.g. inodes.
874                  */
875                 if (--fsflushcnt <= 0) {
876                         fsflushcnt = tune.t_fsflushr;
877                         cv_signal(&fsflush_cv);
878                 }
879
880                 vmmeter();
881                 calcloadavg(genloadavg(&loadavg), hp_avenrun);
882                 for (i = 0; i < 3; i++)
883                         /*
884                          * At the moment avenrun[] can only hold 31
885                          * bits of load average as it is a signed
886                          * int in the API. We need to ensure that
887                          * hp_avenrun[i] >> (16 - FSHIFT) will not be
888                          * too large. If it is, we put the largest value
889                          * that we can use into avenrun[i]. This is
890                          * kludgey, but about all we can do until we
```

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

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

905                 /*
906                  * Wake up the swapper thread if necessary.
907                  */
908                 if (runin ||
909                     (runout && (avefree < desfree || wake_sched_sec))) {
910                         t = &t0;
911                         thread_lock(t);
912                         if (t->t_state == TS_STOPPED) {
913                                 runin = runout = 0;
914                                 wake_sched_sec = 0;
915                                 t->t_whystop = 0;
916                                 t->t_whatstop = 0;
917                                 t->t_schedflag &= ~TS_ALLSTART;
918                                 THREAD_TRANSITION(t);
919                                 setfrontdq(t);
920                         }
921                         thread_unlock(t);
922                 }
923         }

925         /*
926          * Wake up the swapper if any high priority swapped-out threads
927          * became runable during the last tick.
928          */
929         if (wake_sched) {
930                 t = &t0;
931                 thread_lock(t);
932                 if (t->t_state == TS_STOPPED) {
933                         runin = runout = 0;
934                         wake_sched = 0;
935                         t->t_whystop = 0;
936                         t->t_whatstop = 0;
937                         t->t_schedflag &= ~TS_ALLSTART;
938                         THREAD_TRANSITION(t);
939                         setfrontdq(t);
940                 }
941                 thread_unlock(t);
904         }
905 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   21374 Fri Mar 28 23:33:28 2014
new/usr/src/uts/common/os/condvar.c
patch remove-dont-swap-flag
**********************************************************
______unchanged_portion_omitted_

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

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

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

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

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

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

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



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

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

 535         if (panicstr)
 536                 return (rval);

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

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

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

_____unchanged_portion_omitted_

```
**********************************************************
   94480 Fri Mar 28 23:33:30 2014
new/usr/src/uts/common/os/cpu.c
patch remove-load-flag
patch remove-on-swapq-flag
**********************************************************
______unchanged_portion_omitted_

2522 /*
2523  * Bind a thread to a CPU as requested.
2524  */
2525 int
2526 cpu_bind_thread(kthread_id_t tp, processorid_t bind, processorid_t *obind,
2527     int *error)
2528 {
2529         processorid_t   binding;
2530         cpu_t           *cp = NULL;

2532         ASSERT(MUTEX_HELD(&cpu_lock));
2533         ASSERT(MUTEX_HELD(&ttoproc(tp)->p_lock));

2535         thread_lock(tp);

2537         /*
2538          * Record old binding, but change the obind, which was initialized
2539          * to PBIND_NONE, only if this thread has a binding.  This avoids
2540          * reporting PBIND_NONE for a process when some LWPs are bound.
2541          */
2542         binding = tp->t_bind_cpu;
2543         if (binding != PBIND_NONE)
2544                 *obind = binding;       /* record old binding */

2546         switch (bind) {
2547         case PBIND_QUERY:
2548                 /* Just return the old binding */
2549                 thread_unlock(tp);
2550                 return (0);

2552         case PBIND_QUERY_TYPE:
2553                 /* Return the binding type */
2554                 *obind = TB_CPU_IS_SOFT(tp) ? PBIND_SOFT : PBIND_HARD;
2555                 thread_unlock(tp);
2556                 return (0);

2558         case PBIND_SOFT:
2559                 /*
2560                  *  Set soft binding for this thread and return the actual
2561                  *  binding
2562                  */
2563                 TB_CPU_SOFT_SET(tp);
2564                 thread_unlock(tp);
2565                 return (0);

2567         case PBIND_HARD:
2568                 /*
2569                  *  Set hard binding for this thread and return the actual
2570                  *  binding
2571                  */
2572                 TB_CPU_HARD_SET(tp);
2573                 thread_unlock(tp);
2574                 return (0);

2576         default:
2577                 break;
2578         }
```



```
2580         /*
2581          * If this thread/LWP cannot be bound because of permission
2582          * problems, just note that and return success so that the
2583          * other threads/LWPs will be bound.  This is the way
2584          * processor_bind() is defined to work.
2585          *
2586          * Binding will get EPERM if the thread is of system class
2587          * or hasprocperm() fails.
2588          */
2589         if (tp->t_cid == 0 || !hasprocperm(tp->t_cred, CRED())) {
2590                 *error = EPERM;
2591                 thread_unlock(tp);
2592                 return (0);
2593         }

2595         binding = bind;
2596         if (binding != PBIND_NONE) {
2597                 cp = cpu_get((processorid_t)binding);
2598                 /*
2599                  * Make sure binding is valid and is in right partition.
2600                  */
2601                 if (cp == NULL || tp->t_cpupart != cp->cpu_part) {
2602                         *error = EINVAL;
2603                         thread_unlock(tp);
2604                         return (0);
2605                 }
2606         }
2607         tp->t_bind_cpu = binding;       /* set new binding */

2609         /*
2610          * If there is no system-set reason for affinity, set
2611          * the t_bound_cpu field to reflect the binding.
2612          */
2613         if (tp->t_affinitycnt == 0) {
2614                 if (binding == PBIND_NONE) {
2615                         /*
2616                          * We may need to adjust disp_max_unbound_pri
2617                          * since we're becoming unbound.
2618                          */
2619                         disp_adjust_unbound_pri(tp);

2621                         tp->t_bound_cpu = NULL; /* set new binding */

2623                         /*
2624                          * Move thread to lgroup with strongest affinity
2625                          * after unbinding
2626                          */
2627                         if (tp->t_lgrp_affinity)
2628                                 lgrp_move_thread(tp,
2629                                     lgrp_choose(tp, tp->t_cpupart), 1);

2631                         if (tp->t_state == TS_ONPROC &&
2632                             tp->t_cpu->cpu_part != tp->t_cpupart)
2633                                 cpu_surrender(tp);
2634                 } else {
2635                         lpl_t   *lpl;

2637                         tp->t_bound_cpu = cp;
2638                         ASSERT(cp->cpu_lpl != NULL);

2640                         /*
2641                          * Set home to lgroup with most affinity containing CPU
2642                          * that thread is being bound or minimum bounding
2643                          * lgroup if no affinities set
2644                          */
2645                         if (tp->t_lgrp_affinity)
```

```
2646                                    lpl = lgrp_affinity_best(tp, tp->t_cpupart,
2647                                        LGRP_NONE, B_FALSE);
2648                            else
2649                                    lpl = cp->cpu_lpl;

2651                            if (tp->t_lpl != lpl) {
2652                                    /* can't grab cpu_lock */
2653                                    lgrp_move_thread(tp, lpl, 1);
2654                            }

2656                            /*
2657                             * Make the thread switch to the bound CPU.
2658                             * If the thread is runnable, we need to
2659                             * requeue it even if t_cpu is already set
2660                             * to the right CPU, since it may be on a
2661                             * kpreempt queue and need to move to a local
2662                             * queue.  We could check t_disp_queue to
2663                             * avoid unnecessary overhead if it's already
2664                             * on the right queue, but since this isn't
2665                             * a performance-critical operation it doesn't
2666                             * seem worth the extra code and complexity.
2667                             *
2668                             * If the thread is weakbound to the cpu then it will
2669                             * resist the new binding request until the weak
2670                             * binding drops.  The cpu_surrender or requeueing
2671                             * below could be skipped in such cases (since it
2672                             * will have no effect), but that would require
2673                             * thread_allowmigrate to acquire thread_lock so
2674                             * we'll take the very occasional hit here instead.
2675                             */
2676                            if (tp->t_state == TS_ONPROC) {
2677                                    cpu_surrender(tp);
2678                            } else if (tp->t_state == TS_RUN) {
2679                                    cpu_t *ocp = tp->t_cpu;

2681                                    (void) dispdeq(tp);
2682                                    setbackdq(tp);
2683                                    /*
2684                                     * On the bound CPU's disp queue now.
2684                                     * Either on the bound CPU's disp queue now,
2685                                     * or swapped out or on the swap queue.
2685                                     */
2686                                    ASSERT(tp->t_disp_queue == cp->cpu_disp ||
2687                                        tp->t_weakbound_cpu == ocp);
2688                                        tp->t_weakbound_cpu == ocp ||
2689                                        (tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ))
2690                                        != TS_LOAD);
2688                            }
2689                    }
2690            }

2692            /*
2693             * Our binding has changed; set TP_CHANGEBIND.
2694             */
2695            tp->t_proc_flag |= TP_CHANGEBIND;
2696            aston(tp);

2698            thread_unlock(tp);

2700            return (0);
2701 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
   15482 Fri Mar 28 23:33:31 2014
new/usr/src/uts/common/os/panic.c
patch remove-dont-swap-flag
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1999, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright (c) 2011, Joyent, Inc. All rights reserved.
 */

/*
 * When the operating system detects that it is in an invalid state, a panic
 * is initiated in order to minimize potential damage to user data and to
 * facilitate debugging.  There are three major tasks to be performed in
 * a system panic: recording information about the panic in memory (and thus
 * making it part of the crash dump), synchronizing the file systems to
 * preserve user file data, and generating the crash dump.  We define the
 * system to be in one of four states with respect to the panic code:
 *
 * CALM    - the state of the system prior to any thread initiating a panic
 *
 * QUIESCE - the state of the system when the first thread to initiate
 *           a system panic records information about the cause of the panic
 *           and renders the system quiescent by stopping other processors
 *
 * SYNC    - the state of the system when we synchronize the file systems
 * DUMP    - the state when we generate the crash dump.
 *
 * The transitions between these states are irreversible: once we begin
 * panicking, we only make one attempt to perform the actions associated with
 * each state.
 *
 * The panic code itself must be re-entrant because actions taken during any
 * state may lead to another system panic.  Additionally, any Solaris
 * thread may initiate a panic at any time, and so we must have synchronization
 * between threads which attempt to initiate a state transition simultaneously.
 * The panic code makes use of a special locking primitive, a trigger, to
 * perform this synchronization.  A trigger is simply a word which is set
 * atomically and can only be set once.  We declare three triggers, one for
 * each transition between the four states.  When a thread enters the panic
 * code it attempts to set each trigger; if it fails it moves on to the
 * next trigger.  A special case is the first trigger: if two threads race
 * to perform the transition to QUIESCE, the losing thread may execute before
```



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

#include <sys/types.h>
#include <sys/varargs.h>
#include <sys/sysmacros.h>
#include <sys/cmn_err.h>
#include <sys/cpuvar.h>
```

```
#include <sys/thread.h>
#include <sys/t_lock.h>
#include <sys/cred.h>
#include <sys/systm.h>
#include <sys/archsystm.h>
#include <sys/uadmin.h>
#include <sys/callb.h>
#include <sys/vfs.h>
#include <sys/log.h>
#include <sys/disp.h>
#include <sys/param.h>
#include <sys/dumphdr.h>
#include <sys/ftrace.h>
#include <sys/reboot.h>
#include <sys/debug.h>
#include <sys/stack.h>
#include <sys/spl.h>
#include <sys/errorq.h>
#include <sys/panic.h>
#include <sys/fm/util.h>
#include <sys/clock_impl.h>

/*
 * Panic variables which are set once during the QUIESCE state by the
 * first thread to initiate a panic.  These are examined by post-mortem
 * debugging tools; the inconsistent use of 'panic' versus 'panic_' in
 * the variable naming is historical and allows legacy tools to work.
 */
#pragma align STACK_ALIGN(panic_stack)
char panic_stack[PANICSTKSIZE];         /* reserved stack for panic_thread */
kthread_t *panic_thread;                /* first thread to call panicsys() */
cpu_t panic_cpu;                        /* cpu from first call to panicsys() */
label_t panic_regs;                     /* setjmp label from panic_thread */
label_t panic_pcb;                      /* t_pcb at time of panic */
struct regs *panic_reg;                 /* regs struct from first panicsys() */
char *volatile panicstr;                /* format string to first panicsys() */
va_list panicargs;                      /* arguments to first panicsys() */
clock_t panic_lbolt;                    /* lbolt at time of panic */
int64_t panic_lbolt64;                  /* lbolt64 at time of panic */
hrtime_t panic_hrtime;                  /* hrtime at time of panic */
timespec_t panic_hrestime;              /* hrestime at time of panic */
int panic_ipl;                          /* ipl on panic_cpu at time of panic */
ushort_t panic_schedflag;               /* t_schedflag for panic_thread */
cpu_t *panic_bound_cpu;                 /* t_bound_cpu for panic_thread */
char panic_preempt;                     /* t_preempt for panic_thread */

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

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

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



```
int panic_forced = 0;

/*
 * Triggers for panic state transitions:
 */
int panic_quiesce;                      /* trigger for CALM    -> QUIESCE */
int panic_sync;                         /* trigger for QUIESCE -> SYNC */
int panic_dump;                         /* trigger for SYNC    -> DUMP */

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

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

        caddr_t intr_stack = NULL;
        uint_t intr_actv;

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

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

        t->t_schedflag |= TS_DONT_SWAP;
        t->t_bound_cpu = cp;
        t->t_preempt++;

        panic_enter_hw(s);

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

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

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

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

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

```
259                 panic_data_t *pdp = (panic_data_t *)panicbuf;
261                 pdp->pd_version = PANICBUFVERS;
262                 pdp->pd_msgoff = sizeof (panic_data_t) - sizeof (panic_nv_t);
264                 (void) strncpy(pdp->pd_uuid, dump_get_uuid(),
265                     sizeof (pdp->pd_uuid));
267                 if (t->t_panic_trap != NULL)
268                         panic_savetrap(pdp, t->t_panic_trap);
269                 else
270                         panic_saveregs(pdp, rp);
272                 (void) vsnprintf(&panicbuf[pdp->pd_msgoff],
273                     PANICBUFSIZE - pdp->pd_msgoff, format, alist);
275                 /*
276                  * Call into the platform code to stop the other CPUs.
277                  * We currently have all interrupts blocked, and expect that
278                  * the platform code will lower ipl only as far as needed to
279                  * perform cross-calls, and will acquire as *few* locks as is
280                  * possible -- panicstr is not set so we can still deadlock.
281                  */
282                 panic_stopcpus(cp, t, s);
284                 panicstr = (char *)format;
285                 va_copy(panicargs, alist);
286                 panic_lbolt = LBOLT_NO_ACCOUNT;
287                 panic_lbolt64 = LBOLT_NO_ACCOUNT64;
288                 panic_hrestime = hrestime;
289                 panic_hrtime = gethrtime_waitfree();
290                 panic_thread = t;
291                 panic_regs = t->t_pcb;
292                 panic_reg = rp;
293                 panic_cpu = *cp;
294                 panic_ipl = spltoipl(s);
295                 panic_schedflag = schedflag;
296                 panic_bound_cpu = bound_cpu;
297                 panic_preempt = preempt;
298                 panic_pcb = pcb;
300                 if (intr_stack != NULL) {
301                         panic_cpu.cpu_intr_stack = intr_stack;
302                         panic_cpu.cpu_intr_actv = intr_actv;
303                 }
305                 /*
306                  * Lower ipl to 10 to keep clock() from running, but allow
307                  * keyboard interrupts to enter the debugger.  These callbacks
308                  * are executed with panicstr set so they can bypass locks.
309                  */
310                 splx(ipltospl(CLOCK_LEVEL));
311                 panic_quiesce_hw(pdp);
312                 (void) FTRACE_STOP();
313                 (void) callb_execute_class(CB_CL_PANIC, NULL);
315                 if (log_intrq != NULL)
316                         log_flushq(log_intrq);
318                 /*
319                  * If log_consq has been initialized and syslogd has started,
320                  * print any messages in log_consq that haven't been consumed.
321                  */
322                 if (log_consq != NULL && log_consq != log_backlogq)
323                         log_printq(log_consq);
```



```
325                 fm_banner();
327 #if defined(__x86)
328                 /*
329                  * A hypervisor panic originates outside of Solaris, so we
330                  * don't want to prepend the panic message with misleading
331                  * pointers from within Solaris.
332                  */
333                 if (!IN_XPV_PANIC())
334 #endif
335                         printf("\n\rpanic[cpu%d]/thread=%p: ", cp->cpu_id,
336                             (void *)t);
337                 vprintf(format, alist);
338                 printf("\n\n");
340                 if (t->t_panic_trap != NULL) {
341                         panic_showtrap(t->t_panic_trap);
342                         printf("\n");
343                 }
345                 traceregs(rp);
346                 printf("\n");
348                 if (((boothowto & RB_DEBUG) || obpdebug) &&
349                     !nopanicdebug && !panic_forced) {
350                         if (dumpvp != NULL) {
351                                 debug_enter("panic: entering debugger "
352                                     "(continue to save dump)");
353                         } else {
354                                 debug_enter("panic: entering debugger "
355                                     "(no dump device, continue to reboot)");
356                         }
357                 }
359         } else if (panic_dump != 0 || panic_sync != 0 || panicstr != NULL) {
360                 printf("\n\rpanic[cpu%d]/thread=%p: ", cp->cpu_id, (void *)t);
361                 vprintf(format, alist);
362                 printf("\n");
363         } else
364                 goto spin;
366         /*
367          * Prior to performing sync or dump, we make sure that do_polled_io is
368          * set, but we'll leave ipl at 10; deadman(), a CY_HIGH_LEVEL cyclic,
369          * will re-enter panic if we are not making progress with sync or dump.
370          */
372         /*
373          * Sync the filesystems.  Reset t_cred if not set because much of
374          * the filesystem code depends on CRED() being valid.
375          */
376         if (!in_sync && panic_trigger(&panic_sync)) {
377                 if (t->t_cred == NULL)
378                         t->t_cred = kcred;
379                 splx(ipltospl(CLOCK_LEVEL));
380                 do_polled_io = 1;
381                 vfs_syncall();
382         }
384         /*
385          * Take the crash dump.  If the dump trigger is already set, try to
386          * enter the debugger again before rebooting the system.
387          */
388         if (panic_trigger(&panic_dump)) {
389                 panic_dump_hw(s);
390                 splx(ipltospl(CLOCK_LEVEL));
```

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

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

*unchanged_portion_omitted_*

```
**********************************************************
    2754 Fri Mar 28 23:33:32 2014
new/usr/src/uts/common/os/sched.c
patch delete-swapped_lock
patch sched-cleanup
patch remove-useless-var2
patch remove-dead-sched-code
patch remove-as_swapout
patch remove-load-flag
patch remove-on-swapq-flag
patch remove-swapenq-flag
patch remove-dont-swap-flag
patch remove-swapinout-class-ops
patch remove-useless-var
**********************************************************
```

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

/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

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

#include <sys/param.h>
#include <sys/types.h>
#include <sys/sysmacros.h>
#include <sys/systm.h>
#include <sys/proc.h>
#include <sys/cpuvar.h>
#include <sys/var.h>
#include <sys/tuneable.h>
#include <sys/cmn_err.h>
#include <sys/buf.h>
#include <sys/disp.h>
#include <sys/vmsystm.h>
#include <sys/vmparam.h>
#include <sys/class.h>
#include <sys/vtrace.h>
#include <sys/modctl.h>
#include <sys/debug.h>
#include <sys/tnf_probe.h>
#include <sys/procfs.h>

#include <vm/seg.h>
#include <vm/seg_kp.h>
```



```
#include <vm/as.h>
#include <vm/rm.h>
#include <vm/seg_kmem.h>
#include <sys/callb.h>

/*
 * The swapper sleeps on runout when there is no one to swap in.
 * It sleeps on runin when it could not find space to swap someone
 * in or after swapping someone in.
 */
char    runout;
char    runin;
char    wake_sched;     /* flag tells clock to wake swapper on next tick */
char    wake_sched_sec; /* flag tells clock to wake swapper after a second */

/*
 * The swapper swaps processes to reduce memory demand and runs
 * when avefree < desfree.  The swapper resorts to SOFTSWAP when
 * avefree < desfree which results in swapping out all processes
 * sleeping for more than maxslp seconds.  HARDSWAP occurs when the
 * system is on the verge of thrashing and this results in swapping
 * out runnable threads or threads sleeping for less than maxslp secs.
 *
 * The swapper runs through all the active processes in the system
 * and invokes the scheduling class specific swapin/swapout routine
 * for every thread in the process to obtain an effective priority
 * for the process.  A priority of -1 implies that the thread isn't
 * swappable.  This effective priority is used to find the most
 * eligible process to swapout or swapin.
 *
 * NOTE:  Threads which have been swapped are not linked on any
 *        queue and their dispatcher lock points at the "swapped_lock".
 *
 * Processes containing threads with the TS_DONT_SWAP flag set cannot be
 * swapped out immediately by the swapper.  This is due to the fact that
 * such threads may be holding locks which may be needed by the swapper
 * to push its pages out.  The TS_SWAPENQ flag is set on such threads
 * to prevent them running in user mode.  When such threads reach a
 * safe point (i.e., are not holding any locks - CL_TRAPRET), they
 * queue themseleves onto the swap queue which is processed by the
 * swapper.  This results in reducing memory demand when the system
 * is desparate for memory as the thread can't run in user mode.
 *
 * The swap queue consists of threads, linked via t_link, which are
 * haven't been swapped, are runnable but not on the run queue.  The
 * swap queue is protected by the "swapped_lock".  The dispatcher
 * lock (t_lockp) of all threads on the swap queue points at the
 * "swapped_lock".  Thus, the entire queue and/or threads on the
 * queue can be locked by acquiring "swapped_lock".
 */
static kthread_t *tswap_queue;
extern disp_lock_t swapped_lock; /* protects swap queue and threads on it */

int     maxslp = 0;
pgcnt_t avefree;        /* 5 sec moving average of free memory */
pgcnt_t avefree30;      /* 30 sec moving average of free memory */

/*
 * Minimum size used to decide if sufficient memory is available
 * before a process is swapped in.  This is necessary since in most
 * cases the actual size of a process (p_swrss) being swapped in
 * is usually 2 pages (kernel stack pages).  This is due to the fact
 * almost all user pages of a process are stolen by pageout before
 * the swapper decides to swapout it out.
 */
int     min_procsize = 12;
```

```
119 static int      swapin(proc_t *);
120 static int      swapout(proc_t *, uint_t *, int);
121 static void     process_swap_queue();

123 #ifdef __sparc
124 extern void lwp_swapin(kthread_t *);
125 #endif /* __sparc */

127 /*
128  * Counters to keep track of the number of swapins or swapouts.
129  */
130 uint_t tot_swapped_in, tot_swapped_out;
131 uint_t softswap, hardswap, swapqswap;

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

141 /*
 61  * Memory scheduler.
 62  */
 63 void
 64 sched()
 65 {
 66         kthread_id_t    t;
148         pri_t           proc_pri;
149         pri_t           thread_pri;
150         pri_t           swapin_pri;
151         int             desperate;
152         pgcnt_t         needs;
153         int             divisor;
154         proc_t          *prp;
155         proc_t          *swapout_prp;
156         proc_t          *swapin_prp;
157         spgcnt_t        avail;
158         int             chosen_pri;
159         time_t          swapout_time;
160         time_t          swapin_proc_time;
 67         callb_cpr_t     cprinfo;
 68         kmutex_t        swap_cpr_lock;

 70         mutex_init(&swap_cpr_lock, NULL, MUTEX_DEFAULT, NULL);
 71         CALLB_CPR_INIT(&cprinfo, &swap_cpr_lock, callb_generic_cpr, "sched");
166         if (maxslp == 0)
167                 maxslp = MAXSLP;
168 loop:
169         needs = 0;
170         desperate = 0;

 73         for (;;) {
172         swapin_pri = v.v_nglobpris;
173         swapin_prp = NULL;
174         chosen_pri = -1;

176         process_swap_queue();

178         /*
179          * Set desperate if
180          *      1.  At least 2 runnable processes (on average).
181          *      2.  Short (5 sec) and longer (30 sec) average is less
182          *          than minfree and desfree respectively.
```



```
183          *      3.  Pagein + pageout rate is excessive.
184          */
 74                 if (avenrun[0] >= 2 * FSCALE &&
 75                     (MAX(avefree, avefree30) < desfree) &&
 76                     (pginrate + pgoutrate > maxpgio || avefree < minfree)) {
188                 TRACE_4(TR_FAC_SCHED, TR_DESPERATE,
189                     "desp:avefree: %d, avefree30: %d, freemem: %d"
190                     " pginrate: %d\n", avefree, avefree30, freemem, pginrate);
191                 desperate = 1;
192                 goto unload;
193         }

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

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

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

223                                 thread_lock(t);
224                                 thread_pri = CL_SWAPIN(t, 0);
225                                 thread_unlock(t);

227                                 if (t->t_stime - proc_time > 0)
228                                         proc_time = t->t_stime;
229                                 if (thread_pri > proc_pri)
230                                         proc_pri = thread_pri;
231                         } while ((t = t->t_forw) != prp->p_tlist);
232                         mutex_exit(&prp->p_lock);

234                         if (proc_pri == -1)
235                                 continue;

237                         TRACE_3(TR_FAC_SCHED, TR_CHOOSE_SWAPIN,
238                             "prp %p epri %d proc_time %d",
239                             prp, proc_pri, proc_time);

241                         /*
242                          * Swapin processes with a high effective priority.
243                          */
244                         if (swapin_prp == NULL || proc_pri > chosen_pri) {
245                                 swapin_prp = prp;
246                                 chosen_pri = proc_pri;
247                                 swapin_pri = proc_pri;
248                                 swapin_proc_time = proc_time;
```

```
249                     }
250             } else {
251                     /*
252                      * No need to soft swap if we have sufficient
253                      * memory.
254                      */
255                     if (avefree > desfree ||
256                         avefree < desfree && freemem > desfree)
257                             continue;

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                     }

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;

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);

287                     if (proc_pri != -1) {
288                             uint_t swrss;

290                             mutex_exit(&pidlock);

292                             TRACE_1(TR_FAC_SCHED, TR_SOFTSWAP,
293                                 "softswap:prp %p", prp);

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);

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);

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     }

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;

343     needs = MIN(swapin_prp->p_swrss, lotsfree);
344     needs = MAX(needs, min_procsize);
345     needs = needs / divisor;

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;

355             TRACE_2(TR_FAC_SCHED, TR_SWAPIN_VALUES,
356                 "swapin_values: prp %p needs %lu", swapin_prp, needs);

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     }

378     TRACE_2(TR_FAC_SCHED, TR_UNLOAD,
379         "unload: prp %p needs %lu", swapin_prp, needs);
```

```
unload:
                        /*
                         * Unload all unloadable modules, free all other memory
                         * resources we can find, then look for a thread to
                         * hardswap.
         * resources we can find, then look for a thread to hardswap.
                         */
                        modreap();
                        segkp_cache_free();

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

                /*
                 * No need to soft swap if we have sufficient
                 * memory.
                 */
                if (not_swappable(prp))
                        continue;

                if (avefree > minfree ||
                    avefree < minfree && freemem > desfree) {
                        swapout_prp = NULL;
                        break;
                }

                /*
                 * Skip processes that are exiting
                 * or whose address spaces are locked.
                 */
                mutex_enter(&prp->p_lock);
                if ((prp->p_flag & SEXITING) ||
                    (prp->p_as != NULL && AS_ISPGLCK(prp->p_as))) {
                        mutex_exit(&prp->p_lock);
                        continue;
                }

                proc_pri = -1;
                t = prp->p_tlist;
                do {
                        if ((t->t_schedflag & (TS_SWAPENQ |
                            TS_ON_SWAPQ | TS_LOAD)) != TS_LOAD)
                                continue;

                        thread_lock(t);
                        thread_pri = CL_SWAPOUT(t, HARDSWAP);
                        thread_unlock(t);
                        if (thread_pri > proc_pri)
                                proc_pri = thread_pri;
                } while ((t = t->t_forw) != prp->p_tlist);

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

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

                        if (swapout_prp == NULL || proc_pri < chosen_pri) {
                                swapout_prp = prp;
```



```
                                chosen_pri = proc_pri;
                        }
                }
        }

        /*
         * Acquire the "p_lock" before dropping "pidlock"
         * to prevent the proc structure from being freed
         * if the process exits before swapout completes.
         */
        if (swapout_prp != NULL)
                mutex_enter(&swapout_prp->p_lock);
        mutex_exit(&pidlock);

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

                swapped = swapout(prp, &swrss, HARDSWAP);
                if (swapped) {
                        /*
                         * If desperate, we want to give the space obtained
                         * by swapping this process out to processes in core,
                         * so we give them a chance by increasing deficit.
                         */
                        prp->p_swrss += swrss;
                        if (desperate)
                                deficit += MIN(prp->p_swrss, lotsfree);
                        hardswap++;
                }
                mutex_exit(&swapout_prp->p_lock);

                if (swapped)
                        goto loop;
                }

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

block:
                t = curthread;
                thread_lock(t);
        runin++;
                t->t_schedflag |= (TS_ALLSTART & ~TS_CSTART);
                t->t_whystop = PR_SUSPENDED;
                t->t_whatstop = SUSPEND_NORMAL;
                (void) new_mstate(t, LMS_SLEEP);
                mutex_enter(&swap_cpr_lock);
                CALLB_CPR_SAFE_BEGIN(&cprinfo);
                mutex_exit(&swap_cpr_lock);
                thread_stop(t);         /* change to stop state and drop lock */
                swtch();
                mutex_enter(&swap_cpr_lock);
                CALLB_CPR_SAFE_END(&cprinfo, &swap_cpr_lock);
                mutex_exit(&swap_cpr_lock);
        goto loop;
}

/*
 * Remove the specified thread from the swap queue.
 */
static void
```

```
swapdeq(kthread_id_t tp)
{
        kthread_id_t *tpp;

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

        tpp = &tswap_queue;
        for (;;) {
                ASSERT(*tpp != NULL);
                if (*tpp == tp)
                        break;
                tpp = &(*tpp)->t_link;
        }
        *tpp = tp->t_link;
        tp->t_schedflag &= ~TS_ON_SWAPQ;
}

/*
 * Swap in lwps.  Returns nonzero on success (i.e., if at least one lwp is
 * swapped in) and 0 on failure.
 */
static int
swapin(proc_t *pp)
{
        kthread_id_t tp;
        int err;
        int num_swapped_in = 0;
        struct cpu *cpup = CPU;
        pri_t thread_pri;

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

top:
        tp = pp->p_tlist;
        do {
                /*
                 * Only swapin eligible lwps (specified by the scheduling
                 * class) which are unloaded and ready to run.
                 */
                thread_lock(tp);
                thread_pri = CL_SWAPIN(tp, 0);
                if (thread_pri != -1 && tp->t_state == TS_RUN &&
                    (tp->t_schedflag & TS_LOAD) == 0) {
                        size_t stack_size;
                        pgcnt_t stack_pages;

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

                        thread_unlock(tp);
                        /*
                         * Now drop the p_lock since the stack needs
                         * to brought in.
                         */
                        mutex_exit(&pp->p_lock);

                        stack_size = swapsize(tp->t_swap);
                        stack_pages = btopr(stack_size);
                        /* Kernel probe */
                        TNF_PROBE_4(swapin_lwp, "vm swap swapin", /* CSTYLED */,
                            tnf_pid,            pid,            pp->p_pid,
                            tnf_lwpid,          lwpid,          tp->t_tid,
                            tnf_kthread_id,     tid,            tp,
                            tnf_ulong,          page_count,     stack_pages);
```



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

                        /*
                         * Re-acquire the p_lock.
                         */
                        mutex_enter(&pp->p_lock);
                        if (err) {
                                num_swapped_in = 0;
                                break;
                        } else {
#ifdef __sparc
                                lwp_swapin(tp);
#endif /* __sparc */
                                CPU_STATS_ADDQ(cpup, vm, swapin, 1);
                                CPU_STATS_ADDQ(cpup, vm, pgswapin,
                                    stack_pages);

                                pp->p_swapcnt--;
                                pp->p_swrss -= stack_pages;

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

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

                                nswapped--;
                                tot_swapped_in++;
                                num_swapped_in++;

                                TRACE_2(TR_FAC_SCHED, TR_SWAPIN,
                                    "swapin: pp %p stack_pages %lu",
                                    pp, stack_pages);
                                goto top;
                        }
                }
                thread_unlock(tp);
        } while ((tp = tp->t_forw) != pp->p_tlist);
        return (num_swapped_in);
}

/*
 * Swap out lwps.  Returns nonzero on success (i.e., if at least one lwp is
 * swapped out) and 0 on failure.
 */
static int
swapout(proc_t *pp, uint_t *swrss, int swapflags)
{
        kthread_id_t tp;
        pgcnt_t ws_pages = 0;
        int err;
        int swapped_lwps = 0;
        struct as *as = pp->p_as;
        struct cpu *cpup = CPU;
        pri_t thread_pri;

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

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

```
643 top:
644         tp = pp->p_tlist;
645         do {
646                 klwp_t *lwp = ttolwp(tp);
647 
648                 /*
649                  * Swapout eligible lwps (specified by the scheduling
650                  * class) which don't have TS_DONT_SWAP set.  Set the
651                  * "intent to swap" flag (TS_SWAPENQ) on threads
652                  * which have TS_DONT_SWAP set so that they can be
653                  * swapped if and when they reach a safe point.
654                  */
655                 thread_lock(tp);
656                 thread_pri = CL_SWAPOUT(tp, swapflags);
657                 if (thread_pri != -1) {
658                         if (tp->t_schedflag & TS_DONT_SWAP) {
659                                 tp->t_schedflag |= TS_SWAPENQ;
660                                 tp->t_trapret = 1;
661                                 aston(tp);
662                         } else {
663                                 pgcnt_t stack_pages;
664                                 size_t stack_size;
665 
666                                 ASSERT((tp->t_schedflag &
667                                     (TS_DONT_SWAP | TS_LOAD)) == TS_LOAD);
668 
669                                 if (lock_try(&tp->t_lock)) {
670                                         /*
671                                          * Remove thread from the swap_queue.
672                                          */
673                                         if (tp->t_schedflag & TS_ON_SWAPQ) {
674                                                 ASSERT(!(tp->t_schedflag &
675                                                     TS_SWAPENQ));
676                                                 swapdeq(tp);
677                                         } else if (tp->t_state == TS_RUN)
678                                                 dq_srundec(tp);
679 
680                                         tp->t_schedflag &=
681                                             ~(TS_LOAD | TS_SWAPENQ);
682                                         lock_clear(&tp->t_lock);
683 
684                                         /*
685                                          * Set swapout time if the thread isn't
686                                          * sleeping.
687                                          */
688                                         if (tp->t_state != TS_SLEEP)
689                                                 tp->t_stime = ddi_get_lbolt();
690                                         thread_unlock(tp);
691 
692                                         nswapped++;
693                                         tot_swapped_out++;
694 
695                                         lwp->lwp_ru.nswap++;
696 
697                                         /*
698                                          * Now drop the p_lock since the
699                                          * stack needs to pushed out.
700                                          */
701                                         mutex_exit(&pp->p_lock);
702 
703                                         stack_size = swapsize(tp->t_swap);
704                                         stack_pages = btopr(stack_size);
705                                         ws_pages += stack_pages;
706                                         /* Kernel probe */
707                                         TNF_PROBE_4(swapout_lwp,
708                                             "vm swap swapout",
```



```
709                                             /* CSTYLED */,
710                                             tnf_pid, pid, pp->p_pid,
711                                             tnf_lwpid, lwpid, tp->t_tid,
712                                             tnf_kthread_id, tid, tp,
713                                             tnf_ulong, page_count,
714                                             stack_pages);
715 
716                                         rw_enter(&kas.a_lock, RW_READER);
717                                         err = segkp_fault(segkp->s_as->a_hat,
718                                             segkp, tp->t_swap, stack_size,
719                                             F_SOFTUNLOCK, S_WRITE);
720                                         rw_exit(&kas.a_lock);
721 
722                                         if (err) {
723                                                 cmn_err(CE_PANIC,
724                                                     "swapout: segkp_fault "
725                                                     "failed err: %d", err);
726                                         }
727                                         CPU_STATS_ADDQ(cpup,
728                                             vm, pgswapout, stack_pages);
729 
730                                         mutex_enter(&pp->p_lock);
731                                         pp->p_swapcnt++;
732                                         swapped_lwps++;
733                                         goto top;
734                                 }
735                         }
736                 }
737                 thread_unlock(tp);
738         } while ((tp = tp->t_forw) != pp->p_tlist);
739 
740         /*
741          * Unload address space when all lwps are swapped out.
742          */
743         if (pp->p_swapcnt == pp->p_lwpcnt) {
744                 size_t as_size = 0;
745 
746                 /*
747                  * Avoid invoking as_swapout() if the process has
748                  * no MMU resources since pageout will eventually
749                  * steal pages belonging to this address space.  This
750                  * saves CPU cycles as the number of pages that are
751                  * potentially freed or pushed out by the segment
752                  * swapout operation is very small.
753                  */
754                 if (rm_asrss(pp->p_as) != 0)
755                         as_size = as_swapout(as);
756 
757                 CPU_STATS_ADDQ(cpup, vm, pgswapout, btop(as_size));
758                 CPU_STATS_ADDQ(cpup, vm, swapout, 1);
759                 ws_pages += btop(as_size);
760 
761                 TRACE_2(TR_FAC_SCHED, TR_SWAPOUT,
762                     "swapout: pp %p pages_pushed %lu", pp, ws_pages);
763                 /* Kernel probe */
764                 TNF_PROBE_2(swapout_process, "vm swap swapout", /* CSTYLED */,
765                     tnf_pid,    pid,            pp->p_pid,
766                     tnf_ulong,  page_count,     ws_pages);
767         }
768         *swrss = ws_pages;
769         return (swapped_lwps);
770 }
771 
772 void
773 swapout_lwp(klwp_t *lwp)
774 {
```

```
775         kthread_id_t tp = curthread;

777         ASSERT(curthread == lwptot(lwp));

779         /*
780          * Don't insert the thread onto the swap queue if
781          * sufficient memory is available.
782          */
783         if (avefree > desfree || avefree < desfree && freemem > desfree) {
784                 thread_lock(tp);
785                 tp->t_schedflag &= ~TS_SWAPENQ;
786                 thread_unlock(tp);
787                 return;
788         }

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

_____***unchanged_portion_omitted_***

```
**********************************************************
   39428 Fri Mar 28 23:33:34 2014
new/usr/src/uts/common/os/timers.c
patch remove-load-flag
**********************************************************
______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 Fri Mar 28 23:33:35 2014
new/usr/src/uts/common/os/waitq.c
patch remove-dont-swap-flag
****
_____***unchanged_portion_omitted_***

```
 197 /*
 198  * Put specified thread to specified wait queue without dropping thread's lock.
 199  * Returns 1 if thread was successfully placed on project's wait queue, or
 200  * 0 if wait queue is blocked.
 201  */
 202 int
 203 waitq_enqueue(waitq_t *wq, kthread_t *t)
 204 {
 205         ASSERT(THREAD_LOCK_HELD(t));
 206         ASSERT(t->t_sleepq == NULL);
 207         ASSERT(t->t_waitq == NULL);
 208         ASSERT(t->t_link == NULL);

 210         disp_lock_enter_high(&wq->wq_lock);

 212         /*
 213          * Can't enqueue anything on a blocked wait queue
 214          */
 215         if (wq->wq_blocked) {
 216                 disp_lock_exit_high(&wq->wq_lock);
 217                 return (0);
 218         }

 220         /*
 221          * Mark the time when thread is placed on wait queue. The microstate
 222          * accounting code uses this timestamp to determine wait times.
 223          */
 224         t->t_waitrq = gethrtime_unscaled();

 226         /*
 227          * Mark thread as not swappable.  If necessary, it will get
 228          * swapped out when it returns to the userland.
 229          */
 230         t->t_schedflag |= TS_DONT_SWAP;
 226         DTRACE_SCHED1(cpucaps__sleep, kthread_t *, t);
 227         waitq_link(wq, t);

 229         THREAD_WAIT(t, &wq->wq_lock);
 230         return (1);
 231 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    7397 Fri Mar 28 23:33:36 2014
new/usr/src/uts/common/sys/class.h
patch remove-swapinout-class-ops
**********************************************************
______unchanged_portion_omitted_

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

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

 116 #ifdef  _KERNEL

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

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

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

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

 141 #endif
```



```
 143 #define CL_ADMIN(clp, uaddr, reqpcredp) \
 144         (*(clp)->cl_funcs->sclass.cl_admin)(uaddr, reqpcredp)

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

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

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

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

 157 #define CL_FORKRET(t, ct)       (*(t)->t_clfuncs->cl_forkret)(t, ct)

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

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

 165 #define CL_PARMSGET(t, clparmsp) \
 166         (*(t)->t_clfuncs->cl_parmsget)(t, (void *)clparmsp)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

 203 #define CL_DONICE(t, cr, inc, ret) \
```

```
 204         (*(t)->t_clfuncs->cl_donice)(t, cr, inc, ret)

 206 #define CL_DOPRIO(t, cr, inc, ret) \
 207         (*(t)->t_clfuncs->cl_doprio)(t, cr, inc, ret)

 209 #define CL_GLOBPRI(t)           (*(t)->t_clfuncs->cl_globpri)(t)

 211 #define CL_SET_PROCESS_GROUP(t, s, b, f) \
 212         (*(t)->t_clfuncs->cl_set_process_group)(s, b, f)

 214 #define CL_YIELD(tp)            (*(tp)->t_clfuncs->cl_yield)(tp)

 216 #define CL_ALLOC(pp, cid, flag) \
 217         (sclass[cid].cl_funcs->sclass.cl_alloc) (pp, flag)

 219 #define CL_FREE(cid, bufp)      (sclass[cid].cl_funcs->sclass.cl_free) (bufp)

 221 #ifdef  __cplusplus
 222 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    5777 Fri Mar 28 23:33:38 2014
new/usr/src/uts/common/sys/disp.h
patch remove-load-flag
**********************************************************
______unchanged_portion_omitted_

  82 #if defined(_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 /*
 100  * Minimum amount of time that a thread can remain runnable before it can
 101  * be stolen by another CPU (in nanoseconds).
 102  */
 103 extern hrtime_t nosteal_nsec;

 105 /*
 106  * Kernel preemption occurs if a higher-priority thread is runnable with
 107  * a priority at or above kpreemptpri.
 108  *
 109  * So that other processors can watch for such threads, a separate
 110  * dispatch queue with unbound work above kpreemptpri is maintained.
 111  * This is part of the CPU partition structure (cpupart_t).
 112  */
 113 extern  pri_t   kpreemptpri;    /* level above which preemption takes place */

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

 118 /*
 119  * Macro for use by scheduling classes to decide whether the thread is about
 120  * to be scheduled or not.  This returns the maximum run priority.
 121  */
 122 #define DISP_MAXRUNPRI(t)       ((t)->t_disp_queue->disp_maxrunpri)

 124 /*
 125  * Platform callbacks for various dispatcher operations
 126  *
 127  * idle_cpu() is invoked when a cpu goes idle, and has nothing to do.
 128  * disp_enq_thread() is invoked when a thread is placed on a run queue.
 129  */
 130 extern void     (*idle_cpu)();
 131 extern void     (*disp_enq_thread)(struct cpu *, int);


 134 extern int              dispdeq(kthread_t *);
 135 extern void             dispinit(void);
 136 extern void             disp_add(sclass_t *);
 137 extern int              intr_active(struct cpu *, int);
 138 extern int              servicing_interrupt(void);
 139 extern void             preempt(void);
 140 extern void             setbackdq(kthread_t *);
```



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

 165 #define KPREEMPT_SYNC           (-1)
 166 #define kpreempt_disable()                                      \
 167         {                                                      \
 168                 curthread->t_preempt++;                        \
 169                 ASSERT(curthread->t_preempt >= 1);             \
 170         }
 171 #define kpreempt_enable()                                       \
 172         {                                                      \
 173                 ASSERT(curthread->t_preempt >= 1);             \
 174                 if (--curthread->t_preempt == 0 &&             \
 175                     CPU->cpu_kprunrun)                         \
 176                         kpreempt(KPREEMPT_SYNC);               \
 177         }

 179 #endif  /* _KERNEL */

 181 #ifdef  __cplusplus
 182 }
______unchanged_portion_omitted_
```

```
**********************************************************
   29295 Fri Mar 28 23:33:39 2014
new/usr/src/uts/common/sys/proc.h
patch remove-as_swapout
**********************************************************
______unchanged_portion_omitted_

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

 131 /*
 132  * One structure allocated per active process.  Per-process data (user.h) is
 133  * also inside the proc structure.
 132  * One structure allocated per active process.  It contains all
 133  * data needed about the process while the process may be swapped
 134  * out.  Other per-process data (user.h) is also inside the proc structure.
 135  * Lightweight-process data (lwp.h) and the kernel stack may be swapped out.
 134  */
 135 typedef struct  proc {
 136         /*
 137          * Fields requiring no explicit locking
 138          */
 139         struct  vnode *p_exec;          /* pointer to a.out vnode */
 140         struct  as *p_as;               /* process address space pointer */
 141         struct  plock *p_lockp;         /* ptr to proc struct's mutex lock */
 142         kmutex_t p_crlock;              /* lock for p_cred */
 143         struct  cred    *p_cred;        /* process credentials */
 144         /*
 145          * Fields protected by pidlock
 146          */
 149         int     p_swapcnt;              /* number of swapped out lwps */
 147         char    p_stat;                 /* status of process */
 148         char    p_wcode;                /* current wait code */
 149         ushort_t p_pidflag;             /* flags protected only by pidlock */
 150         int     p_wdata;                /* current wait return value */
 151         pid_t   p_ppid;                 /* process id of parent */
 152         struct  proc    *p_link;        /* forward link */
 153         struct  proc    *p_parent;      /* ptr to parent process */
 154         struct  proc    *p_child;       /* ptr to first child process */
 155         struct  proc    *p_sibling;     /* ptr to next sibling proc on chain */
 156         struct  proc    *p_psibling;    /* ptr to prev sibling proc on chain */
 157         struct  proc    *p_sibling_ns;  /* prt to siblings with new state */
 158         struct  proc    *p_child_ns;    /* prt to children with new state */
 159         struct  proc    *p_next;        /* active chain link next */
 160         struct  proc    *p_prev;        /* active chain link prev */
 161         struct  proc    *p_nextofkin;   /* gets accounting info at exit */
 162         struct  proc    *p_orphan;
 163         struct  proc    *p_nextorph;
 164         struct  proc    *p_pglink;      /* process group hash chain link next */
 165         struct  proc    *p_ppglink;     /* process group hash chain link prev */
 166         struct  sess    *p_sessp;       /* session information */
 167         struct  pid     *p_pidp;        /* process ID info */
 168         struct  pid     *p_pgidp;       /* process group ID info */
 169         /*
 170          * Fields protected by p_lock
 171          */
 172         kcondvar_t p_cv;                /* proc struct's condition variable */
 173         kcondvar_t p_flag_cv;
 174         kcondvar_t p_lwpexit;           /* waiting for some lwp to exit */
 175         kcondvar_t p_holdlwps;          /* process is waiting for its lwps */
 176                                         /* to to be held.  */
 177         uint_t  p_proc_flag;            /* /proc-related flags */
```



```
 178         uint_t  p_flag;                 /* protected while set. */
 179                                         /* flags defined below */
 180         clock_t p_utime;                /* user time, this process */
 181         clock_t p_stime;                /* system time, this process */
 182         clock_t p_cutime;               /* sum of children's user time */
 183         clock_t p_cstime;               /* sum of children's system time */
 184         avl_tree_t *p_segacct;          /* System V shared segment list */
 185         avl_tree_t *p_semacct;          /* System V semaphore undo list */
 186         caddr_t p_bssbase;              /* base addr of last bss below heap */
 187         caddr_t p_brkbase;              /* base addr of heap */
 188         size_t  p_brksize;              /* heap size in bytes */
 189         uint_t  p_brkpageszc;           /* preferred heap max page size code */
 190         /*
 191          * Per process signal stuff.
 192          */
 193         k_sigset_t p_sig;               /* signals pending to this process */
 194         k_sigset_t p_extsig;            /* signals sent from another contract */
 195         k_sigset_t p_ignore;            /* ignore when generated */
 196         k_sigset_t p_siginfo;           /* gets signal info with signal */
 197         struct sigqueue *p_sigqueue;    /* queued siginfo structures */
 198         struct sigqhdr *p_sigqhdr;      /* hdr to sigqueue structure pool */
 199         struct sigqhdr *p_signhdr;      /* hdr to signotify structure pool */
 200         uchar_t p_stopsig;              /* jobcontrol stop signal */

 202         /*
 203          * Special per-process flag when set will fix misaligned memory
 204          * references.
 205          */
 206         char    p_fixalignment;

 208         /*
 209          * Per process lwp and kernel thread stuff
 210          */
 211         id_t    p_lwpid;                /* most recently allocated lwpid */
 212         int     p_lwpcnt;               /* number of lwps in this process */
 213         int     p_lwprcnt;              /* number of not stopped lwps */
 214         int     p_lwpdaemon;            /* number of TP_DAEMON lwps */
 215         int     p_lwpwait;              /* number of lwps in lwp_wait() */
 216         int     p_lwpdwait;             /* number of daemons in lwp_wait() */
 217         int     p_zombcnt;              /* number of zombie lwps */
 218         kthread_t *p_tlist;             /* circular list of threads */
 219         lwpdir_t *p_lwpdir;             /* thread (lwp) directory */
 220         lwpdir_t *p_lwpfree;            /* p_lwpdir free list */
 221         tidhash_t *p_tidhash;           /* tid (lwpid) lookup hash table */
 222         uint_t  p_lwpdir_sz;            /* number of p_lwpdir[] entries */
 223         uint_t  p_tidhash_sz;           /* number of p_tidhash[] entries */
 224         ret_tidhash_t *p_ret_tidhash;   /* retired tidhash hash tables */
 225         uint64_t p_lgrpset;             /* unprotected hint of set of lgrps */
 226                                         /* on which process has threads */
 227         volatile lgrp_id_t  p_t1_lgrpid; /* main's thread lgroup id */
 228         volatile lgrp_id_t  p_tr_lgrpid; /* text replica's lgroup id */
 229 #if defined(_LP64)
 230         uintptr_t  p_lgrpres2;          /* reserved for lgrp migration */
 231 #endif
 232         /*
 233          * /proc (process filesystem) debugger interface stuff.
 234          */
 235         k_sigset_t p_sigmask;           /* mask of traced signals (/proc) */
 236         k_fltset_t p_fltmask;           /* mask of traced faults (/proc) */
 237         struct vnode *p_trace;          /* pointer to primary /proc vnode */
 238         struct vnode *p_plist;          /* list of /proc vnodes for process */
 239         kthread_t *p_agenttp;           /* thread ptr for /proc agent lwp */
 240         avl_tree_t p_warea;             /* list of watched areas */
 241         avl_tree_t p_wpage;             /* remembered watched pages (vfork) */
 242         watched_page_t *p_wprot;        /* pages that need to have prot set */
 243         int     p_mapcnt;               /* number of active pr_mappage()s */
```

```
244         kmutex_t p_maplock;             /* lock for pr_mappage() */
245         struct  proc  *p_rlink;         /* linked list for server */
246         kcondvar_t p_srwchan_cv;
247         size_t  p_stksize;              /* process stack size in bytes */
248         uint_t  p_stkpageszc;           /* preferred stack max page size code */

250         /*
251          * Microstate accounting, resource usage, and real-time profiling
252          */
253         hrtime_t p_mstart;              /* hi-res process start time */
254         hrtime_t p_mterm;               /* hi-res process termination time */
255         hrtime_t p_mlreal;              /* elapsed time sum over defunct lwps */
256         hrtime_t p_acct[NMSTATES];      /* microstate sum over defunct lwps */
257         hrtime_t p_cacct[NMSTATES];     /* microstate sum over child procs */
258         struct lrusage p_ru;            /* lrusage sum over defunct lwps */
259         struct lrusage p_cru;           /* lrusage sum over child procs */
260         struct itimerval p_rprof_timer; /* ITIMER_REALPROF interval timer */
261         uintptr_t p_rprof_cyclic;       /* ITIMER_REALPROF cyclic */
262         uint_t  p_defunct;              /* number of defunct lwps */
263         /*
264          * profiling. A lock is used in the event of multiple lwp's
265          * using the same profiling base/size.
266          */
267         kmutex_t p_pflock;              /* protects user profile arguments */
268         struct prof p_prof;             /* profile arguments */

270         /*
271          * Doors.
272          */
273         door_pool_t             p_server_threads; /* common thread pool */
274         struct door_node        *p_door_list;   /* active doors */
275         struct door_node        *p_unref_list;
276         kcondvar_t              p_unref_cv;
277         char                    p_unref_thread; /* unref thread created */

279         /*
280          * Kernel probes
281          */
282         uchar_t                 p_tnf_flags;

284         /*
285          * Solaris Audit
286          */
287         struct p_audit_data     *p_audit_data; /* per process audit structure */

289         pctxop_t        *p_pctx;

291 #if defined(__x86)
292         /*
293          * LDT support.
294          */
295         kmutex_t        p_ldtlock;      /* protects the following fields */
296         user_desc_t     *p_ldt;         /* Pointer to private LDT */
297         system_desc_t   p_ldt_desc;     /* segment descriptor for private LDT */
298         ushort_t        p_ldtlimit;     /* highest selector used */
299 #endif
300         size_t p_swrss;                 /* resident set size before last swap */
301         struct aio      *p_aio;         /* pointer to async I/O struct */
302         struct itimer   **p_itimer;     /* interval timers */
303         timeout_id_t    p_alarmid;      /* alarm's timeout id */
304         caddr_t         p_usrstack;     /* top of the process stack */
305         uint_t          p_stkprot;      /* stack memory protection */
306         uint_t          p_datprot;      /* data memory protection */
307         model_t         p_model;        /* data model determined at exec time */
308         struct lwpchan_data *p_lcp;     /* lwpchan cache */
309         kmutex_t        p_lcp_lock;     /* protects assignments to p_lcp */
```



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

327         avl_tree_t      p_ct_held;      /* held contracts */
328         struct ct_equeue **p_ct_equeue; /* process-type event queues */

330         struct cont_process *p_ct_process; /* process contract */
331         list_node_t     p_ct_member;    /* process contract membership */
332         sigqueue_t      *p_killsqp;     /* sigqueue pointer for SIGKILL */

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

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

356         /*
357          * The user structure
358          */
359         struct user p_user;             /* (see sys/user.h) */
360 } proc_t;
```

unchanged_portion_omitted_

```
**********************************************************
   15085 Fri Mar 28 23:33:41 2014
new/usr/src/uts/common/sys/systm.h
patch clock-wakeup-remove
**********************************************************
```

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


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

  30 #ifndef _SYS_SYSTM_H
  31 #define _SYS_SYSTM_H

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

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

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

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

  57 #ifdef _KERNEL
  58 #include <sys/varargs.h>
  59 #include <sys/uadmin.h>

  61 extern int hz;                  /* system clock rate */
```



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

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

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

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

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

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

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

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

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

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

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

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

 105 /*
 106  * Use NFS client operations in the global zone only.  Under contract with
 107  * admin/install; do not change without coordinating with that consolidation.
 108  */
 109 extern int nfs_global_client_only;

 111 extern void report_stack_exec(proc_t *, caddr_t);

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

 120 /*
 121  * for tod fault detection
 122  */
```

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

_____***unchanged_portion_omitted_***

```
**********************************************************
   26128 Fri Mar 28 23:33:42 2014
new/usr/src/uts/common/sys/thread.h
patch delete-t_stime
patch delete-swapped_lock
patch remove-load-flag
patch remove-on-swapq-flag
patch remove-swapenq-flag
patch remove-dont-swap-flag
**********************************************************
______unchanged_portion_omitted_

  96 typedef struct _kthread *kthread_id_t;

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

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

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

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



```
 150         void            *t_lgrp_reserv[2];      /* reserved for future */
 151         struct _kthread *t_intr; /* interrupted (pinned) thread */
 152         uint64_t        t_intr_start;   /* timestamp when time slice began */
 153         kt_did_t        t_did;  /* thread id for kernel debuggers */
 154         caddr_t t_tnf_tpdp;     /* Trace facility data pointer */
 155         struct _kcpc_ctx *t_cpc_ctx;    /* performance counter context */
 156         struct _kcpc_set *t_cpc_set;    /* set this thread has bound */

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

 188         /*
 189          * Pointer to the dispatcher lock protecting t_state and state-related
 190          * flags.  This pointer can change during waits on the lock, so
 191          * it should be grabbed only by thread_lock().
 192          */
 193         disp_lock_t     *t_lockp;       /* pointer to the dispatcher lock */
 194         ushort_t        t_oldspl;       /* spl level before dispatcher locked */
 195         volatile char   t_pre_sys;      /* pre-syscall work needed */
 196         lock_t          t_lock_flush;   /* for lock_mutex_flush() impl */
 197         struct _disp    *t_disp_queue;  /* run queue for chosen CPU */
 198         clock_t         t_disp_time;    /* last time this thread was running */
 199         uint_t          t_kpri_req;     /* kernel priority required */

 201         /*
 202          * Post-syscall / post-trap flags.
 203          *      No lock is required to set these.
 204          *      These must be cleared only by the thread itself.
 205          *
 206          *      t_astflag indicates that some post-trap processing is required,
 207          *              possibly a signal or a preemption.  The thread will not
 208          *              return to user with this set.
 209          *      t_post_sys indicates that some unusualy post-system call
 210          *              handling is required, such as an error or tracing.
 211          *      t_sig_check indicates that some condition in ISSIG() must be
 212          *              checked, but doesn't prevent returning to user.
```

```
213          *      t_post_sys_ast is a way of checking whether any of these three
214          *              flags are set.
215          */
216         union __tu {
217                 struct __ts {
218                         volatile char   _t_astflag;     /* AST requested */
219                         volatile char   _t_sig_check;   /* ISSIG required */
220                         volatile char   _t_post_sys;    /* post_syscall req */
221                         volatile char   _t_trapret;     /* call CL_TRAPRET */
222                 } _ts;
223                 volatile int    _t_post_sys_ast;        /* OR of these flags */
224         } _tu;
225 #define t_astflag       _tu._ts._t_astflag
226 #define t_sig_check     _tu._ts._t_sig_check
227 #define t_post_sys      _tu._ts._t_post_sys
228 #define t_trapret       _tu._ts._t_trapret
229 #define t_post_sys_ast  _tu._t_post_sys_ast

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

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

248         struct turnstile *t_prioinv;

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

255         struct turnstile *t_ts;

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

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

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

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

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



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

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

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

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

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

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

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

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

```
345 /*
346  * Thread flag (t_flag) definitions.
347  *      These flags must be changed only for the current thread,
348  *      and not during preemption code, since the code being
349  *      preempted could be modifying the flags.
350  *
351  *      For the most part these flags do not need locking.
352  *      The following flags will only be changed while the thread_lock is held,
353  *      to give assurrance that they are consistent with t_state:
354  *              T_WAKEABLE
355  */
356 #define T_INTR_THREAD   0x0001  /* thread is an interrupt thread */
357 #define T_WAKEABLE      0x0002  /* thread is blocked, signals enabled */
358 #define T_TOMASK        0x0004  /* use lwp_sigoldmask on return from signal */
359 #define T_TALLOCSTK     0x0008  /* thread structure allocated from stk */
360 #define T_FORKALL       0x0010  /* thread was cloned by forkall() */
361 #define T_WOULDBLOCK    0x0020  /* for lockfs */
362 #define T_DONTBLOCK     0x0040  /* for lockfs */
363 #define T_DONTPEND      0x0080  /* for lockfs */
364 #define T_SYS_PROF      0x0100  /* profiling on for duration of system call */
365 #define T_WAITCVSEM     0x0200  /* waiting for a lwp_cv or lwp_sema on sleepq */
366 #define T_WATCHPT       0x0400  /* thread undergoing a watchpoint emulation */
367 #define T_PANIC         0x0800  /* thread initiated a system panic */
368 #define T_LWPREUSE      0x1000  /* stack and LWP can be reused */
369 #define T_CAPTURING     0x2000  /* thread is in page capture logic */
370 #define T_VFPARENT      0x4000  /* thread is vfork parent, must call vfwait */
371 #define T_DONTDTRACE    0x8000  /* disable DTrace probes */

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

394 /*
395  * Thread scheduler flag (t_schedflag) definitions.
396  *      The thread must be locked via thread_lock() or equiv. to change these.
397  */
402 #define TS_LOAD         0x0001  /* thread is in memory */
403 #define TS_DONT_SWAP    0x0002  /* thread/lwp should not be swapped */
404 #define TS_SWAPENQ      0x0004  /* swap thread when it reaches a safe point */
405 #define TS_ON_SWAPQ     0x0008  /* thread is on the swap queue */
398 #define TS_SIGNALLED    0x0010  /* thread was awakened by cv_signal() */
399 #define TS_PROJWAITQ    0x0020  /* thread is on its project's waitq */
400 #define TS_ZONEWAITQ    0x0040  /* thread is on its zone's waitq */
401 #define TS_CSTART       0x0100  /* setrun() by continuelwps() */
402 #define TS_UNPAUSE      0x0200  /* setrun() by unpauselwps() */
403 #define TS_XSTART       0x0400  /* setrun() by SIGCONT */
404 #define TS_PSTART       0x0800  /* setrun() by /proc */
405 #define TS_RESUME       0x1000  /* setrun() by CPR resume process */
```



```
406 #define TS_CREATE       0x2000  /* setrun() by syslwp_create() */
407 #define TS_RUNQMATCH    0x4000  /* exact run queue balancing by setbackdq() */
408 #define TS_ALLSTART     \
409         (TS_CSTART|TS_UNPAUSE|TS_XSTART|TS_PSTART|TS_RESUME|TS_CREATE)
410 #define TS_ANYWAITQ     (TS_PROJWAITQ|TS_ZONEWAITQ)

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

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

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

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

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

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

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

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

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

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

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

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

471 /* The assigned priority of a thread */
```

```
472 #define ASSIGNED_PRIO(t)        ((t)->t_pri)

474 /*
483  * Macros to determine whether a thread can be swapped.
484  * If t_lock is held, the thread is either on a processor or being swapped.
485  */
486 #define SWAP_OK(t)      (!LOCK_HELD(&(t)->t_lock))

488 /*
475  * proctot(x)
476  *      convert a proc pointer to a thread pointer. this only works with
477  *      procs that have only one lwp.
478  *
479  * proctolwp(x)
480  *      convert a proc pointer to a lwp pointer. this only works with
481  *      procs that have only one lwp.
482  *
483  * ttolwp(x)
484  *      convert a thread pointer to its lwp pointer.
485  *
486  * ttoproc(x)
487  *      convert a thread pointer to its proc pointer.
488  *
489  * ttoproj(x)
490  *      convert a thread pointer to its project pointer.
491  *
492  * ttozone(x)
493  *      convert a thread pointer to its zone pointer.
494  *
495  * lwptot(x)
496  *      convert a lwp pointer to its thread pointer.
497  *
498  * lwptoproc(x)
499  *      convert a lwp to its proc pointer.
500  */
501 #define proctot(x)      ((x)->p_tlist)
502 #define proctolwp(x)    ((x)->p_tlist->t_lwp)
503 #define ttolwp(x)       ((x)->t_lwp)
504 #define ttoproc(x)      ((x)->t_procp)
505 #define ttoproj(x)      ((x)->t_proj)
506 #define ttozone(x)      ((x)->t_procp->p_zone)
507 #define lwptot(x)       ((x)->lwp_thread)
508 #define lwptoproc(x)    ((x)->lwp_procp)

510 #define t_pc            t_pcb.val[0]
511 #define t_sp            t_pcb.val[1]

513 #ifdef  _KERNEL

515 extern  kthread_t       *threadp(void); /* inline, returns thread pointer */
516 #define curthread       (threadp())             /* current thread pointer */
517 #define curproc         (ttoproc(curthread))    /* current process pointer */
518 #define curproj         (ttoproj(curthread))    /* current project pointer */
519 #define curzone         (curproc->p_zone)       /* current zone pointer */

521 extern  struct _kthread t0;             /* the scheduler thread */
522 extern  kmutex_t        pidlock;        /* global process lock */

524 /*
525  * thread_free_lock is used by the tick accounting thread to keep a thread
526  * from being freed while it is being examined.
527  *
528  * Thread structures are 32-byte aligned structures. That is why we use the
529  * following formula.
530  */
531 #define THREAD_FREE_BITS        10
```



```
532 #define THREAD_FREE_NUM         (1 << THREAD_FREE_BITS)
533 #define THREAD_FREE_MASK        (THREAD_FREE_NUM - 1)
534 #define THREAD_FREE_1           PTR24_LSB
535 #define THREAD_FREE_2           (PTR24_LSB + THREAD_FREE_BITS)
536 #define THREAD_FREE_SHIFT(t)    \
537         (((ulong_t)(t) >> THREAD_FREE_1) ^ ((ulong_t)(t) >> THREAD_FREE_2))
538 #define THREAD_FREE_HASH(t)     (THREAD_FREE_SHIFT(t) & THREAD_FREE_MASK)

540 typedef struct thread_free_lock {
541         kmutex_t        tf_lock;
542         uchar_t         tf_pad[64 - sizeof (kmutex_t)];
543 } thread_free_lock_t;
______unchanged_portion_omitted_

612 /*
613  * Macros to change thread state and the associated lock.
614  */
615 #define THREAD_SET_STATE(tp, state, lp) \
616                 ((tp)->t_state = state, (tp)->t_lockp = lp)

618 /*
619  * Point it at the transition lock, which is always held.
620  * The previosly held lock is dropped.
621  */
622 #define THREAD_TRANSITION(tp)   thread_transition(tp);
623 /*
624  * Set the thread's lock to be the transition lock, without dropping
625  * previosly held lock.
626  */
627 #define THREAD_TRANSITION_NOLOCK(tp)    ((tp)->t_lockp = &transition_lock)

629 /*
630  * Put thread in run state, and set the lock pointer to the dispatcher queue
631  * lock pointer provided.  This lock should be held.
632  */
633 #define THREAD_RUN(tp, lp)      THREAD_SET_STATE(tp, TS_RUN, lp)

635 /*
636  * Put thread in wait state, and set the lock pointer to the wait queue
637  * lock pointer provided.  This lock should be held.
638  */
639 #define THREAD_WAIT(tp, lp)     THREAD_SET_STATE(tp, TS_WAIT, lp)

641 /*
656  * Put thread in run state, and set the lock pointer to the dispatcher queue
657  * lock pointer provided (i.e., the "swapped_lock").  This lock should be held.
658  */
659 #define THREAD_SWAP(tp, lp)     THREAD_SET_STATE(tp, TS_RUN, lp)

661 /*
642  * Put the thread in zombie state and set the lock pointer to NULL.
643  * The NULL will catch anything that tries to lock a zombie.
644  */
645 #define THREAD_ZOMB(tp)         THREAD_SET_STATE(tp, TS_ZOMB, NULL)

647 /*
648  * Set the thread into ONPROC state, and point the lock at the CPUs
649  * lock for the onproc thread(s).  This lock should be held, so the
650  * thread deoes not become unlocked, since these stores can be reordered.
651  */
652 #define THREAD_ONPROC(tp, cpu)  \
653                 THREAD_SET_STATE(tp, TS_ONPROC, &(cpu)->cpu_thread_lock)

655 /*
656  * Set the thread into the TS_SLEEP state, and set the lock pointer to
657  * to some sleep queue's lock.  The new lock should already be held.
```

```
 658  */
 659 #define THREAD_SLEEP(tp, lp)    {                               \
 660                         disp_lock_t     *tlp;                   \
 661                         tlp = (tp)->t_lockp;                    \
 662                         THREAD_SET_STATE(tp, TS_SLEEP, lp);     \
 663                         disp_lock_exit_high(tlp);               \
 664                         }

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

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

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

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

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

*_____unchanged_portion_omitted_*

```
**********************************************************
    5125 Fri Mar 28 23:33:43 2014
new/usr/src/uts/common/sys/vmsystm.h
patch sched-cleanup
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

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

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */

#ifndef _SYS_VMSYSTM_H
#define _SYS_VMSYSTM_H

#include <sys/proc.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * Miscellaneous virtual memory subsystem variables and structures.
 */
#ifdef _KERNEL
extern pgcnt_t  freemem;        /* remaining blocks of free memory */
extern pgcnt_t  avefree;        /* 5 sec moving average of free memory */
extern pgcnt_t  avefree30;      /* 30 sec moving average of free memory */
extern pgcnt_t  deficit;        /* estimate of needs of new swapped in procs */
extern pgcnt_t  nscan;          /* number of scans in last second */
extern pgcnt_t  desscan;        /* desired pages scanned per second */
extern pgcnt_t  slowscan;
extern pgcnt_t  fastscan;
extern pgcnt_t  pushes;         /* number of pages pushed to swap device */
```



```
/* writable copies of tunables */
extern pgcnt_t  maxpgio;        /* max paging i/o per sec before start swaps */
extern pgcnt_t  lotsfree;       /* max free before clock freezes */
extern pgcnt_t  desfree;        /* minimum free pages before swapping begins */
extern pgcnt_t  minfree;        /* no of pages to try to keep free via daemon */
extern pgcnt_t  needfree;       /* no of pages currently being waited for */
extern pgcnt_t  throttlefree;   /* point at which we block PG_WAIT calls */
extern pgcnt_t  pageout_reserve; /* point at which we deny non-PG_WAIT calls */
extern pgcnt_t  pages_before_pager; /* XXX */

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

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

/*
 * Flags passed by the swapper to swapout routines of each
 * scheduling class.
 */
#define HARDSWAP        1
#define SOFTSWAP        2

/*
 * Values returned by valid_usr_range()
 */
#define RANGE_OKAY      (0)
#define RANGE_BADADDR   (1)
#define RANGE_BADPROT   (2)

/*
 * map_pgsz: temporary - subject to change.
 */
#define MAPPGSZ_VA      0x01
#define MAPPGSZ_STK     0x02
#define MAPPGSZ_HEAP    0x04
#define MAPPGSZ_ISM     0x08

/*
 * Flags for map_pgszcvec
 */
#define MAPPGSZC_SHM    0x01
#define MAPPGSZC_PRIVM  0x02
#define MAPPGSZC_STACK  0x04
#define MAPPGSZC_HEAP   0x08

/*
 * vacalign values for choose_addr
 */
#define ADDR_NOVACALIGN 0
#define ADDR_VACALIGN   1

struct as;
struct page;
struct anon;

extern int maxslp;
extern ulong_t pginrate;
extern ulong_t pgoutrate;
extern void swapout_lwp(klwp_t *);
```

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

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

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

152 extern  int pf_is_memory(pfn_t);

154 extern  void    dcache_flushall(void);

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

158 extern  size_t  exec_get_spslew(void);

160 #endif  /* _KERNEL */

162 #ifdef  __cplusplus
163 }
```

unchanged_portion_omitted_

```
**********************************************************
   11727 Fri Mar 28 23:33:45 2014
new/usr/src/uts/common/vm/as.h
patch remove-as_swapout
**********************************************************
______unchanged_portion_omitted_

 218 #ifdef _KERNEL

 220 /*
 221  * Flags for as_gap.
 222  */
 223 #define AH_DIR          0x1     /* direction flag mask */
 224 #define AH_LO           0x0     /* find lowest hole */
 225 #define AH_HI           0x1     /* find highest hole */
 226 #define AH_CONTAIN      0x2     /* hole must contain `addr' */

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

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

 250 /*
 251  * Macros to test lock states.
 252  */
 253 #define AS_LOCK_HELD(as, lock)          RW_LOCK_HELD((lock))
 254 #define AS_READ_HELD(as, lock)          RW_READ_HELD((lock))
 255 #define AS_WRITE_HELD(as, lock)         RW_WRITE_HELD((lock))

 257 /*
 258  * macros to walk thru segment lists
 259  */
 260 #define AS_SEGFIRST(as)         avl_first(&(as)->a_segtree)
 261 #define AS_SEGNEXT(as, seg)     AVL_NEXT(&(as)->a_segtree, (seg))
 262 #define AS_SEGPREV(as, seg)     AVL_PREV(&(as)->a_segtree, (seg))

 264 void    as_init(void);
 265 void    as_avlinit(struct as *);
 266 struct  seg *as_segat(struct as *as, caddr_t addr);
 267 void    as_rangelock(struct as *as);
 268 void    as_rangeunlock(struct as *as);
 269 struct  as *as_alloc();
 270 void    as_free(struct as *as);
 271 int     as_dup(struct as *as, struct proc *forkedproc);
 272 struct  seg *as_findseg(struct as *as, caddr_t addr, int tail);
 273 int     as_addseg(struct as *as, struct seg *newseg);
 274 struct  seg *as_removeseg(struct as *as, struct seg *seg);
 275 faultcode_t as_fault(struct hat *hat, struct as *as, caddr_t addr, size_t size,
 276                 enum fault_type type, enum seg_rw rw);
```



```
 277 faultcode_t as_faulta(struct as *as, caddr_t addr, size_t size);
 278 int     as_setprot(struct as *as, caddr_t addr, size_t size, uint_t prot);
 279 int     as_checkprot(struct as *as, caddr_t addr, size_t size, uint_t prot);
 280 int     as_unmap(struct as *as, caddr_t addr, size_t size);
 281 int     as_map(struct as *as, caddr_t addr, size_t size, int ((*crfp)()),
 282                 void *argsp);
 283 void    as_purge(struct as *as);
 284 int     as_gap(struct as *as, size_t minlen, caddr_t *basep, size_t *lenp,
 285                 uint_t flags, caddr_t addr);
 286 int     as_gap_aligned(struct as *as, size_t minlen, caddr_t *basep,
 287             size_t *lenp, uint_t flags, caddr_t addr, size_t align,
 288             size_t redzone, size_t off);

 290 int     as_memory(struct as *as, caddr_t *basep, size_t *lenp);
 291 size_t  as_swapout(struct as *as);
 291 int     as_incore(struct as *as, caddr_t addr, size_t size, char *vec,
 292                 size_t *sizep);
 293 int     as_ctl(struct as *as, caddr_t addr, size_t size, int func, int attr,
 294                 uintptr_t arg, ulong_t *lock_map, size_t pos);
 295 int     as_pagelock(struct as *as, struct page ***ppp, caddr_t addr,
 296                 size_t size, enum seg_rw rw);
 297 void    as_pageunlock(struct as *as, struct page **pp, caddr_t addr,
 298                 size_t size, enum seg_rw rw);
 299 int     as_setpagesize(struct as *as, caddr_t addr, size_t size, uint_t szc,
 300                 boolean_t wait);
 301 int     as_set_default_lpsize(struct as *as, caddr_t addr, size_t size);
 302 void    as_setwatch(struct as *as);
 303 void    as_clearwatch(struct as *as);
 304 int     as_getmemid(struct as *, caddr_t, memid_t *);

 306 int     as_add_callback(struct as *, void (*)(), void *, uint_t,
 307                         caddr_t, size_t, int);
 308 uint_t  as_delete_callback(struct as *, void *);

 310 #endif  /* _KERNEL */

 312 #ifdef  __cplusplus
 313 }
______unchanged_portion_omitted_
```

```
**********************************************************
   36950 Fri Mar 28 23:33:46 2014
new/usr/src/uts/common/vm/seg_kp.c
patch remove-dont-swap-flag
**********************************************************
______unchanged_portion_omitted_

 758 /*
 759  * segkp_map_red() will check the current frame pointer against the
 760  * stack base.  If the amount of stack remaining is questionable
 761  * (less than red_minavail), then segkp_map_red() will map in the redzone
 762  * and return 1.  Otherwise, it will return 0.  segkp_map_red() can
 763  * _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.
 764  *
 765  * It is up to the caller to remember whether segkp_map_red() successfully
 766  * mapped the redzone, and, if so, to call segkp_unmap_red() at a later
 767  * time.
 770  * time.  Note that the caller must _remain_ non-swappable until after
 771  * calling segkp_unmap_red().
 768  *
 769  * Currently, this routine is only called from pagefault() (which necessarily
 770  * satisfies the above conditions).
 771  */
 772 #if defined(STACK_GROWTH_DOWN)
 773 int
 774 segkp_map_red(void)
 775 {
 776         uintptr_t fp = STACK_BIAS + (uintptr_t)getfp();
 777 #ifndef _LP64
 778         caddr_t stkbase;
 779 #endif

 785         ASSERT(curthread->t_schedflag & TS_DONT_SWAP);

 781         /*
 782          * Optimize for the common case where we simply return.
 783          */
 784         if ((curthread->t_red_pp == NULL) &&
 785             (fp - (uintptr_t)curthread->t_stkbase >= red_minavail))
 786                 return (0);

 788 #if defined(_LP64)
 789         /*
 790          * XXX  We probably need something better than this.
 791          */
 792         panic("kernel stack overflow");
 793         /*NOTREACHED*/
 794 #else /* _LP64 */
 795         if (curthread->t_red_pp == NULL) {
 796                 page_t *red_pp;
 797                 struct seg kseg;

 799                 caddr_t red_va = (caddr_t)
 800                     (((uintptr_t)curthread->t_stkbase & (uintptr_t)PAGEMASK) -
 801                     PAGESIZE);

 803                 ASSERT(page_exists(&kvp, (u_offset_t)(uintptr_t)red_va) ==
 804                     NULL);

 806                 /*
 807                  * Allocate the physical for the red page.
 808                  */
```



```
 809                 /*
 810                  * No PG_NORELOC here to avoid waits. Unlikely to get
 811                  * a relocate happening in the short time the page exists
 812                  * and it will be OK anyway.
 813                  */

 815                 kseg.s_as = &kas;
 816                 red_pp = page_create_va(&kvp, (u_offset_t)(uintptr_t)red_va,
 817                     PAGESIZE, PG_WAIT | PG_EXCL, &kseg, red_va);
 818                 ASSERT(red_pp != NULL);

 820                 /*
 821                  * So we now have a page to jam into the redzone...
 822                  */
 823                 page_io_unlock(red_pp);

 825                 hat_memload(kas.a_hat, red_va, red_pp,
 826                     (PROT_READ|PROT_WRITE), HAT_LOAD_LOCK);
 827                 page_downgrade(red_pp);

 829                 /*
 830                  * The page is left SE_SHARED locked so we can hold on to
 831                  * the page_t pointer.
 832                  */
 833                 curthread->t_red_pp = red_pp;

 835                 atomic_add_32(&red_nmapped, 1);
 836                 while (fp - (uintptr_t)curthread->t_stkbase < red_closest) {
 837                         (void) cas32(&red_closest, red_closest,
 838                             (uint32_t)(fp - (uintptr_t)curthread->t_stkbase));
 839                 }
 840                 return (1);
 841         }

 843         stkbase = (caddr_t)(((uintptr_t)curthread->t_stkbase &
 844             (uintptr_t)PAGEMASK) - PAGESIZE);

 846         atomic_add_32(&red_ndoubles, 1);

 848         if (fp - (uintptr_t)stkbase < RED_DEEP_THRESHOLD) {
 849                 /*
 850                  * Oh boy.  We're already deep within the mapped-in
 851                  * redzone page, and the caller is trying to prepare
 852                  * for a deep stack run.  We're running without a
 853                  * redzone right now:  if the caller plows off the
 854                  * end of the stack, it'll plow another thread or
 855                  * LWP structure.  That situation could result in
 856                  * a very hard-to-debug panic, so, in the spirit of
 857                  * recording the name of one's killer in one's own
 858                  * blood, we're going to record hrestime and the calling
 859                  * thread.
 860                  */
 861                 red_deep_hires = hrestime.tv_nsec;
 862                 red_deep_thread = curthread;
 863         }

 865         /*
 866          * If this is a DEBUG kernel, and we've run too deep for comfort, toss.
 867          */
 868         ASSERT(fp - (uintptr_t)stkbase >= RED_DEEP_THRESHOLD);
 869         return (0);
 870 #endif /* _LP64 */
 871 }

 873 void
 874 segkp_unmap_red(void)
```

```
875 {
876         page_t *pp;
877         caddr_t red_va = (caddr_t)(((uintptr_t)curthread->t_stkbase &
878             (uintptr_t)PAGEMASK) - PAGESIZE);
879
880         ASSERT(curthread->t_red_pp != NULL);
887         ASSERT(curthread->t_schedflag & TS_DONT_SWAP);
882         /*
883          * Because we locked the mapping down, we can't simply rely
884          * on page_destroy() to clean everything up;  we need to call
885          * hat_unload() to explicitly unlock the mapping resources.
886          */
887         hat_unload(kas.a_hat, red_va, PAGESIZE, HAT_UNLOAD_UNLOCK);
888
889         pp = curthread->t_red_pp;
890
891         ASSERT(pp == page_find(&kvp, (u_offset_t)(uintptr_t)red_va));
892
893         /*
894          * Need to upgrade the SE_SHARED lock to SE_EXCL.
895          */
896         if (!page_tryupgrade(pp)) {
897                 /*
898                  * As there is now wait for upgrade, release the
899                  * SE_SHARED lock and wait for SE_EXCL.
900                  */
901                 page_unlock(pp);
902                 pp = page_lookup(&kvp, (u_offset_t)(uintptr_t)red_va, SE_EXCL);
903                 /* pp may be NULL here, hence the test below */
904         }
905
906         /*
907          * Destroy the page, with dontfree set to zero (i.e. free it).
908          */
909         if (pp != NULL)
910                 page_destroy(pp, 0);
911         curthread->t_red_pp = NULL;
912 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   92640 Fri Mar 28 23:33:47 2014
new/usr/src/uts/common/vm/vm_as.c
patch remove-as_swapout
**********************************************************
______unchanged_portion_omitted_

2142 /*
2143  * Swap the pages associated with the address space as out to
2144  * secondary storage, returning the number of bytes actually
2145  * swapped.
2146  *
2147  * The value returned is intended to correlate well with the process's
2148  * memory requirements.  Its usefulness for this purpose depends on
2149  * how well the segment-level routines do at returning accurate
2150  * information.
2151  */
2152 size_t
2153 as_swapout(struct as *as)
2154 {
2155         struct seg *seg;
2156         size_t swpcnt = 0;

2158         /*
2159          * Kernel-only processes have given up their address
2160          * spaces.  Of course, we shouldn't be attempting to
2161          * swap out such processes in the first place...
2162          */
2163         if (as == NULL)
2164                 return (0);

2166         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

2168         /* Prevent XHATs from attaching */
2169         mutex_enter(&as->a_contents);
2170         AS_SETBUSY(as);
2171         mutex_exit(&as->a_contents);


2174         /*
2175          * Free all mapping resources associated with the address
2176          * space.  The segment-level swapout routines capitalize
2177          * on this unmapping by scavanging pages that have become
2178          * unmapped here.
2179          */
2180         hat_swapout(as->a_hat);
2181         if (as->a_xhat != NULL)
2182                 xhat_swapout_all(as);

2184         mutex_enter(&as->a_contents);
2185         AS_CLRBUSY(as);
2186         mutex_exit(&as->a_contents);

2188         /*
2189          * Call the swapout routines of all segments in the address
2190          * space to do the actual work, accumulating the amount of
2191          * space reclaimed.
2192          */
2193         for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {
2194                 struct seg_ops *ov = seg->s_ops;

2196                 /*
2197                  * We have to check to see if the seg has
2198                  * an ops vector because the seg may have
2199                  * been in the middle of being set up when
2200                  * the process was picked for swapout.
```



```
2201                  */
2202                 if ((ov != NULL) && (ov->swapout != NULL))
2203                         swpcnt += SEGOP_SWAPOUT(seg);
2204         }
2205         AS_LOCK_EXIT(as, &as->a_lock);
2206         return (swpcnt);
2207 }

2209 /*
2143  * Determine whether data from the mappings in interval [addr, addr + size)
2144  * are in the primary memory (core) cache.
2145  */
2146 int
2147 as_incore(struct as *as, caddr_t addr,
2148     size_t size, char *vec, size_t *sizep)
2149 {
2150         struct seg *seg;
2151         size_t ssize;
2152         caddr_t raddr;          /* rounded down addr */
2153         size_t rsize;           /* rounded up size */
2154         size_t isize;                   /* iteration size */
2155         int error = 0;          /* result, assume success */

2157         *sizep = 0;
2158         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2159         rsize = ((((size_t)addr + size) + PAGEOFFSET) & PAGEMASK) -
2160             (size_t)raddr;

2162         if (raddr + rsize < raddr)              /* check for wraparound */
2163                 return (ENOMEM);

2165         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
2166         seg = as_segat(as, raddr);
2167         if (seg == NULL) {
2168                 AS_LOCK_EXIT(as, &as->a_lock);
2169                 return (-1);
2170         }

2172         for (; rsize != 0; rsize -= ssize, raddr += ssize) {
2173                 if (raddr >= seg->s_base + seg->s_size) {
2174                         seg = AS_SEGNEXT(as, seg);
2175                         if (seg == NULL || raddr != seg->s_base) {
2176                                 error = -1;
2177                                 break;
2178                         }
2179                 }
2180                 if ((raddr + rsize) > (seg->s_base + seg->s_size))
2181                         ssize = seg->s_base + seg->s_size - raddr;
2182                 else
2183                         ssize = rsize;
2184                 *sizep += isize = SEGOP_INCORE(seg, raddr, ssize, vec);
2185                 if (isize != ssize) {
2186                         error = -1;
2187                         break;
2188                 }
2189                 vec += btopr(ssize);
2190         }
2191         AS_LOCK_EXIT(as, &as->a_lock);
2192         return (error);
2193 }
______unchanged_portion_omitted_
```

```
**********************************************************
   13985 Fri Mar 28 23:33:49 2014
new/usr/src/uts/i86pc/os/mlsetup.c
patch fix-compile2
**********************************************************
______unchanged_portion_omitted_


/*
 * Setup routine called right before main(). Interposing this function
 * before main() allows us to call it in a machine-independent fashion.
 */
void
mlsetup(struct regs *rp)
{
        u_longlong_t prop_value;
        extern struct classfuncs sys_classfuncs;
        extern disp_t cpu0_disp;
        extern char t0stack[];
        extern int post_fastreboot;
        extern uint64_t plat_dr_options;

        ASSERT_STACK_ALIGNED();

        /*
         * initialize cpu_self
         */
        cpu[0]->cpu_self = cpu[0];

#if defined(__xpv)
        /*
         * Point at the hypervisor's virtual cpu structure
         */
        cpu[0]->cpu_m.mcpu_vcpu_info = &HYPERVISOR_shared_info->vcpu_info[0];
#endif

        /*
         * Set up dummy cpu_pri_data values till psm spl code is
         * installed.  This allows splx() to work on amd64.
         */

        cpu[0]->cpu_pri_data = dummy_cpu_pri;

        /*
         * check if we've got special bits to clear or set
         * when checking cpu features
         */

        if (bootprop_getval("cpuid_feature_ecx_include", &prop_value) != 0)
                cpuid_feature_ecx_include = 0;
        else
                cpuid_feature_ecx_include = (uint32_t)prop_value;

        if (bootprop_getval("cpuid_feature_ecx_exclude", &prop_value) != 0)
                cpuid_feature_ecx_exclude = 0;
        else
                cpuid_feature_ecx_exclude = (uint32_t)prop_value;

        if (bootprop_getval("cpuid_feature_edx_include", &prop_value) != 0)
                cpuid_feature_edx_include = 0;
        else
                cpuid_feature_edx_include = (uint32_t)prop_value;

        if (bootprop_getval("cpuid_feature_edx_exclude", &prop_value) != 0)
                cpuid_feature_edx_exclude = 0;
        else
```



```
                cpuid_feature_edx_exclude = (uint32_t)prop_value;

        /*
         * Initialize idt0, gdt0, ldt0_default, ktss0 and dftss.
         */
        init_desctbls();

        /*
         * lgrp_init() and possibly cpuid_pass1() need PCI config
         * space access
         */
#if defined(__xpv)
        if (DOMAIN_IS_INITDOMAIN(xen_info))
                pci_cfgspace_init();
#else
        pci_cfgspace_init();
        /*
         * Initialize the platform type from CPU 0 to ensure that
         * determine_platform() is only ever called once.
         */
        determine_platform();
#endif

        /*
         * The first lightweight pass (pass0) through the cpuid data
         * was done in locore before mlsetup was called.  Do the next
         * pass in C code.
         *
         * The x86_featureset is initialized here based on the capabilities
         * of the boot CPU.  Note that if we choose to support CPUs that have
         * different feature sets (at which point we would almost certainly
         * want to set the feature bits to correspond to the feature
         * minimum) this value may be altered.
         */
        cpuid_pass1(cpu[0], x86_featureset);

#if !defined(__xpv)
        if ((get_hwenv() & HW_XEN_HVM) != 0)
                xen_hvm_init();

        /*
         * Before we do anything with the TSCs, we need to work around
         * Intel erratum BT81.  On some CPUs, warm reset does not
         * clear the TSC.  If we are on such a CPU, we will clear TSC ourselves
         * here.  Other CPUs will clear it when we boot them later, and the
         * resulting skew will be handled by tsc_sync_master()/_slave();
         * note that such skew already exists and has to be handled anyway.
         *
         * We do this only on metal.  This same problem can occur with a
         * hypervisor that does not happen to virtualise a TSC that starts from
         * zero, regardless of CPU type; however, we do not expect hypervisors
         * that do not virtualise TSC that way to handle writes to TSC
         * correctly, either.
         */
        if (get_hwenv() == HW_NATIVE &&
            cpuid_getvendor(CPU) == X86_VENDOR_Intel &&
            cpuid_getfamily(CPU) == 6 &&
            (cpuid_getmodel(CPU) == 0x2d || cpuid_getmodel(CPU) == 0x3e) &&
            is_x86_feature(x86_featureset, X86FSET_TSC)) {
                (void) wrmsr(REG_TSC, 0UL);
        }

        /*
         * Patch the tsc_read routine with appropriate set of instructions,
         * depending on the processor family and architecure, to read the
         * time-stamp counter while ensuring no out-of-order execution.
```

```
227          * Patch it while the kernel text is still writable.
228          *
229          * Note: tsc_read is not patched for intel processors whose family
230          * is >6 and for amd whose family >f (in case they don't support rdtscp
231          * instruction, unlikely). By default tsc_read will use cpuid for
232          * serialization in such cases. The following code needs to be
233          * revisited if intel processors of family >= f retains the
234          * instruction serialization nature of mfence instruction.
235          * Note: tsc_read is not patched for x86 processors which do
236          * not support "mfence". By default tsc_read will use cpuid for
237          * serialization in such cases.
238          *
239          * The Xen hypervisor does not correctly report whether rdtscp is
240          * supported or not, so we must assume that it is not.
241          */
242         if ((get_hwenv() & HW_XEN_HVM) == 0 &&
243             is_x86_feature(x86_featureset, X86FSET_TSCP))
244                 patch_tsc_read(X86_HAVE_TSCP);
245         else if (cpuid_getvendor(CPU) == X86_VENDOR_AMD &&
246             cpuid_getfamily(CPU) <= 0xf &&
247             is_x86_feature(x86_featureset, X86FSET_SSE2))
248                 patch_tsc_read(X86_TSC_MFENCE);
249         else if (cpuid_getvendor(CPU) == X86_VENDOR_Intel &&
250             cpuid_getfamily(CPU) <= 6 &&
251             is_x86_feature(x86_featureset, X86FSET_SSE2))
252                 patch_tsc_read(X86_TSC_LFENCE);

254 #endif  /* !__xpv */

256 #if defined(__i386) && !defined(__xpv)
257         /*
258          * Some i386 processors do not implement the rdtsc instruction,
259          * or at least they do not implement it correctly. Patch them to
260          * return 0.
261          */
262         if (!is_x86_feature(x86_featureset, X86FSET_TSC))
263                 patch_tsc_read(X86_NO_TSC);
264 #endif  /* __i386 && !__xpv */

266 #if defined(__amd64) && !defined(__xpv)
267         patch_memops(cpuid_getvendor(CPU));
268 #endif  /* __amd64 && !__xpv */

270 #if !defined(__xpv)
271         /* XXPV what, if anything, should be dorked with here under xen? */

273         /*
274          * While we're thinking about the TSC, let's set up %cr4 so that
275          * userland can issue rdtsc, and initialize the TSC_AUX value
276          * (the cpuid) for the rdtscp instruction on appropriately
277          * capable hardware.
278          */
279         if (is_x86_feature(x86_featureset, X86FSET_TSC))
280                 setcr4(getcr4() & ~CR4_TSD);

282         if (is_x86_feature(x86_featureset, X86FSET_TSCP))
283                 (void) wrmsr(MSR_AMD_TSCAUX, 0);

285         if (is_x86_feature(x86_featureset, X86FSET_DE))
286                 setcr4(getcr4() | CR4_DE);
287 #endif /* __xpv */

289         /*
290          * initialize t0
291          */
292         t0.t_stk = (caddr_t)rp - MINFRAME;
```



```
293         t0.t_stkbase = t0stack;
294         t0.t_pri = maxclsyspri - 3;
295         t0.t_schedflag = 0;
295         t0.t_schedflag = TS_LOAD | TS_DONT_SWAP;
296         t0.t_procp = &p0;
297         t0.t_plockp = &p0lock.pl_lock;
298         t0.t_lwp = &lwp0;
299         t0.t_forw = &t0;
300         t0.t_back = &t0;
301         t0.t_next = &t0;
302         t0.t_prev = &t0;
303         t0.t_cpu = cpu[0];
304         t0.t_disp_queue = &cpu0_disp;
305         t0.t_bind_cpu = PBIND_NONE;
306         t0.t_bind_pset = PS_NONE;
307         t0.t_bindflag = (uchar_t)default_binding_mode;
308         t0.t_cpupart = &cp_default;
309         t0.t_clfuncs = &sys_classfuncs.thread;
310         t0.t_copyops = NULL;
311         THREAD_ONPROC(&t0, CPU);

313         lwp0.lwp_thread = &t0;
314         lwp0.lwp_regs = (void *)rp;
315         lwp0.lwp_procp = &p0;
316         t0.t_tid = p0.p_lwpcnt = p0.p_lwprcnt = p0.p_lwpid = 1;

318         p0.p_exec = NULL;
319         p0.p_stat = SRUN;
320         p0.p_flag = SSYS;
321         p0.p_tlist = &t0;
322         p0.p_stksize = 2*PAGESIZE;
323         p0.p_stkpageszc = 0;
324         p0.p_as = &kas;
325         p0.p_lockp = &p0lock;
326         p0.p_brkpageszc = 0;
327         p0.p_t1_lgrpid = LGRP_NONE;
328         p0.p_tr_lgrpid = LGRP_NONE;
329         sigorset(&p0.p_ignore, &ignoredefault);

331         CPU->cpu_thread = &t0;
332         bzero(&cpu0_disp, sizeof (disp_t));
333         CPU->cpu_disp = &cpu0_disp;
334         CPU->cpu_disp->disp_cpu = CPU;
335         CPU->cpu_dispthread = &t0;
336         CPU->cpu_idle_thread = &t0;
337         CPU->cpu_flags = CPU_READY | CPU_RUNNING | CPU_EXISTS | CPU_ENABLE;
338         CPU->cpu_dispatch_pri = t0.t_pri;

340         CPU->cpu_id = 0;

342         CPU->cpu_pri = 12;              /* initial PIL for the boot CPU */

344         /*
345          * The kernel doesn't use LDTs unless a process explicitly requests one.
346          */
347         p0.p_ldt_desc = null_sdesc;

349         /*
350          * Initialize thread/cpu microstate accounting
351          */
352         init_mstate(&t0, LMS_SYSTEM);
353         init_cpu_mstate(CPU, CMS_SYSTEM);

355         /*
356          * Initialize lists of available and active CPUs.
357          */
```

```
358         cpu_list_init(CPU);

360         pg_cpu_bootstrap(CPU);

362         /*
363          * Now that we have taken over the GDT, IDT and have initialized
364          * active CPU list it's time to inform kmdb if present.
365          */
366         if (boothowto & RB_DEBUG)
367                 kdi_idt_sync();

369         /*
370          * Explicitly set console to text mode (0x3) if this is a boot
371          * post Fast Reboot, and the console is set to CONS_SCREEN_TEXT.
372          */
373         if (post_fastreboot && boot_console_type(NULL) == CONS_SCREEN_TEXT)
374                 set_console_mode(0x3);

376         /*
377          * If requested (boot -d) drop into kmdb.
378          *
379          * This must be done after cpu_list_init() on the 64-bit kernel
380          * since taking a trap requires that we re-compute gsbase based
381          * on the cpu list.
382          */
383         if (boothowto & RB_DEBUGENTER)
384                 kmdb_enter();

386         cpu_vm_data_init(CPU);

388         rp->r_fp = 0;   /* terminate kernel stack traces! */

390         prom_init("kernel", (void *)NULL);

392         /* User-set option overrides firmware value. */
393         if (bootprop_getval(PLAT_DR_OPTIONS_NAME, &prop_value) == 0) {
394                 plat_dr_options = (uint64_t)prop_value;
395         }
396 #if defined(__xpv)
397         /* No support of DR operations on xpv */
398         plat_dr_options = 0;
399 #else   /* __xpv */
400         /* Flag PLAT_DR_FEATURE_ENABLED should only be set by DR driver. */
401         plat_dr_options &= ~PLAT_DR_FEATURE_ENABLED;
402 #ifndef __amd64
403         /* Only enable CPU/memory DR on 64 bits kernel. */
404         plat_dr_options &= ~PLAT_DR_FEATURE_MEMORY;
405         plat_dr_options &= ~PLAT_DR_FEATURE_CPU;
406 #endif  /* __amd64 */
407 #endif  /* __xpv */

409         /*
410          * Get value of "plat_dr_physmax" boot option.
411          * It overrides values calculated from MSCT or SRAT table.
412          */
413         if (bootprop_getval(PLAT_DR_PHYSMAX_NAME, &prop_value) == 0) {
414                 plat_dr_physmax = ((uint64_t)prop_value) >> PAGESHIFT;
415         }

417         /* Get value of boot_ncpus. */
418         if (bootprop_getval(BOOT_NCPUS_NAME, &prop_value) != 0) {
419                 boot_ncpus = NCPU;
420         } else {
421                 boot_ncpus = (int)prop_value;
422                 if (boot_ncpus <= 0 || boot_ncpus > NCPU)
423                         boot_ncpus = NCPU;
```



```
424         }

426         /*
427          * Set max_ncpus and boot_max_ncpus to boot_ncpus if platform doesn't
428          * support CPU DR operations.
429          */
430         if (plat_dr_support_cpu() == 0) {
431                 max_ncpus = boot_max_ncpus = boot_ncpus;
432         } else {
433                 if (bootprop_getval(PLAT_MAX_NCPUS_NAME, &prop_value) != 0) {
434                         max_ncpus = NCPU;
435                 } else {
436                         max_ncpus = (int)prop_value;
437                         if (max_ncpus <= 0 || max_ncpus > NCPU) {
438                                 max_ncpus = NCPU;
439                         }
440                         if (boot_ncpus > max_ncpus) {
441                                 boot_ncpus = max_ncpus;
442                         }
443                 }

445                 if (bootprop_getval(BOOT_MAX_NCPUS_NAME, &prop_value) != 0) {
446                         boot_max_ncpus = boot_ncpus;
447                 } else {
448                         boot_max_ncpus = (int)prop_value;
449                         if (boot_max_ncpus <= 0 || boot_max_ncpus > NCPU) {
450                                 boot_max_ncpus = boot_ncpus;
451                         } else if (boot_max_ncpus > max_ncpus) {
452                                 boot_max_ncpus = max_ncpus;
453                         }
454                 }
455         }

457         /*
458          * Initialize the lgrp framework
459          */
460         lgrp_init(LGRP_INIT_STAGE1);

462         if (boothowto & RB_HALT) {
463                 prom_printf("unix: kernel halted by -h flag\n");
464                 prom_enter_mon();
465         }

467         ASSERT_STACK_ALIGNED();

469         /*
470          * Fill out cpu_ucode_info.  Update microcode if necessary.
471          */
472         ucode_check(CPU);

474         if (workaround_errata(CPU) != 0)
475                 panic("critical workaround(s) missing for boot cpu");
476 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   61422 Fri Mar 28 23:33:50 2014
new/usr/src/uts/i86pc/os/trap.c
patch fix-compile2
**********************************************************
______unchanged_portion_omitted_

 453 #endif /* OPTERON_ERRATUM_91 */

 455 /*
 456  * Called from the trap handler when a processor trap occurs.
 457  *
 458  * Note: All user-level traps that might call stop() must exit
 459  * trap() by 'goto out' or by falling through.
 460  * Note Also: trap() is usually called with interrupts enabled, (PS_IE == 1)
 461  * however, there are paths that arrive here with PS_IE == 0 so special care
 462  * must be taken in those cases.
 463  */
 464 void
 465 trap(struct regs *rp, caddr_t addr, processorid_t cpuid)
 466 {
 467         kthread_t *ct = curthread;
 468         enum seg_rw rw;
 469         unsigned type;
 470         proc_t *p = ttoproc(ct);
 471         klwp_t *lwp = ttolwp(ct);
 472         uintptr_t lofault;
 473         label_t *onfault;
 474         faultcode_t pagefault(), res, errcode;
 475         enum fault_type fault_type;
 476         k_siginfo_t siginfo;
 477         uint_t fault = 0;
 478         int mstate;
 479         int sicode = 0;
 480         int watchcode;
 481         int watchpage;
 482         caddr_t vaddr;
 483         int singlestep_twiddle;
 484         size_t sz;
 485         int ta;
 486 #ifdef __amd64
 487         uchar_t instr;
 488 #endif

 490         ASSERT_STACK_ALIGNED();

 492         type = rp->r_trapno;
 493         CPU_STATS_ADDQ(CPU, sys, trap, 1);
 494         ASSERT(ct->t_schedflag & TS_DONT_SWAP);

 495         if (type == T_PGFLT) {

 497                 errcode = rp->r_err;
 498                 if (errcode & PF_ERR_WRITE)
 499                         rw = S_WRITE;
 500                 else if ((caddr_t)rp->r_pc == addr ||
 501                     (mmu.pt_nx != 0 && (errcode & PF_ERR_EXEC)))
 502                         rw = S_EXEC;
 503                 else
 504                         rw = S_READ;

 506 #if defined(__i386)
 507                 /*
 508                  * Pentium Pro work-around
 509                  */
 510                 if ((errcode & PF_ERR_PROT) && pentiumpro_bug4046376) {
```



```
 511                         uint_t  attr;
 512                         uint_t  priv_violation;
 513                         uint_t  access_violation;

 515                         if (hat_getattr(addr < (caddr_t)kernelbase ?
 516                             curproc->p_as->a_hat : kas.a_hat, addr, &attr)
 517                             == -1) {
 518                                 errcode &= ~PF_ERR_PROT;
 519                         } else {
 520                                 priv_violation = (errcode & PF_ERR_USER) &&
 521                                     !(attr & PROT_USER);
 522                                 access_violation = (errcode & PF_ERR_WRITE) &&
 523                                     !(attr & PROT_WRITE);
 524                                 if (!priv_violation && !access_violation)
 525                                         goto cleanup;
 526                         }
 527                 }
 528 #endif /* __i386 */

 530         } else if (type == T_SGLSTP && lwp != NULL)
 531                 lwp->lwp_pcb.pcb_drstat = (uintptr_t)addr;

 533         if (tdebug)
 534                 showregs(type, rp, addr);

 536         if (USERMODE(rp->r_cs)) {
 537                 /*
 538                  * Set up the current cred to use during this trap. u_cred
 539                  * no longer exists.  t_cred is used instead.
 540                  * The current process credential applies to the thread for
 541                  * the entire trap.  If trapping from the kernel, this
 542                  * should already be set up.
 543                  */
 544                 if (ct->t_cred != p->p_cred) {
 545                         cred_t *oldcred = ct->t_cred;
 546                         /*
 547                          * DTrace accesses t_cred in probe context.  t_cred
 548                          * must always be either NULL, or point to a valid,
 549                          * allocated cred structure.
 550                          */
 551                         ct->t_cred = crgetcred();
 552                         crfree(oldcred);
 553                 }
 554                 ASSERT(lwp != NULL);
 555                 type |= USER;
 556                 ASSERT(lwptoregs(lwp) == rp);
 557                 lwp->lwp_state = LWP_SYS;

 559                 switch (type) {
 560                 case T_PGFLT + USER:
 561                         if ((caddr_t)rp->r_pc == addr)
 562                                 mstate = LMS_TFAULT;
 563                         else
 564                                 mstate = LMS_DFAULT;
 565                         break;
 566                 default:
 567                         mstate = LMS_TRAP;
 568                         break;
 569                 }
 570                 /* Kernel probe */
 571                 TNF_PROBE_1(thread_state, "thread", /* CSTYLED */,
 572                     tnf_microstate, state, mstate);
 573                 mstate = new_mstate(ct, mstate);

 575                 bzero(&siginfo, sizeof (siginfo));
 576         }
```

```
578         switch (type) {
579         case T_PGFLT + USER:
580         case T_SGLSTP:
581         case T_SGLSTP + USER:
582         case T_BPTFLT + USER:
583                 break;

585         default:
586                 FTRACE_2("trap(): type=0x%lx, regs=0x%lx",
587                     (ulong_t)type, (ulong_t)rp);
588                 break;
589         }

591         switch (type) {
592         case T_SIMDFPE:
593                 /* Make sure we enable interrupts before die()ing */
594                 sti();  /* The SIMD exception comes in via cmninttrap */
595                 /*FALLTHROUGH*/
596         default:
597                 if (type & USER) {
598                         if (tudebug)
599                                 showregs(type, rp, (caddr_t)0);
600                         printf("trap: Unknown trap type %d in user mode\n",
601                             type & ~USER);
602                         siginfo.si_signo = SIGILL;
603                         siginfo.si_code  = ILL_ILLTRP;
604                         siginfo.si_addr  = (caddr_t)rp->r_pc;
605                         siginfo.si_trapno = type & ~USER;
606                         fault = FLTILL;
607                         break;
608                 } else {
609                         (void) die(type, rp, addr, cpuid);
610                         /*NOTREACHED*/
611                 }

613         case T_PGFLT:           /* system page fault */
614                 /*
615                  * If we're under on_trap() protection (see <sys/ontrap.h>),
616                  * set ot_trap and bounce back to the on_trap() call site
617                  * via the installed trampoline.
618                  */
619                 if ((ct->t_ontrap != NULL) &&
620                     (ct->t_ontrap->ot_prot & OT_DATA_ACCESS)) {
621                         ct->t_ontrap->ot_trap |= OT_DATA_ACCESS;
622                         rp->r_pc = ct->t_ontrap->ot_trampoline;
623                         goto cleanup;
624                 }

626                 /*
627                  * See if we can handle as pagefault. Save lofault and onfault
628                  * across this. Here we assume that an address less than
629                  * KERNELBASE is a user fault.  We can do this as copy.s
630                  * routines verify that the starting address is less than
631                  * KERNELBASE before starting and because we know that we
632                  * always have KERNELBASE mapped as invalid to serve as a
633                  * "barrier".
634                  */
635                 lofault = ct->t_lofault;
636                 onfault = ct->t_onfault;
637                 ct->t_lofault = 0;

639                 mstate = new_mstate(ct, LMS_KFAULT);

641                 if (addr < (caddr_t)kernelbase) {
642                         res = pagefault(addr,
```



```
643                             (errcode & PF_ERR_PROT)? F_PROT: F_INVAL, rw, 0);
644                         if (res == FC_NOMAP &&
645                             addr < p->p_usrstack &&
646                             grow(addr))
647                                 res = 0;
648                 } else {
649                         res = pagefault(addr,
650                             (errcode & PF_ERR_PROT)? F_PROT: F_INVAL, rw, 1);
651                 }
652                 (void) new_mstate(ct, mstate);

654                 /*
655                  * Restore lofault and onfault. If we resolved the fault, exit.
656                  * If we didn't and lofault wasn't set, die.
657                  */
658                 ct->t_lofault = lofault;
659                 ct->t_onfault = onfault;
660                 if (res == 0)
661                         goto cleanup;

663 #if defined(OPTERON_ERRATUM_93) && defined(_LP64)
664                 if (lofault == 0 && opteron_erratum_93) {
665                         /*
666                          * Workaround for Opteron Erratum 93. On return from
667                          * a System Managment Interrupt at a HLT instruction
668                          * the %rip might be truncated to a 32 bit value.
669                          * BIOS is supposed to fix this, but some don't.
670                          * If this occurs we simply restore the high order bits.
671                          * The HLT instruction is 1 byte of 0xf4.
672                          */
673                         uintptr_t       rip = rp->r_pc;

675                         if ((rip & 0xffffffffUL) == rip) {
676                                 rip |= 0xffffffffUL << 32;
677                                 if (hat_getpfnum(kas.a_hat, (caddr_t)rip) !=
678                                     PFN_INVALID &&
679                                     (*(uchar_t *)rip == 0xf4 ||
680                                     *(uchar_t *)(rip - 1) == 0xf4)) {
681                                         rp->r_pc = rip;
682                                         goto cleanup;
683                                 }
684                         }
685                 }
686 #endif /* OPTERON_ERRATUM_93 && _LP64 */

688 #ifdef OPTERON_ERRATUM_91
689                 if (lofault == 0 && opteron_erratum_91) {
690                         /*
691                          * Workaround for Opteron Erratum 91. Prefetches may
692                          * generate a page fault (they're not supposed to do
693                          * that!). If this occurs we simply return back to the
694                          * instruction.
695                          */
696                         caddr_t         pc = (caddr_t)rp->r_pc;

698                         /*
699                          * If the faulting PC is not mapped, this is a
700                          * legitimate kernel page fault that must result in a
701                          * panic. If the faulting PC is mapped, it could contain
702                          * a prefetch instruction. Check for that here.
703                          */
704                         if (hat_getpfnum(kas.a_hat, pc) != PFN_INVALID) {
705                                 if (cmp_to_prefetch((uchar_t *)pc)) {
706 #ifdef DEBUG
707                                         cmn_err(CE_WARN, "Opteron erratum 91 "
708                                             "occurred: kernel prefetch"
```

```
709                                             " at %p generated a page fault!",
710                                             (void *)rp->r_pc);
711 #endif /* DEBUG */
712                                         goto cleanup;
713                                 }
714                         }
715                         (void) die(type, rp, addr, cpuid);
716                 }
717 #endif /* OPTERON_ERRATUM_91 */

719                 if (lofault == 0)
720                         (void) die(type, rp, addr, cpuid);

722                 /*
723                  * Cannot resolve fault.  Return to lofault.
724                  */
725                 if (lodebug) {
726                         showregs(type, rp, addr);
727                         traceregs(rp);
728                 }
729                 if (FC_CODE(res) == FC_OBJERR)
730                         res = FC_ERRNO(res);
731                 else
732                         res = EFAULT;
733                 rp->r_r0 = res;
734                 rp->r_pc = ct->t_lofault;
735                 goto cleanup;

737         case T_PGFLT + USER:    /* user page fault */
738                 if (faultdebug) {
739                         char *fault_str;

741                         switch (rw) {
742                         case S_READ:
743                                 fault_str = "read";
744                                 break;
745                         case S_WRITE:
746                                 fault_str = "write";
747                                 break;
748                         case S_EXEC:
749                                 fault_str = "exec";
750                                 break;
751                         default:
752                                 fault_str = "";
753                                 break;
754                         }
755                         printf("user %s fault:  addr=0x%lx errcode=0x%x\n",
756                             fault_str, (uintptr_t)addr, errcode);
757                 }

759 #if defined(OPTERON_ERRATUM_100) && defined(_LP64)
760                 /*
761                  * Workaround for AMD erratum 100
762                  *
763                  * A 32-bit process may receive a page fault on a non
764                  * 32-bit address by mistake. The range of the faulting
765                  * address will be
766                  *
767                  *      0xffffffff80000000 .. 0xffffffffffffffff or
768                  *      0x0000000100000000 .. 0x000000017fffffff
769                  *
770                  * The fault is always due to an instruction fetch, however
771                  * the value of r_pc should be correct (in 32 bit range),
772                  * so we ignore the page fault on the bogus address.
773                  */
774                 if (p->p_model == DATAMODEL_ILP32 &&
```



```
775                     (0xffffffff80000000 <= (uintptr_t)addr ||
776                     (0x100000000 <= (uintptr_t)addr &&
777                     (uintptr_t)addr <= 0x17fffffff))) {
778                         if (!opteron_erratum_100)
779                                 panic("unexpected erratum #100");
780                         if (rp->r_pc <= 0xffffffff)
781                                 goto out;
782                 }
783 #endif /* OPTERON_ERRATUM_100 && _LP64 */

785                 ASSERT(!(curthread->t_flag & T_WATCHPT));
786                 watchpage = (pr_watch_active(p) && pr_is_watchpage(addr, rw));
787 #ifdef __i386
788                 /*
789                  * In 32-bit mode, the lcall (system call) instruction fetches
790                  * one word from the stack, at the stack pointer, because of the
791                  * way the call gate is constructed.  This is a bogus
792                  * read and should not be counted as a read watchpoint.
793                  * We work around the problem here by testing to see if
794                  * this situation applies and, if so, simply jumping to
795                  * the code in locore.s that fields the system call trap.
796                  * The registers on the stack are already set up properly
797                  * due to the match between the call gate sequence and the
798                  * trap gate sequence.  We just have to adjust the pc.
799                  */
800                 if (watchpage && addr == (caddr_t)rp->r_sp &&
801                     rw == S_READ && instr_is_lcall_syscall((caddr_t)rp->r_pc)) {
802                         extern void watch_syscall(void);

804                         rp->r_pc += LCALLSIZE;
805                         watch_syscall();        /* never returns */
806                         /* NOTREACHED */
807                 }
808 #endif /* __i386 */
809                 vaddr = addr;
810                 if (!watchpage || (sz = instr_size(rp, &vaddr, rw)) <= 0)
811                         fault_type = (errcode & PF_ERR_PROT)? F_PROT: F_INVAL;
812                 else if ((watchcode = pr_is_watchpoint(&vaddr, &ta,
813                     sz, NULL, rw)) != 0) {
814                         if (ta) {
815                                 do_watch_step(vaddr, sz, rw,
816                                     watchcode, rp->r_pc);
817                                 fault_type = F_INVAL;
818                         } else {
819                                 bzero(&siginfo, sizeof (siginfo));
820                                 siginfo.si_signo = SIGTRAP;
821                                 siginfo.si_code = watchcode;
822                                 siginfo.si_addr = vaddr;
823                                 siginfo.si_trapafter = 0;
824                                 siginfo.si_pc = (caddr_t)rp->r_pc;
825                                 fault = FLTWATCH;
826                                 break;
827                         }
828                 } else {
829                         /* XXX pr_watch_emul() never succeeds (for now) */
830                         if (rw != S_EXEC && pr_watch_emul(rp, vaddr, rw))
831                                 goto out;
832                         do_watch_step(vaddr, sz, rw, 0, 0);
833                         fault_type = F_INVAL;
834                 }

836                 res = pagefault(addr, fault_type, rw, 0);

838                 /*
839                  * If pagefault() succeeded, ok.
840                  * Otherwise attempt to grow the stack.
```

```
841                  */
842                 if (res == 0 ||
843                     (res == FC_NOMAP &&
844                     addr < p->p_usrstack &&
845                     grow(addr))) {
846                         lwp->lwp_lastfault = FLTPAGE;
847                         lwp->lwp_lastfaddr = addr;
848                         if (prismember(&p->p_fltmask, FLTPAGE)) {
849                                 bzero(&siginfo, sizeof (siginfo));
850                                 siginfo.si_addr = addr;
851                                 (void) stop_on_fault(FLTPAGE, &siginfo);
852                         }
853                         goto out;
854                 } else if (res == FC_PROT && addr < p->p_usrstack &&
855                     (mmu.pt_nx != 0 && (errcode & PF_ERR_EXEC))) {
856                         report_stack_exec(p, addr);
857                 }

859 #ifdef OPTERON_ERRATUM_91
860                 /*
861                  * Workaround for Opteron Erratum 91. Prefetches may generate a
862                  * page fault (they're not supposed to do that!). If this
863                  * occurs we simply return back to the instruction.
864                  *
865                  * We rely on copyin to properly fault in the page with r_pc.
866                  */
867                 if (opteron_erratum_91 &&
868                     addr != (caddr_t)rp->r_pc &&
869                     instr_is_prefetch((caddr_t)rp->r_pc)) {
870 #ifdef DEBUG
871                         cmn_err(CE_WARN, "Opteron erratum 91 occurred: "
872                             "prefetch at %p in pid %d generated a trap!",
873                             (void *)rp->r_pc, p->p_pid);
874 #endif /* DEBUG */
875                         goto out;
876                 }
877 #endif /* OPTERON_ERRATUM_91 */

879                 if (tudebug)
880                         showregs(type, rp, addr);
881                 /*
882                  * In the case where both pagefault and grow fail,
883                  * set the code to the value provided by pagefault.
884                  * We map all errors returned from pagefault() to SIGSEGV.
885                  */
886                 bzero(&siginfo, sizeof (siginfo));
887                 siginfo.si_addr = addr;
888                 switch (FC_CODE(res)) {
889                 case FC_HWERR:
890                 case FC_NOSUPPORT:
891                         siginfo.si_signo = SIGBUS;
892                         siginfo.si_code = BUS_ADRERR;
893                         fault = FLTACCESS;
894                         break;
895                 case FC_ALIGN:
896                         siginfo.si_signo = SIGBUS;
897                         siginfo.si_code = BUS_ADRALN;
898                         fault = FLTACCESS;
899                         break;
900                 case FC_OBJERR:
901                         if ((siginfo.si_errno = FC_ERRNO(res)) != EINTR) {
902                                 siginfo.si_signo = SIGBUS;
903                                 siginfo.si_code = BUS_OBJERR;
904                                 fault = FLTACCESS;
905                         }
906                         break;
```



```
907                 default:        /* FC_NOMAP or FC_PROT */
908                         siginfo.si_signo = SIGSEGV;
909                         siginfo.si_code =
910                             (res == FC_NOMAP)? SEGV_MAPERR : SEGV_ACCERR;
911                         fault = FLTBOUNDS;
912                         break;
913                 }
914                 break;

916         case T_ILLINST + USER:  /* invalid opcode fault */
917                 /*
918                  * If the syscall instruction is disabled due to LDT usage, a
919                  * user program that attempts to execute it will trigger a #ud
920                  * trap. Check for that case here. If this occurs on a CPU which
921                  * doesn't even support syscall, the result of all of this will
922                  * be to emulate that particular instruction.
923                  */
924                 if (p->p_ldt != NULL &&
925                     ldt_rewrite_syscall(rp, p, X86FSET_ASYSC))
926                         goto out;

928 #ifdef __amd64
929                 /*
930                  * Emulate the LAHF and SAHF instructions if needed.
931                  * See the instr_is_lsahf function for details.
932                  */
933                 if (p->p_model == DATAMODEL_LP64 &&
934                     instr_is_lsahf((caddr_t)rp->r_pc, &instr)) {
935                         emulate_lsahf(rp, instr);
936                         goto out;
937                 }
938 #endif

940                 /*FALLTHROUGH*/

942                 if (tudebug)
943                         showregs(type, rp, (caddr_t)0);
944                 siginfo.si_signo = SIGILL;
945                 siginfo.si_code  = ILL_ILLOPC;
946                 siginfo.si_addr  = (caddr_t)rp->r_pc;
947                 fault = FLTILL;
948                 break;

950         case T_ZERODIV + USER:          /* integer divide by zero */
951                 if (tudebug && tudebugfpe)
952                         showregs(type, rp, (caddr_t)0);
953                 siginfo.si_signo = SIGFPE;
954                 siginfo.si_code  = FPE_INTDIV;
955                 siginfo.si_addr  = (caddr_t)rp->r_pc;
956                 fault = FLTIZDIV;
957                 break;

959         case T_OVFLW + USER:    /* integer overflow */
960                 if (tudebug && tudebugfpe)
961                         showregs(type, rp, (caddr_t)0);
962                 siginfo.si_signo = SIGFPE;
963                 siginfo.si_code  = FPE_INTOVF;
964                 siginfo.si_addr  = (caddr_t)rp->r_pc;
965                 fault = FLTIOVF;
966                 break;

968         case T_NOEXTFLT + USER: /* math coprocessor not available */
969                 if (tudebug && tudebugfpe)
970                         showregs(type, rp, addr);
971                 if (fpnoextflt(rp)) {
972                         siginfo.si_signo = SIGILL;
```

```
 973                         siginfo.si_code  = ILL_ILLOPC;
 974                         siginfo.si_addr  = (caddr_t)rp->r_pc;
 975                         fault = FLTILL;
 976                 }
 977                 break;

 979         case T_EXTOVRFLT:       /* extension overrun fault */
 980                 /* check if we took a kernel trap on behalf of user */
 981                 {
 982                         extern  void ndptrap_frstor(void);
 983                         if (rp->r_pc != (uintptr_t)ndptrap_frstor) {
 984                                 sti(); /* T_EXTOVRFLT comes in via cmninttrap */
 985                                 (void) die(type, rp, addr, cpuid);
 986                         }
 987                         type |= USER;
 988                 }
 989                 /*FALLTHROUGH*/
 990         case T_EXTOVRFLT + USER:        /* extension overrun fault */
 991                 if (tudebug && tudebugfpe)
 992                         showregs(type, rp, addr);
 993                 if (fpextovrflt(rp)) {
 994                         siginfo.si_signo = SIGSEGV;
 995                         siginfo.si_code  = SEGV_MAPERR;
 996                         siginfo.si_addr  = (caddr_t)rp->r_pc;
 997                         fault = FLTBOUNDS;
 998                 }
 999                 break;

1001         case T_EXTERRFLT:       /* x87 floating point exception pending */
1002                 /* check if we took a kernel trap on behalf of user */
1003                 {
1004                         extern  void ndptrap_frstor(void);
1005                         if (rp->r_pc != (uintptr_t)ndptrap_frstor) {
1006                                 sti(); /* T_EXTERRFLT comes in via cmninttrap */
1007                                 (void) die(type, rp, addr, cpuid);
1008                         }
1009                         type |= USER;
1010                 }
1011                 /*FALLTHROUGH*/

1013         case T_EXTERRFLT + USER: /* x87 floating point exception pending */
1014                 if (tudebug && tudebugfpe)
1015                         showregs(type, rp, addr);
1016                 if (sicode = fpexterrflt(rp)) {
1017                         siginfo.si_signo = SIGFPE;
1018                         siginfo.si_code  = sicode;
1019                         siginfo.si_addr  = (caddr_t)rp->r_pc;
1020                         fault = FLTFPE;
1021                 }
1022                 break;

1024         case T_SIMDFPE + USER:          /* SSE and SSE2 exceptions */
1025                 if (tudebug && tudebugsse)
1026                         showregs(type, rp, addr);
1027                 if (!is_x86_feature(x86_featureset, X86FSET_SSE) &&
1028                     !is_x86_feature(x86_featureset, X86FSET_SSE2)) {
1029                         /*
1030                          * There are rumours that some user instructions
1031                          * on older CPUs can cause this trap to occur; in
1032                          * which case send a SIGILL instead of a SIGFPE.
1033                          */
1034                         siginfo.si_signo = SIGILL;
1035                         siginfo.si_code  = ILL_ILLTRP;
1036                         siginfo.si_addr  = (caddr_t)rp->r_pc;
1037                         siginfo.si_trapno = type & ~USER;
1038                         fault = FLTILL;
```



```
1039                 } else if ((sicode = fpsimderrflt(rp)) != 0) {
1040                         siginfo.si_signo = SIGFPE;
1041                         siginfo.si_code = sicode;
1042                         siginfo.si_addr = (caddr_t)rp->r_pc;
1043                         fault = FLTFPE;
1044                 }

1046                 sti();  /* The SIMD exception comes in via cmninttrap */
1047                 break;

1049         case T_BPTFLT:  /* breakpoint trap */
1050                 /*
1051                  * Kernel breakpoint traps should only happen when kmdb is
1052                  * active, and even then, it'll have interposed on the IDT, so
1053                  * control won't get here.  If it does, we've hit a breakpoint
1054                  * without the debugger, which is very strange, and very
1055                  * fatal.
1056                  */
1057                 if (tudebug && tudebugbpt)
1058                         showregs(type, rp, (caddr_t)0);

1060                 (void) die(type, rp, addr, cpuid);
1061                 break;

1063         case T_SGLSTP: /* single step/hw breakpoint exception */

1065                 /* Now evaluate how we got here */
1066                 if (lwp != NULL && (lwp->lwp_pcb.pcb_drstat & DR_SINGLESTEP)) {
1067                         /*
1068                          * i386 single-steps even through lcalls which
1069                          * change the privilege level. So we take a trap at
1070                          * the first instruction in privileged mode.
1071                          *
1072                          * Set a flag to indicate that upon completion of
1073                          * the system call, deal with the single-step trap.
1074                          *
1075                          * The same thing happens for sysenter, too.
1076                          */
1077                         singlestep_twiddle = 0;
1078                         if (rp->r_pc == (uintptr_t)sys_sysenter ||
1079                             rp->r_pc == (uintptr_t)brand_sys_sysenter) {
1080                                 singlestep_twiddle = 1;
1081 #if defined(__amd64)
1082                                 /*
1083                                  * Since we are already on the kernel's
1084                                  * %gs, on 64-bit systems the sysenter case
1085                                  * needs to adjust the pc to avoid
1086                                  * executing the swapgs instruction at the
1087                                  * top of the handler.
1088                                  */
1089                                 if (rp->r_pc == (uintptr_t)sys_sysenter)
1090                                         rp->r_pc = (uintptr_t)
1091                                             _sys_sysenter_post_swapgs;
1092                                 else
1093                                         rp->r_pc = (uintptr_t)
1094                                             _brand_sys_sysenter_post_swapgs;
1095 #endif
1096                         }
1097 #if defined(__i386)
1098                         else if (rp->r_pc == (uintptr_t)sys_call ||
1099                             rp->r_pc == (uintptr_t)brand_sys_call) {
1100                                 singlestep_twiddle = 1;
1101                         }
1102 #endif
1103                         else {
1104                                 /* not on sysenter/syscall; uregs available */
```

```
1105                                if (tudebug && tudebugbpt)
1106                                        showregs(type, rp, (caddr_t)0);
1107                        }
1108                        if (singlestep_twiddle) {
1109                                rp->r_ps &= ~PS_T; /* turn off trace */
1110                                lwp->lwp_pcb.pcb_flags |= DEBUG_PENDING;
1111                                ct->t_post_sys = 1;
1112                                aston(curthread);
1113                                goto cleanup;
1114                        }
1115                }
1116                /* XXX - needs review on debugger interface? */
1117                if (boothowto & RB_DEBUG)
1118                        debug_enter((char *)NULL);
1119                else
1120                        (void) die(type, rp, addr, cpuid);
1121                break;

1123        case T_NMIFLT:  /* NMI interrupt */
1124                printf("Unexpected NMI in system mode\n");
1125                goto cleanup;

1127        case T_NMIFLT + USER:   /* NMI interrupt */
1128                printf("Unexpected NMI in user mode\n");
1129                break;

1131        case T_GPFLT:   /* general protection violation */
1132                /*
1133                 * Any #GP that occurs during an on_trap .. no_trap bracket
1134                 * with OT_DATA_ACCESS or OT_SEGMENT_ACCESS protection,
1135                 * or in a on_fault .. no_fault bracket, is forgiven
1136                 * and we trampoline.  This protection is given regardless
1137                 * of whether we are 32/64 bit etc - if a distinction is
1138                 * required then define new on_trap protection types.
1139                 *
1140                 * On amd64, we can get a #gp from referencing addresses
1141                 * in the virtual address hole e.g. from a copyin or in
1142                 * update_sregs while updating user segment registers.
1143                 *
1144                 * On the 32-bit hypervisor we could also generate one in
1145                 * mfn_to_pfn by reaching around or into where the hypervisor
1146                 * lives which is protected by segmentation.
1147                 */

1149                /*
1150                 * If we're under on_trap() protection (see <sys/ontrap.h>),
1151                 * set ot_trap and trampoline back to the on_trap() call site
1152                 * for OT_DATA_ACCESS or OT_SEGMENT_ACCESS.
1153                 */
1154                if (ct->t_ontrap != NULL) {
1155                        int ttype =  ct->t_ontrap->ot_prot &
1156                            (OT_DATA_ACCESS | OT_SEGMENT_ACCESS);

1158                        if (ttype != 0) {
1159                                ct->t_ontrap->ot_trap |= ttype;
1160                                if (tudebug)
1161                                        showregs(type, rp, (caddr_t)0);
1162                                rp->r_pc = ct->t_ontrap->ot_trampoline;
1163                                goto cleanup;
1164                        }
1165                }

1167                /*
1168                 * If we're under lofault protection (copyin etc.),
1169                 * longjmp back to lofault with an EFAULT.
1170                 */
```



```
1171                if (ct->t_lofault) {
1172                        /*
1173                         * Fault is not resolvable, so just return to lofault
1174                         */
1175                        if (lodebug) {
1176                                showregs(type, rp, addr);
1177                                traceregs(rp);
1178                        }
1179                        rp->r_r0 = EFAULT;
1180                        rp->r_pc = ct->t_lofault;
1181                        goto cleanup;
1182                }

1184                /*
1185                 * We fall through to the next case, which repeats
1186                 * the OT_SEGMENT_ACCESS check which we've already
1187                 * done, so we'll always fall through to the
1188                 * T_STKFLT case.
1189                 */
1190                /*FALLTHROUGH*/
1191        case T_SEGFLT:  /* segment not present fault */
1192                /*
1193                 * One example of this is #NP in update_sregs while
1194                 * attempting to update a user segment register
1195                 * that points to a descriptor that is marked not
1196                 * present.
1197                 */
1198                if (ct->t_ontrap != NULL &&
1199                    ct->t_ontrap->ot_prot & OT_SEGMENT_ACCESS) {
1200                        ct->t_ontrap->ot_trap |= OT_SEGMENT_ACCESS;
1201                        if (tudebug)
1202                                showregs(type, rp, (caddr_t)0);
1203                        rp->r_pc = ct->t_ontrap->ot_trampoline;
1204                        goto cleanup;
1205                }
1206                /*FALLTHROUGH*/
1207        case T_STKFLT:  /* stack fault */
1208        case T_TSSFLT:  /* invalid TSS fault */
1209                if (tudebug)
1210                        showregs(type, rp, (caddr_t)0);
1211                if (kern_gpfault(rp))
1212                        (void) die(type, rp, addr, cpuid);
1213                goto cleanup;

1215        /*
1216         * ONLY 32-bit PROCESSES can USE a PRIVATE LDT! 64-bit apps
1217         * should have no need for them, so we put a stop to it here.
1218         *
1219         * So: not-present fault is ONLY valid for 32-bit processes with
1220         * a private LDT trying to do a system call. Emulate it.
1221         *
1222         * #gp fault is ONLY valid for 32-bit processes also, which DO NOT
1223         * have a private LDT, and are trying to do a system call. Emulate it.
1224         */

1226        case T_SEGFLT + USER:   /* segment not present fault */
1227        case T_GPFLT + USER:    /* general protection violation */
1228 #ifdef _SYSCALL32_IMPL
1229                if (p->p_model != DATAMODEL_NATIVE) {
1230 #endif /* _SYSCALL32_IMPL */
1231                if (instr_is_lcall_syscall((caddr_t)rp->r_pc)) {
1232                        if (type == T_SEGFLT + USER)
1233                                ASSERT(p->p_ldt != NULL);

1235                        if ((p->p_ldt == NULL && type == T_GPFLT + USER) ||
1236                            type == T_SEGFLT + USER) {
```

```
1238                        /*
1239                         * The user attempted a system call via the obsolete
1240                         * call gate mechanism. Because the process doesn't have
1241                         * an LDT (i.e. the ldtr contains 0), a #gp results.
1242                         * Emulate the syscall here, just as we do above for a
1243                         * #np trap.
1244                         */

1246                        /*
1247                         * Since this is a not-present trap, rp->r_pc points to
1248                         * the trapping lcall instruction. We need to bump it
1249                         * to the next insn so the app can continue on.
1250                         */
1251                        rp->r_pc += LCALLSIZE;
1252                        lwp->lwp_regs = rp;

1254                        /*
1255                         * Normally the microstate of the LWP is forced back to
1256                         * LMS_USER by the syscall handlers. Emulate that
1257                         * behavior here.
1258                         */
1259                        mstate = LMS_USER;

1261                        dosyscall();
1262                        goto out;
1263                        }
1264                }
1265 #ifdef _SYSCALL32_IMPL
1266                }
1267 #endif /* _SYSCALL32_IMPL */
1268                /*
1269                 * If the current process is using a private LDT and the
1270                 * trapping instruction is sysenter, the sysenter instruction
1271                 * has been disabled on the CPU because it destroys segment
1272                 * registers. If this is the case, rewrite the instruction to
1273                 * be a safe system call and retry it. If this occurs on a CPU
1274                 * which doesn't even support sysenter, the result of all of
1275                 * this will be to emulate that particular instruction.
1276                 */
1277                if (p->p_ldt != NULL &&
1278                    ldt_rewrite_syscall(rp, p, X86FSET_SEP))
1279                        goto out;

1281                /*FALLTHROUGH*/

1283        case T_BOUNDFLT + USER: /* bound fault */
1284        case T_STKFLT + USER:   /* stack fault */
1285        case T_TSSFLT + USER:   /* invalid TSS fault */
1286                if (tudebug)
1287                        showregs(type, rp, (caddr_t)0);
1288                siginfo.si_signo = SIGSEGV;
1289                siginfo.si_code  = SEGV_MAPERR;
1290                siginfo.si_addr  = (caddr_t)rp->r_pc;
1291                fault = FLTBOUNDS;
1292                break;

1294        case T_ALIGNMENT + USER:        /* user alignment error (486) */
1295                if (tudebug)
1296                        showregs(type, rp, (caddr_t)0);
1297                bzero(&siginfo, sizeof (siginfo));
1298                siginfo.si_signo = SIGBUS;
1299                siginfo.si_code = BUS_ADRALN;
1300                siginfo.si_addr = (caddr_t)rp->r_pc;
1301                fault = FLTACCESS;
1302                break;
```



```
1304        case T_SGLSTP + USER: /* single step/hw breakpoint exception */
1305                if (tudebug && tudebugbpt)
1306                        showregs(type, rp, (caddr_t)0);

1308                /* Was it single-stepping? */
1309                if (lwp->lwp_pcb.pcb_drstat & DR_SINGLESTEP) {
1310                        pcb_t *pcb = &lwp->lwp_pcb;

1312                        rp->r_ps &= ~PS_T;
1313                        /*
1314                         * If both NORMAL_STEP and WATCH_STEP are in effect,
1315                         * give precedence to WATCH_STEP.  If neither is set,
1316                         * user must have set the PS_T bit in %efl; treat this
1317                         * as NORMAL_STEP.
1318                         */
1319                        if ((fault = undo_watch_step(&siginfo)) == 0 &&
1320                            ((pcb->pcb_flags & NORMAL_STEP) ||
1321                            !(pcb->pcb_flags & WATCH_STEP))) {
1322                                siginfo.si_signo = SIGTRAP;
1323                                siginfo.si_code = TRAP_TRACE;
1324                                siginfo.si_addr = (caddr_t)rp->r_pc;
1325                                fault = FLTTRACE;
1326                        }
1327                        pcb->pcb_flags &= ~(NORMAL_STEP|WATCH_STEP);
1328                }
1329                break;

1331        case T_BPTFLT + USER:   /* breakpoint trap */
1332                if (tudebug && tudebugbpt)
1333                        showregs(type, rp, (caddr_t)0);
1334                /*
1335                 * int 3 (the breakpoint instruction) leaves the pc referring
1336                 * to the address one byte after the breakpointed address.
1337                 * If the P_PR_BPTADJ flag has been set via /proc, We adjust
1338                 * it back so it refers to the breakpointed address.
1339                 */
1340                if (p->p_proc_flag & P_PR_BPTADJ)
1341                        rp->r_pc--;
1342                siginfo.si_signo = SIGTRAP;
1343                siginfo.si_code  = TRAP_BRKPT;
1344                siginfo.si_addr  = (caddr_t)rp->r_pc;
1345                fault = FLTBPT;
1346                break;

1348        case T_AST:
1349                /*
1350                 * This occurs only after the cs register has been made to
1351                 * look like a kernel selector, either through debugging or
1352                 * possibly by functions like setcontext().  The thread is
1353                 * about to cause a general protection fault at common_iret()
1354                 * in locore.  We let that happen immediately instead of
1355                 * doing the T_AST processing.
1356                 */
1357                goto cleanup;

1359        case T_AST + USER:      /* profiling, resched, h/w error pseudo trap */
1360                if (lwp->lwp_pcb.pcb_flags & ASYNC_HWERR) {
1361                        proc_t *p = ttoproc(curthread);
1362                        extern void print_msg_hwerr(ctid_t ct_id, proc_t *p);

1364                        lwp->lwp_pcb.pcb_flags &= ~ASYNC_HWERR;
1365                        print_msg_hwerr(p->p_ct_process->conp_contract.ct_id,
1366                            p);
1367                        contract_process_hwerr(p->p_ct_process, p);
1368                        siginfo.si_signo = SIGKILL;
```

```
1369                        siginfo.si_code = SI_NOINFO;
1370                } else if (lwp->lwp_pcb.pcb_flags & CPC_OVERFLOW) {
1371                        lwp->lwp_pcb.pcb_flags &= ~CPC_OVERFLOW;
1372                        if (kcpc_overflow_ast()) {
1373                                /*
1374                                 * Signal performance counter overflow
1375                                 */
1376                                if (tudebug)
1377                                        showregs(type, rp, (caddr_t)0);
1378                                bzero(&siginfo, sizeof (siginfo));
1379                                siginfo.si_signo = SIGEMT;
1380                                siginfo.si_code = EMT_CPCOVF;
1381                                siginfo.si_addr = (caddr_t)rp->r_pc;
1382                                fault = FLTCPCOVF;
1383                        }
1384                }

1386                break;
1387        }

1389        /*
1390         * We can't get here from a system trap
1391         */
1392        ASSERT(type & USER);

1394        if (fault) {
1395                /* We took a fault so abort single step. */
1396                lwp->lwp_pcb.pcb_flags &= ~(NORMAL_STEP|WATCH_STEP);
1397                /*
1398                 * Remember the fault and fault adddress
1399                 * for real-time (SIGPROF) profiling.
1400                 */
1401                lwp->lwp_lastfault = fault;
1402                lwp->lwp_lastfaddr = siginfo.si_addr;

1404                DTRACE_PROC2(fault, int, fault, ksiginfo_t *, &siginfo);

1406                /*
1407                 * If a debugger has declared this fault to be an
1408                 * event of interest, stop the lwp.  Otherwise just
1409                 * deliver the associated signal.
1410                 */
1411                if (siginfo.si_signo != SIGKILL &&
1412                    prismember(&p->p_fltmask, fault) &&
1413                    stop_on_fault(fault, &siginfo) == 0)
1414                        siginfo.si_signo = 0;
1415        }

1417        if (siginfo.si_signo)
1418                trapsig(&siginfo, (fault != FLTFPE && fault != FLTCPCOVF));

1420        if (lwp->lwp_oweupc)
1421                profil_tick(rp->r_pc);

1423        if (ct->t_astflag | ct->t_sig_check) {
1424                /*
1425                 * Turn off the AST flag before checking all the conditions that
1426                 * may have caused an AST.  This flag is on whenever a signal or
1427                 * unusual condition should be handled after the next trap or
1428                 * syscall.
1429                 */
1430                astoff(ct);
1431                /*
1432                 * If a single-step trap occurred on a syscall (see above)
1433                 * recognize it now.  Do this before checking for signals
1434                 * because deferred_singlestep_trap() may generate a SIGTRAP to
```



```
1435                 * the LWP or may otherwise mark the LWP to call issig(FORREAL).
1436                 */
1437                if (lwp->lwp_pcb.pcb_flags & DEBUG_PENDING)
1438                        deferred_singlestep_trap((caddr_t)rp->r_pc);

1440                ct->t_sig_check = 0;

1442                mutex_enter(&p->p_lock);
1443                if (curthread->t_proc_flag & TP_CHANGEBIND) {
1444                        timer_lwpbind();
1445                        curthread->t_proc_flag &= ~TP_CHANGEBIND;
1446                }
1447                mutex_exit(&p->p_lock);

1449                /*
1450                 * for kaio requests that are on the per-process poll queue,
1451                 * aiop->aio_pollq, they're AIO_POLL bit is set, the kernel
1452                 * should copyout their result_t to user memory. by copying
1453                 * out the result_t, the user can poll on memory waiting
1454                 * for the kaio request to complete.
1455                 */
1456                if (p->p_aio)
1457                        aio_cleanup(0);
1458                /*
1459                 * If this LWP was asked to hold, call holdlwp(), which will
1460                 * stop.  holdlwps() sets this up and calls pokelwps() which
1461                 * sets the AST flag.
1462                 *
1463                 * Also check TP_EXITLWP, since this is used by fresh new LWPs
1464                 * through lwp_rtt().  That flag is set if the lwp_create(2)
1465                 * syscall failed after creating the LWP.
1466                 */
1467                if (ISHOLD(p))
1468                        holdlwp();

1470                /*
1471                 * All code that sets signals and makes ISSIG evaluate true must
1472                 * set t_astflag afterwards.
1473                 */
1474                if (ISSIG_PENDING(ct, lwp, p)) {
1475                        if (issig(FORREAL))
1476                                psig();
1477                        ct->t_sig_check = 1;
1478                }

1480                if (ct->t_rprof != NULL) {
1481                        realsigprof(0, 0, 0);
1482                        ct->t_sig_check = 1;
1483                }

1485                /*
1486                 * /proc can't enable/disable the trace bit itself
1487                 * because that could race with the call gate used by
1488                 * system calls via "lcall". If that happened, an
1489                 * invalid EFLAGS would result. prstep()/prnostep()
1490                 * therefore schedule an AST for the purpose.
1491                 */
1492                if (lwp->lwp_pcb.pcb_flags & REQUEST_STEP) {
1493                        lwp->lwp_pcb.pcb_flags &= ~REQUEST_STEP;
1494                        rp->r_ps |= PS_T;
1495                }
1496                if (lwp->lwp_pcb.pcb_flags & REQUEST_NOSTEP) {
1497                        lwp->lwp_pcb.pcb_flags &= ~REQUEST_NOSTEP;
1498                        rp->r_ps &= ~PS_T;
1499                }
1500        }
```

```
1502 out:    /* We can't get here from a system trap */
1503         ASSERT(type & USER);

1505         if (ISHOLD(p))
1506                 holdlwp();

1508         /*
1509          * Set state to LWP_USER here so preempt won't give us a kernel
1510          * priority if it occurs after this point.  Call CL_TRAPRET() to
1511          * restore the user-level priority.
1512          *
1513          * It is important that no locks (other than spinlocks) be entered
1514          * after this point before returning to user mode (unless lwp_state
1515          * is set back to LWP_SYS).
1516          */
1517         lwp->lwp_state = LWP_USER;

1519         if (ct->t_trapret) {
1520                 ct->t_trapret = 0;
1521                 thread_lock(ct);
1522                 CL_TRAPRET(ct);
1523                 thread_unlock(ct);
1524         }
1525         if (CPU->cpu_runrun || curthread->t_schedflag & TS_ANYWAITQ)
1526                 preempt();
1527         prunstop();
1528         (void) new_mstate(ct, mstate);

1530         /* Kernel probe */
1531         TNF_PROBE_1(thread_state, "thread", /* CSTYLED */,
1532             tnf_microstate, state, LMS_USER);

1534         return;

1536 cleanup:        /* system traps end up here */
1537         ASSERT(!(type & USER));
1538 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   35882 Fri Mar 28 23:33:52 2014
new/usr/src/uts/intel/ia32/os/syscall.c
patch fix-compile2
**********************************************************
______unchanged_portion_omitted_

 137 /*
 138  * Called from syscall() when a non-trivial 32-bit system call occurs.
 139  *      Sets up the args and returns a pointer to the handler.
 140  */
 141 struct sysent *
 142 syscall_entry(kthread_t *t, long *argp)
 143 {
 144         klwp_t *lwp = ttolwp(t);
 145         struct regs *rp = lwptoregs(lwp);
 146         unsigned int code;
 147         struct sysent *callp;
 148         struct sysent *se = LWP_GETSYSENT(lwp);
 149         int error = 0;
 150         uint_t nargs;

 152         ASSERT(t == curthread);
 152         ASSERT(t == curthread && curthread->t_schedflag & TS_DONT_SWAP);

 154         lwp->lwp_ru.sysc++;
 155         lwp->lwp_eosys = NORMALRETURN;  /* assume this will be normal */

 157         /*
 158          * Set lwp_ap to point to the args, even if none are needed for this
 159          * system call.  This is for the loadable-syscall case where the
 160          * number of args won't be known until the system call is loaded, and
 161          * also maintains a non-NULL lwp_ap setup for get_syscall_args(). Note
 162          * that lwp_ap MUST be set to a non-NULL value _BEFORE_ t_sysnum is
 163          * set to non-zero; otherwise get_syscall_args(), seeing a non-zero
 164          * t_sysnum for this thread, will charge ahead and dereference lwp_ap.
 165          */
 166         lwp->lwp_ap = argp;             /* for get_syscall_args */

 168         code = rp->r_r0;
 169         t->t_sysnum = (short)code;
 170         callp = code >= NSYSCALL ? &nosys_ent : se + code;

 172         if ((t->t_pre_sys | syscalltrace) != 0) {
 173                 error = pre_syscall();

 175                 /*
 176                  * pre_syscall() has taken care so that lwp_ap is current;
 177                  * it either points to syscall-entry-saved amd64 regs,
 178                  * or it points to lwp_arg[], which has been re-copied from
 179                  * the ia32 ustack, but either way, it's a current copy after
 180                  * /proc has possibly mucked with the syscall args.
 181                  */

 183                 if (error)
 184                         return (&sysent_err);   /* use dummy handler */
 185         }

 187         /*
 188          * Fetch the system call arguments to the kernel stack copy used
 189          * for syscall handling.
 190          * Note: for loadable system calls the number of arguments required
 191          * may not be known at this point, and will be zero if the system call
 192          * was never loaded.  Once the system call has been loaded, the number
 193          * of args is not allowed to be changed.
 194          */
```



```
 195         if ((nargs = (uint_t)callp->sy_narg) != 0 &&
 196             COPYIN_ARGS32(rp, argp, nargs)) {
 197                 (void) set_errno(EFAULT);
 198                 return (&sysent_err);   /* use dummy handler */
 199         }

 201         return (callp);         /* return sysent entry for caller */
 202 }
______unchanged_portion_omitted_

 227 /*
 228  * Perform pre-system-call processing, including stopping for tracing,
 229  * auditing, etc.
 230  *
 231  * This routine is called only if the t_pre_sys flag is set. Any condition
 232  * requiring pre-syscall handling must set the t_pre_sys flag. If the
 233  * condition is persistent, this routine will repost t_pre_sys.
 234  */
 235 int
 236 pre_syscall()
 237 {
 238         kthread_t *t = curthread;
 239         unsigned code = t->t_sysnum;
 240         klwp_t *lwp = ttolwp(t);
 241         proc_t *p = ttoproc(t);
 242         int     repost;

 244         t->t_pre_sys = repost = 0;      /* clear pre-syscall processing flag */

 246         ASSERT(t->t_schedflag & TS_DONT_SWAP);

 246 #if defined(DEBUG)
 247         /*
 248          * On the i386 kernel, lwp_ap points at the piece of the thread
 249          * stack that we copy the users arguments into.
 250          *
 251          * On the amd64 kernel, the syscall arguments in the rdi..r9
 252          * registers should be pointed at by lwp_ap.  If the args need to
 253          * be copied so that those registers can be changed without losing
 254          * the ability to get the args for /proc, they can be saved by
 255          * save_syscall_args(), and lwp_ap will be restored by post_syscall().
 256          */
 257         if (lwp_getdatamodel(lwp) == DATAMODEL_NATIVE) {
 258 #if defined(_LP64)
 259                 ASSERT(lwp->lwp_ap == (long *)&lwptoregs(lwp)->r_rdi);
 260         } else {
 261 #endif
 262                 ASSERT((caddr_t)lwp->lwp_ap > t->t_stkbase &&
 263                     (caddr_t)lwp->lwp_ap < t->t_stk);
 264         }
 265 #endif  /* DEBUG */

 267         /*
 268          * Make sure the thread is holding the latest credentials for the
 269          * process.  The credentials in the process right now apply to this
 270          * thread for the entire system call.
 271          */
 272         if (t->t_cred != p->p_cred) {
 273                 cred_t *oldcred = t->t_cred;
 274                 /*
 275                  * DTrace accesses t_cred in probe context.  t_cred must
 276                  * always be either NULL, or point to a valid, allocated cred
 277                  * structure.
 278                  */
 279                 t->t_cred = crgetcred();
 280                 crfree(oldcred);
```

```
281         }

283         /*
284          * From the proc(4) manual page:
285          * When entry to a system call is being traced, the traced process
286          * stops after having begun the call to the system but before the
287          * system call arguments have been fetched from the process.
288          */
289         if (PTOU(p)->u_systrap) {
290                 if (prismember(&PTOU(p)->u_entrymask, code)) {
291                         mutex_enter(&p->p_lock);
292                         /*
293                          * Recheck stop condition, now that lock is held.
294                          */
295                         if (PTOU(p)->u_systrap &&
296                             prismember(&PTOU(p)->u_entrymask, code)) {
297                                 stop(PR_SYSENTRY, code);

299                                 /*
300                                  * /proc may have modified syscall args,
301                                  * either in regs for amd64 or on ustack
302                                  * for ia32.  Either way, arrange to
303                                  * copy them again, both for the syscall
304                                  * handler and for other consumers in
305                                  * post_syscall (like audit).  Here, we
306                                  * only do amd64, and just set lwp_ap
307                                  * back to the kernel-entry stack copy;
308                                  * the syscall ml code redoes
309                                  * move-from-regs to set up for the
310                                  * syscall handler after we return.  For
311                                  * ia32, save_syscall_args() below makes
312                                  * an lwp_ap-accessible copy.
313                                  */
314 #if defined(_LP64)
315                                 if (lwp_getdatamodel(lwp) == DATAMODEL_NATIVE) {
316                                         lwp->lwp_argsaved = 0;
317                                         lwp->lwp_ap =
318                                             (long *)&lwptoregs(lwp)->r_rdi;
319                                 }
320 #endif
321                         }
322                         mutex_exit(&p->p_lock);
323                 }
324                 repost = 1;
325         }

327         /*
328          * ia32 kernel, or ia32 proc on amd64 kernel: keep args in
329          * lwp_arg for post-syscall processing, regardless of whether
330          * they might have been changed in /proc above.
331          */
332 #if defined(_LP64)
333         if (lwp_getdatamodel(lwp) != DATAMODEL_NATIVE)
334 #endif
335                 (void) save_syscall_args();

337         if (lwp->lwp_sysabort) {
338                 /*
339                  * lwp_sysabort may have been set via /proc while the process
340                  * was stopped on PR_SYSENTRY.  If so, abort the system call.
341                  * Override any error from the copyin() of the arguments.
342                  */
343                 lwp->lwp_sysabort = 0;
344                 (void) set_errno(EINTR);        /* forces post_sys */
345                 t->t_pre_sys = 1;       /* repost anyway */
346                 return (1);             /* don't do system call, return EINTR */
```



```
347         }

349         /*
350          * begin auditing for this syscall if the c2audit module is loaded
351          * and auditing is enabled
352          */
353         if (audit_active == C2AUDIT_LOADED) {
354                 uint32_t auditing = au_zone_getstate(NULL);

356                 if (auditing & AU_AUDIT_MASK) {
357                         int error;
358                         if (error = audit_start(T_SYSCALL, code, auditing, \
359                             0, lwp)) {
360                                 t->t_pre_sys = 1;       /* repost anyway */
361                                 (void) set_errno(error);
362                                 return (1);
363                         }
364                         repost = 1;
365                 }
366         }

368 #ifndef NPROBE
369         /* Kernel probe */
370         if (tnf_tracing_active) {
371                 TNF_PROBE_1(syscall_start, "syscall thread", /* CSTYLED */,
372                         tnf_sysnum,     sysnum,         t->t_sysnum);
373                 t->t_post_sys = 1;      /* make sure post_syscall runs */
374                 repost = 1;
375         }
376 #endif /* NPROBE */

378 #ifdef SYSCALLTRACE
379         if (syscalltrace) {
380                 int i;
381                 long *ap;
382                 char *cp;
383                 char *sysname;
384                 struct sysent *callp;

386                 if (code >= NSYSCALL)
387                         callp = &nosys_ent;     /* nosys has no args */
388                 else
389                         callp = LWP_GETSYSENT(lwp) + code;
390                 (void) save_syscall_args();
391                 mutex_enter(&systrace_lock);
392                 printf("%d: ", p->p_pid);
393                 if (code >= NSYSCALL)
394                         printf("0x%x", code);
395                 else {
396                         sysname = mod_getsysname(code);
397                         printf("%s[0x%x/0x%p]", sysname == NULL ? "NULL" :
398                             sysname, code, callp->sy_callc);
399                 }
400                 cp = "(";
401                 for (i = 0, ap = lwp->lwp_ap; i < callp->sy_narg; i++, ap++) {
402                         printf("%s%lx", cp, *ap);
403                         cp = ", ";
404                 }
405                 if (i)
406                         printf(")");
407                 printf(" %s id=0x%p\n", PTOU(p)->u_comm, curthread);
408                 mutex_exit(&systrace_lock);
409         }
410 #endif /* SYSCALLTRACE */

412         /*
```

```
413          * If there was a continuing reason for pre-syscall processing,
414          * set the t_pre_sys flag for the next system call.
415          */
416         if (repost)
417                 t->t_pre_sys = 1;
418         lwp->lwp_error = 0;     /* for old drivers */
419         lwp->lwp_badpriv = PRIV_NONE;
420         return (0);
421 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   31077 Fri Mar 28 23:33:54 2014
new/usr/src/uts/sparc/os/syscall.c
patch fix-compile2
**********************************************************
______unchanged_portion_omitted_

 342 /*
 343  * Perform pre-system-call processing, including stopping for tracing,
 344  * auditing, microstate-accounting, etc.
 345  *
 346  * This routine is called only if the t_pre_sys flag is set.  Any condition
 347  * requiring pre-syscall handling must set the t_pre_sys flag.  If the
 348  * condition is persistent, this routine will repost t_pre_sys.
 349  */
 350 int
 351 pre_syscall(int arg0)
 352 {
 353         unsigned int code;
 354         kthread_t *t = curthread;
 355         proc_t *p = ttoproc(t);
 356         klwp_t *lwp = ttolwp(t);
 357         struct regs *rp = lwptoregs(lwp);
 358         int     repost;

 360         t->t_pre_sys = repost = 0;      /* clear pre-syscall processing flag */

 362         ASSERT(t->t_schedflag & TS_DONT_SWAP);

 362         syscall_mstate(LMS_USER, LMS_SYSTEM);

 364         /*
 365          * The syscall arguments in the out registers should be pointed to
 366          * by lwp_ap.  If the args need to be copied so that the outs can
 367          * be changed without losing the ability to get the args for /proc,
 368          * they can be saved by save_syscall_args(), and lwp_ap will be
 369          * restored by post_syscall().
 370          */
 371         ASSERT(lwp->lwp_ap == (long *)&rp->r_o0);

 373         /*
 374          * Make sure the thread is holding the latest credentials for the
 375          * process.  The credentials in the process right now apply to this
 376          * thread for the entire system call.
 377          */
 378         if (t->t_cred != p->p_cred) {
 379                 cred_t *oldcred = t->t_cred;
 380                 /*
 381                  * DTrace accesses t_cred in probe context.  t_cred must
 382                  * always be either NULL, or point to a valid, allocated cred
 383                  * structure.
 384                  */
 385                 t->t_cred = crgetcred();
 386                 crfree(oldcred);
 387         }

 389         /*
 390          * Undo special arrangements to single-step the lwp
 391          * so that a debugger will see valid register contents.
 392          * Also so that the pc is valid for syncfpu().
 393          * Also so that a syscall like exec() can be stepped.
 394          */
 395         if (lwp->lwp_pcb.pcb_step != STEP_NONE) {
 396                 (void) prundostep();
 397                 repost = 1;
 398         }
```



```
 400         /*
 401          * Check for indirect system call in case we stop for tracing.
 402          * Don't allow multiple indirection.
 403          */
 404         code = t->t_sysnum;
 405         if (code == 0 && arg0 != 0) {           /* indirect syscall */
 406                 code = arg0;
 407                 t->t_sysnum = arg0;
 408         }

 410         /*
 411          * From the proc(4) manual page:
 412          * When entry to a system call is being traced, the traced process
 413          * stops after having begun the call to the system but before the
 414          * system call arguments have been fetched from the process.
 415          * If proc changes the args we must refetch them after starting.
 416          */
 417         if (PTOU(p)->u_systrap) {
 418                 if (prismember(&PTOU(p)->u_entrymask, code)) {
 419                         /*
 420                          * Recheck stop condition, now that lock is held.
 421                          */
 422                         mutex_enter(&p->p_lock);
 423                         if (PTOU(p)->u_systrap &&
 424                             prismember(&PTOU(p)->u_entrymask, code)) {
 425                                 stop(PR_SYSENTRY, code);
 426                                 /*
 427                                  * Must refetch args since they were
 428                                  * possibly modified by /proc.  Indicate
 429                                  * that the valid copy is in the
 430                                  * registers.
 431                                  */
 432                                 lwp->lwp_argsaved = 0;
 433                                 lwp->lwp_ap = (long *)&rp->r_o0;
 434                         }
 435                         mutex_exit(&p->p_lock);
 436                 }
 437                 repost = 1;
 438         }

 440         if (lwp->lwp_sysabort) {
 441                 /*
 442                  * lwp_sysabort may have been set via /proc while the process
 443                  * was stopped on PR_SYSENTRY.  If so, abort the system call.
 444                  * Override any error from the copyin() of the arguments.
 445                  */
 446                 lwp->lwp_sysabort = 0;
 447                 (void) set_errno(EINTR); /* sets post-sys processing */
 448                 t->t_pre_sys = 1;       /* repost anyway */
 449                 return (1);             /* don't do system call, return EINTR */
 450         }

 452         /* begin auditing for this syscall */
 453         if (audit_active == C2AUDIT_LOADED) {
 454                 uint32_t auditing = au_zone_getstate(NULL);

 456                 if (auditing & AU_AUDIT_MASK) {
 457                         int error;
 458                         if (error = audit_start(T_SYSCALL, code, auditing, \
 459                             0, lwp)) {
 460                                 t->t_pre_sys = 1;       /* repost anyway */
 461                                 lwp->lwp_error = 0;     /* for old drivers */
 462                                 return (error);
 463                         }
 464                         repost = 1;
```

```
465                 }
466         }

468 #ifndef NPROBE
469         /* Kernel probe */
470         if (tnf_tracing_active) {
471                 TNF_PROBE_1(syscall_start, "syscall thread", /* CSTYLED */,
472                         tnf_sysnum,     sysnum,         t->t_sysnum);
473                 t->t_post_sys = 1;      /* make sure post_syscall runs */
474                 repost = 1;
475         }
476 #endif /* NPROBE */

478 #ifdef SYSCALLTRACE
479         if (syscalltrace) {
480                 int i;
481                 long *ap;
482                 char *cp;
483                 char *sysname;
484                 struct sysent *callp;

486                 if (code >= NSYSCALL)
487                         callp = &nosys_ent;     /* nosys has no args */
488                 else
489                         callp = LWP_GETSYSENT(lwp) + code;
490                 (void) save_syscall_args();
491                 mutex_enter(&systrace_lock);
492                 printf("%d: ", p->p_pid);
493                 if (code >= NSYSCALL)
494                         printf("0x%x", code);
495                 else {
496                         sysname = mod_getsysname(code);
497                         printf("%s[0x%x]", sysname == NULL ? "NULL" :
498                             sysname, code);
499                 }
500                 cp = "(";
501                 for (i = 0, ap = lwp->lwp_ap; i < callp->sy_narg; i++, ap++) {
502                         printf("%s%lx", cp, *ap);
503                         cp = ", ";
504                 }
505                 if (i)
506                         printf(")");
507                 printf(" %s id=0x%p\n", PTOU(p)->u_comm, curthread);
508                 mutex_exit(&systrace_lock);
509         }
510 #endif /* SYSCALLTRACE */

512         /*
513          * If there was a continuing reason for pre-syscall processing,
514          * set the t_pre_sys flag for the next system call.
515          */
516         if (repost)
517                 t->t_pre_sys = 1;
518         lwp->lwp_error = 0;     /* for old drivers */
519         lwp->lwp_badpriv = PRIV_NONE;   /* for privilege tracing */
520         return (0);
521 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   50033 Fri Mar 28 23:33:55 2014
new/usr/src/uts/sparc/v9/os/v9dep.c
patch sched-cleanup
**********************************************************
______unchanged_portion_omitted_

863 void
864 lwp_swapin(kthread_t *tp)
865 {
866         struct machpcb *mpcb = lwptompcb(ttolwp(tp));
867
868         mpcb->mpcb_pa = va_to_pa(mpcb);
869         mpcb->mpcb_wbuf_pa = va_to_pa(mpcb->mpcb_wbuf);
870 }

863 /*
864  * Construct the execution environment for the user's signal
865  * handler and arrange for control to be given to it on return
866  * to userland.  The library code now calls setcontext() to
867  * clean up after the signal handler, so sigret() is no longer
868  * needed.
869  */
870 int
871 sendsig(int sig, k_siginfo_t *sip, void (*hdlr)())
872 {
873         /*
874          * 'volatile' is needed to ensure that values are
875          * correct on the error return from on_fault().
876          */
877         volatile int minstacksz; /* min stack required to catch signal */
878         int newstack = 0;       /* if true, switching to altstack */
879         label_t ljb;
880         caddr_t sp;
881         struct regs *volatile rp;
882         klwp_t *lwp = ttolwp(curthread);
883         proc_t *volatile p = ttoproc(curthread);
884         int fpq_size = 0;
885         struct sigframe {
886                 struct frame frwin;
887                 ucontext_t uc;
888         };
889         siginfo_t *sip_addr;
890         struct sigframe *volatile fp;
891         ucontext_t *volatile tuc = NULL;
892         char *volatile xregs = NULL;
893         volatile size_t xregs_size = 0;
894         gwindows_t *volatile gwp = NULL;
895         volatile int gwin_size = 0;
896         kfpu_t *fpp;
897         struct machpcb *mpcb;
898         volatile int watched = 0;
899         volatile int watched2 = 0;
900         caddr_t tos;

902         /*
903          * Make sure the current last user window has been flushed to
904          * the stack save area before we change the sp.
905          * Restore register window if a debugger modified it.
906          */
907         (void) flush_user_windows_to_stack(NULL);
908         if (lwp->lwp_pcb.pcb_xregstat != XREGNONE)
909                 xregrestore(lwp, 0);

911         mpcb = lwptompcb(lwp);
912         rp = lwptoregs(lwp);
```



```
914         /*
915          * Clear the watchpoint return stack pointers.
916          */
917         mpcb->mpcb_rsp[0] = NULL;
918         mpcb->mpcb_rsp[1] = NULL;

920         minstacksz = sizeof (struct sigframe);

922         /*
923          * We know that sizeof (siginfo_t) is stack-aligned:
924          * 128 bytes for ILP32, 256 bytes for LP64.
925          */
926         if (sip != NULL)
927                 minstacksz += sizeof (siginfo_t);

929         /*
930          * These two fields are pointed to by ABI structures and may
931          * be of arbitrary length. Size them now so we know how big
932          * the signal frame has to be.
933          */
934         fpp = lwptofpu(lwp);
935         fpp->fpu_fprs = _fp_read_fprs();
936         if ((fpp->fpu_en) || (fpp->fpu_fprs & FPRS_FEF)) {
937                 fpq_size = fpp->fpu_q_entrysize * fpp->fpu_qcnt;
938                 minstacksz += SA(fpq_size);
939         }

941         mpcb = lwptompcb(lwp);
942         if (mpcb->mpcb_wbcnt != 0) {
943                 gwin_size = (mpcb->mpcb_wbcnt * sizeof (struct rwindow)) +
944                     (SPARC_MAXREGWINDOW * sizeof (caddr_t)) + sizeof (long);
945                 minstacksz += SA(gwin_size);
946         }

948         /*
949          * Extra registers, if support by this platform, may be of arbitrary
950          * length. Size them now so we know how big the signal frame has to be.
951          * For sparcv9 _LP64 user programs, use asrs instead of the xregs.
952          */
953         minstacksz += SA(xregs_size);

955         /*
956          * Figure out whether we will be handling this signal on
957          * an alternate stack specified by the user. Then allocate
958          * and validate the stack requirements for the signal handler
959          * context. on_fault will catch any faults.
960          */
961         newstack = (sigismember(&PTOU(curproc)->u_sigonstack, sig) &&
962             !(lwp->lwp_sigaltstack.ss_flags & (SS_ONSTACK|SS_DISABLE)));

964         tos = (caddr_t)rp->r_sp + STACK_BIAS;
965         /*
966          * Force proper stack pointer alignment, even in the face of a
967          * misaligned stack pointer from user-level before the signal.
968          * Don't use the SA() macro because that rounds up, not down.
969          */
970         tos = (caddr_t)((uintptr_t)tos & ~(STACK_ALIGN - 1ul));

972         if (newstack != 0) {
973                 fp = (struct sigframe *)
974                     (SA((uintptr_t)lwp->lwp_sigaltstack.ss_sp) +
975                     SA((int)lwp->lwp_sigaltstack.ss_size) - STACK_ALIGN -
976                     SA(minstacksz));
977         } else {
978                 /*
```

```
 979                  * If we were unable to flush all register windows to
 980                  * the stack and we are not now on an alternate stack,
 981                  * just dump core with a SIGSEGV back in psig().
 982                  */
 983                 if (sig == SIGSEGV &&
 984                     mpcb->mpcb_wbcnt != 0 &&
 985                     !(lwp->lwp_sigaltstack.ss_flags & SS_ONSTACK))
 986                         return (0);
 987                 fp = (struct sigframe *)(tos - SA(minstacksz));
 988                 /*
 989                  * Could call grow here, but stack growth now handled below
 990                  * in code protected by on_fault().
 991                  */
 992         }
 993         sp = (caddr_t)fp + sizeof (struct sigframe);

 995         /*
 996          * Make sure process hasn't trashed its stack.
 997          */
 998         if ((caddr_t)fp >= p->p_usrstack ||
 999             (caddr_t)fp + SA(minstacksz) >= p->p_usrstack) {
1000 #ifdef DEBUG
1001                 printf("sendsig: bad signal stack cmd=%s, pid=%d, sig=%d\n",
1002                     PTOU(p)->u_comm, p->p_pid, sig);
1003                 printf("sigsp = 0x%p, action = 0x%p, upc = 0x%lx\n",
1004                     (void *)fp, (void *)hdlr, rp->r_pc);
1005                 printf("fp above USRSTACK\n");
1006 #endif
1007                 return (0);
1008         }

1010         watched = watch_disable_addr((caddr_t)fp, SA(minstacksz), S_WRITE);
1011         if (on_fault(&ljb))
1012                 goto badstack;

1014         tuc = kmem_alloc(sizeof (ucontext_t), KM_SLEEP);
1015         savecontext(tuc, &lwp->lwp_sigoldmask);

1017         /*
1018          * save extra register state if it exists
1019          */
1020         if (xregs_size != 0) {
1021                 xregs_setptr(lwp, tuc, sp);
1022                 xregs = kmem_alloc(xregs_size, KM_SLEEP);
1023                 xregs_get(lwp, xregs);
1024                 copyout_noerr(xregs, sp, xregs_size);
1025                 kmem_free(xregs, xregs_size);
1026                 xregs = NULL;
1027                 sp += SA(xregs_size);
1028         }

1030         copyout_noerr(tuc, &fp->uc, sizeof (*tuc));
1031         kmem_free(tuc, sizeof (*tuc));
1032         tuc = NULL;

1034         if (sip != NULL) {
1035                 zoneid_t zoneid;

1037                 uzero(sp, sizeof (siginfo_t));
1038                 if (SI_FROMUSER(sip) &&
1039                     (zoneid = p->p_zone->zone_id) != GLOBAL_ZONEID &&
1040                     zoneid != sip->si_zoneid) {
1041                         k_siginfo_t sani_sip = *sip;
1042                         sani_sip.si_pid = p->p_zone->zone_zsched->p_pid;
1043                         sani_sip.si_uid = 0;
1044                         sani_sip.si_ctid = -1;
```



```
1045                         sani_sip.si_zoneid = zoneid;
1046                         copyout_noerr(&sani_sip, sp, sizeof (sani_sip));
1047                 } else {
1048                         copyout_noerr(sip, sp, sizeof (*sip));
1049                 }
1050                 sip_addr = (siginfo_t *)sp;
1051                 sp += sizeof (siginfo_t);

1053                 if (sig == SIGPROF &&
1054                     curthread->t_rprof != NULL &&
1055                     curthread->t_rprof->rp_anystate) {
1056                         /*
1057                          * We stand on our head to deal with
1058                          * the real time profiling signal.
1059                          * Fill in the stuff that doesn't fit
1060                          * in a normal k_siginfo structure.
1061                          */
1062                         int i = sip->si_nsysarg;
1063                         while (--i >= 0) {
1064                                 sulword_noerr(
1065                                     (ulong_t *)&sip_addr->si_sysarg[i],
1066                                     (ulong_t)lwp->lwp_arg[i]);
1067                         }
1068                         copyout_noerr(curthread->t_rprof->rp_state,
1069                             sip_addr->si_mstate,
1070                             sizeof (curthread->t_rprof->rp_state));
1071                 }
1072         } else {
1073                 sip_addr = (siginfo_t *)NULL;
1074         }

1076         /*
1077          * When flush_user_windows_to_stack() can't save all the
1078          * windows to the stack, it puts them in the lwp's pcb.
1079          */
1080         if (gwin_size != 0) {
1081                 gwp = kmem_alloc(gwin_size, KM_SLEEP);
1082                 getgwins(lwp, gwp);
1083                 sulword_noerr(&fp->uc.uc_mcontext.gwins, (ulong_t)sp);
1084                 copyout_noerr(gwp, sp, gwin_size);
1085                 kmem_free(gwp, gwin_size);
1086                 gwp = NULL;
1087                 sp += SA(gwin_size);
1088         } else
1089                 sulword_noerr(&fp->uc.uc_mcontext.gwins, (ulong_t)NULL);

1091         if (fpq_size != 0) {
1092                 struct fq *fqp = (struct fq *)sp;
1093                 sulword_noerr(&fp->uc.uc_mcontext.fpregs.fpu_q, (ulong_t)fqp);
1094                 copyout_noerr(mpcb->mpcb_fpu_q, fqp, fpq_size);

1096                 /*
1097                  * forget the fp queue so that the signal handler can run
1098                  * without being harrassed--it will do a setcontext that will
1099                  * re-establish the queue if there still is one
1100                  *
1101                  * NOTE: fp_runq() relies on the qcnt field being zeroed here
1102                  *      to terminate its processing of the queue after signal
1103                  *      delivery.
1104                  */
1105                 mpcb->mpcb_fpu->fpu_qcnt = 0;
1106                 sp += SA(fpq_size);

1108                 /* Also, syscall needs to know about this */
1109                 mpcb->mpcb_flags |= FP_TRAPPED;
```

```
1111         } else {
1112                 sulword_noerr(&fp->uc.uc_mcontext.fpregs.fpu_q, (ulong_t)NULL);
1113                 suword8_noerr(&fp->uc.uc_mcontext.fpregs.fpu_qcnt, 0);
1114         }


1117         /*
1118          * Since we flushed the user's windows and we are changing his
1119          * stack pointer, the window that the user will return to will
1120          * be restored from the save area in the frame we are setting up.
1121          * We copy in save area for old stack pointer so that debuggers
1122          * can do a proper stack backtrace from the signal handler.
1123          */
1124         if (mpcb->mpcb_wbcnt == 0) {
1125                 watched2 = watch_disable_addr(tos, sizeof (struct rwindow),
1126                     S_READ);
1127                 ucopy(tos, &fp->frwin, sizeof (struct rwindow));
1128         }

1130         lwp->lwp_oldcontext = (uintptr_t)&fp->uc;

1132         if (newstack != 0) {
1133                 lwp->lwp_sigaltstack.ss_flags |= SS_ONSTACK;

1135                 if (lwp->lwp_ustack) {
1136                         copyout_noerr(&lwp->lwp_sigaltstack,
1137                             (stack_t *)lwp->lwp_ustack, sizeof (stack_t));
1138                 }
1139         }

1141         no_fault();
1142         mpcb->mpcb_wbcnt = 0;           /* let user go on */

1144         if (watched2)
1145                 watch_enable_addr(tos, sizeof (struct rwindow), S_READ);
1146         if (watched)
1147                 watch_enable_addr((caddr_t)fp, SA(minstacksz), S_WRITE);

1149         /*
1150          * Set up user registers for execution of signal handler.
1151          */
1152         rp->r_sp = (uintptr_t)fp - STACK_BIAS;
1153         rp->r_pc = (uintptr_t)hdlr;
1154         rp->r_npc = (uintptr_t)hdlr + 4;
1155         /* make sure %asi is ASI_PNF */
1156         rp->r_tstate &= ~((uint64_t)TSTATE_ASI_MASK << TSTATE_ASI_SHIFT);
1157         rp->r_tstate |= ((uint64_t)ASI_PNF << TSTATE_ASI_SHIFT);
1158         rp->r_o0 = sig;
1159         rp->r_o1 = (uintptr_t)sip_addr;
1160         rp->r_o2 = (uintptr_t)&fp->uc;
1161         /*
1162          * Don't set lwp_eosys here.  sendsig() is called via psig() after
1163          * lwp_eosys is handled, so setting it here would affect the next
1164          * system call.
1165          */
1166         return (1);

1168 badstack:
1169         no_fault();
1170         if (watched2)
1171                 watch_enable_addr(tos, sizeof (struct rwindow), S_READ);
1172         if (watched)
1173                 watch_enable_addr((caddr_t)fp, SA(minstacksz), S_WRITE);
1174         if (tuc)
1175                 kmem_free(tuc, sizeof (ucontext_t));
1176         if (xregs)
```



```
1177                 kmem_free(xregs, xregs_size);
1178         if (gwp)
1179                 kmem_free(gwp, gwin_size);
1180 #ifdef DEBUG
1181         printf("sendsig: bad signal stack cmd=%s, pid=%d, sig=%d\n",
1182             PTOU(p)->u_comm, p->p_pid, sig);
1183         printf("on fault, sigsp = %p, action = %p, upc = 0x%lx\n",
1184             (void *)fp, (void *)hdlr, rp->r_pc);
1185 #endif
1186         return (0);
1187 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   14116 Fri Mar 28 23:33:57 2014
new/usr/src/uts/sun4/os/mlsetup.c
patch fix-compile2
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/types.h>
#include <sys/systm.h>
#include <sys/archsystm.h>
#include <sys/machsystm.h>
#include <sys/disp.h>
#include <sys/autoconf.h>
#include <sys/promif.h>
#include <sys/prom_plat.h>
#include <sys/promimpl.h>
#include <sys/platform_module.h>
#include <sys/clock.h>
#include <sys/pte.h>
#include <sys/scb.h>
#include <sys/cpu.h>
#include <sys/stack.h>
#include <sys/intreg.h>
#include <sys/ivintr.h>
#include <vm/as.h>
#include <vm/hat_sfmmu.h>
#include <sys/reboot.h>
#include <sys/sysmacros.h>
#include <sys/vtrace.h>
#include <sys/trap.h>
#include <sys/machtrap.h>
#include <sys/privregs.h>
#include <sys/machpcb.h>
#include <sys/proc.h>
#include <sys/cpupart.h>
#include <sys/pset.h>
#include <sys/cpu_module.h>
#include <sys/copyops.h>
#include <sys/panic.h>
#include <sys/bootconf.h>       /* for bootops */
#include <sys/pg.h>
#include <sys/kdi.h>
#include <sys/fpras.h>
```



```
#include <sys/prom_debug.h>
#include <sys/debug.h>

#include <sys/sunddi.h>
#include <sys/lgrp.h>
#include <sys/traptrace.h>

#include <sys/kobj_impl.h>
#include <sys/kdi_machimpl.h>

/*
 * External Routines:
 */
extern void map_wellknown_devices(void);
extern void hsvc_setup(void);
extern void mach_descrip_startup_init(void);
extern void mach_soft_state_init(void);

int     dcache_size;
int     dcache_linesize;
int     icache_size;
int     icache_linesize;
int     ecache_size;
int     ecache_alignsize;
int     ecache_associativity;
int     ecache_setsize;                 /* max possible e$ setsize */
int     cpu_setsize;                    /* max e$ setsize of configured cpus */
int     dcache_line_mask;               /* spitfire only */
int     vac_size;                       /* cache size in bytes */
uint_t  vac_mask;                       /* VAC alignment consistency mask */
int     vac_shift;                      /* log2(vac_size) for ppmapout() */
int     vac = 0;        /* virtual address cache type (none == 0) */

/*
 * fpRAS.  An individual sun4* machine class (or perhaps subclass,
 * eg sun4u/cheetah) must set fpras_implemented to indicate that it implements
 * the fpRAS feature.  The feature can be suppressed by setting fpras_disable
 * or the mechanism can be disabled for individual copy operations with
 * fpras_disableids.  All these are checked in post_startup() code so
 * fpras_disable and fpras_disableids can be set in /etc/system.
 * If/when fpRAS is implemented on non-sun4 architectures these
 * definitions will need to move up to the common level.
 */
int     fpras_implemented;
int     fpras_disable;
int     fpras_disableids;

/*
 * Static Routines:
 */
static void kern_splr_preprom(void);
static void kern_splx_postprom(void);

/*
 * Setup routine called right before main(). Interposing this function
 * before main() allows us to call it in a machine-independent fashion.
 */

void
mlsetup(struct regs *rp, kfpu_t *fp)
{
        struct machpcb *mpcb;

        extern char t0stack[];
        extern struct classfuncs sys_classfuncs;
```

```
128         extern disp_t cpu0_disp;
129         unsigned long long pa;

131 #ifdef TRAPTRACE
132         TRAP_TRACE_CTL *ctlp;
133 #endif /* TRAPTRACE */

135         /* drop into kmdb on boot -d */
136         if (boothowto & RB_DEBUGENTER)
137                 kmdb_enter();

139         /*
140          * initialize cpu_self
141          */
142         cpu0.cpu_self = &cpu0;

144         /*
145          * initialize t0
146          */
147         t0.t_stk = (caddr_t)rp - REGOFF;
148         /* Can't use va_to_pa here - wait until prom_ initialized */
149         t0.t_stkbase = t0stack;
150         t0.t_pri = maxclsyspri - 3;
151         t0.t_schedflag = 0;
151         t0.t_schedflag = TS_LOAD | TS_DONT_SWAP;
152         t0.t_procp = &p0;
153         t0.t_plockp = &p0lock.pl_lock;
154         t0.t_lwp = &lwp0;
155         t0.t_forw = &t0;
156         t0.t_back = &t0;
157         t0.t_next = &t0;
158         t0.t_prev = &t0;
159         t0.t_cpu = &cpu0;                       /* loaded by _start */
160         t0.t_disp_queue = &cpu0_disp;
161         t0.t_bind_cpu = PBIND_NONE;
162         t0.t_bind_pset = PS_NONE;
163         t0.t_bindflag = (uchar_t)default_binding_mode;
164         t0.t_cpupart = &cp_default;
165         t0.t_clfuncs = &sys_classfuncs.thread;
166         t0.t_copyops = NULL;
167         THREAD_ONPROC(&t0, CPU);

169         lwp0.lwp_thread = &t0;
170         lwp0.lwp_procp = &p0;
171         lwp0.lwp_regs = (void *)rp;
172         t0.t_tid = p0.p_lwpcnt = p0.p_lwprcnt = p0.p_lwpid = 1;

174         mpcb = lwptompcb(&lwp0);
175         mpcb->mpcb_fpu = fp;
176         mpcb->mpcb_fpu->fpu_q = mpcb->mpcb_fpu_q;
177         mpcb->mpcb_thread = &t0;
178         lwp0.lwp_fpu = (void *)mpcb->mpcb_fpu;

180         p0.p_exec = NULL;
181         p0.p_stat = SRUN;
182         p0.p_flag = SSYS;
183         p0.p_tlist = &t0;
184         p0.p_stksize = 2*PAGESIZE;
185         p0.p_stkpageszc = 0;
186         p0.p_as = &kas;
187         p0.p_lockp = &p0lock;
188         p0.p_utraps = NULL;
189         p0.p_brkpageszc = 0;
190         p0.p_t1_lgrpid = LGRP_NONE;
191         p0.p_tr_lgrpid = LGRP_NONE;
192         sigorset(&p0.p_ignore, &ignoredefault);
```



```
194         CPU->cpu_thread = &t0;
195         CPU->cpu_dispthread = &t0;
196         bzero(&cpu0_disp, sizeof (disp_t));
197         CPU->cpu_disp = &cpu0_disp;
198         CPU->cpu_disp->disp_cpu = CPU;
199         CPU->cpu_idle_thread = &t0;
200         CPU->cpu_flags = CPU_RUNNING;
201         CPU->cpu_id = getprocessorid();
202         CPU->cpu_dispatch_pri = t0.t_pri;

204         /*
205          * Initialize thread/cpu microstate accounting
206          */
207         init_mstate(&t0, LMS_SYSTEM);
208         init_cpu_mstate(CPU, CMS_SYSTEM);

210         /*
211          * Initialize lists of available and active CPUs.
212          */
213         cpu_list_init(CPU);

215         cpu_vm_data_init(CPU);

217         pg_cpu_bootstrap(CPU);

219         (void) prom_set_preprom(kern_splr_preprom);
220         (void) prom_set_postprom(kern_splx_postprom);
221         PRM_INFO("mlsetup: now ok to call prom_printf");

223         mpcb->mpcb_pa = va_to_pa(t0.t_stk);

225         /*
226          * Claim the physical and virtual resources used by panicbuf,
227          * then map panicbuf.  This operation removes the phys and
228          * virtual addresses from the free lists.
229          */
230         if (prom_claim_virt(PANICBUFSIZE, panicbuf) != panicbuf)
231                 prom_panic("Can't claim panicbuf virtual address");

233         if (prom_retain("panicbuf", PANICBUFSIZE, MMU_PAGESIZE, &pa) != 0)
234                 prom_panic("Can't allocate retained panicbuf physical address");

236         if (prom_map_phys(-1, PANICBUFSIZE, panicbuf, pa) != 0)
237                 prom_panic("Can't map panicbuf");

239         PRM_DEBUG(panicbuf);
240         PRM_DEBUG(pa);

242         /*
243          * Negotiate hypervisor services, if any
244          */
245         hsvc_setup();
246         mach_soft_state_init();

248 #ifdef TRAPTRACE
249         /*
250          * initialize the trap trace buffer for the boot cpu
251          * XXX todo, dynamically allocate this buffer too
252          */
253         ctlp = &trap_trace_ctl[CPU->cpu_id];
254         ctlp->d.vaddr_base = trap_tr0;
255         ctlp->d.offset = ctlp->d.last_offset = 0;
256         ctlp->d.limit = TRAP_TSIZE;             /* XXX dynamic someday */
257         ctlp->d.paddr_base = va_to_pa(trap_tr0);
258 #endif /* TRAPTRACE */
```

```
260         /*
261          * Initialize the Machine Description kernel framework
262          */

264         mach_descrip_startup_init();

266         /*
267          * initialize HV trap trace buffer for the boot cpu
268          */
269         mach_htraptrace_setup(CPU->cpu_id);
270         mach_htraptrace_configure(CPU->cpu_id);

272         /*
273          * lgroup framework initialization. This must be done prior
274          * to devices being mapped.
275          */
276         lgrp_init(LGRP_INIT_STAGE1);

278         cpu_setup();

280         if (boothowto & RB_HALT) {
281                 prom_printf("unix: kernel halted by -h flag\n");
282                 prom_enter_mon();
283         }

285         setcputype();
286         map_wellknown_devices();
287         setcpudelay();
288 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   51350 Fri Mar 28 23:34:00 2014
new/usr/src/uts/sun4/os/trap.c
patch fix-compile2
**********************************************************
______unchanged_portion_omitted_

#if defined(SF_ERRATA_23) || defined(SF_ERRATA_30) /* call ... illegal-insn */
int     ill_calls;
#endif

/*
 * Currently, the only PREFETCH/PREFETCHA instructions which cause traps
 * are the "strong" prefetches (fcn=20-23).  But we check for all flavors of
 * PREFETCH, in case some future variant also causes a DATA_MMU_MISS.
 */
#define IS_PREFETCH(i)  (((i) & 0xc1780000) == 0xc1680000)

#define IS_FLUSH(i)     (((i) & 0xc1f80000) == 0x81d80000)
#define IS_SWAP(i)      (((i) & 0xc1f80000) == 0xc0780000)
#define IS_LDSTUB(i)    (((i) & 0xc1f80000) == 0xc0680000)
#define IS_FLOAT(i)     (((i) & 0x1000000) != 0)
#define IS_STORE(i)     (((i) >> 21) & 1)

/*
 * Called from the trap handler when a processor trap occurs.
 */
/*VARARGS2*/
void
trap(struct regs *rp, caddr_t addr, uint32_t type, uint32_t mmu_fsr)
{
        proc_t *p = ttoproc(curthread);
        klwp_id_t lwp = ttolwp(curthread);
        struct machpcb *mpcb = NULL;
        k_siginfo_t siginfo;
        uint_t op3, fault = 0;
        int stepped = 0;
        greg_t oldpc;
        int mstate;
        char *badaddr;
        faultcode_t res;
        enum fault_type fault_type;
        enum seg_rw rw;
        uintptr_t lofault;
        label_t *onfault;
        int instr;
        int iskernel;
        int watchcode;
        int watchpage;
        extern faultcode_t pagefault(caddr_t, enum fault_type,
            enum seg_rw, int);
#ifdef sun4v
        extern boolean_t tick_stick_emulation_active;
#endif  /* sun4v */

        CPU_STATS_ADDQ(CPU, sys, trap, 1);

#ifdef SF_ERRATA_23 /* call causes illegal-insn */
        ASSERT((curthread->t_schedflag & TS_DONT_SWAP) ||
            (type == T_UNIMP_INSTR));
#else
        ASSERT(curthread->t_schedflag & TS_DONT_SWAP);
#endif /* SF_ERRATA_23 */

        if (USERMODE(rp->r_tstate) || (type & T_USER)) {
                /*
```



```
                 * Set lwp_state before trying to acquire any
                 * adaptive lock
                 */
                ASSERT(lwp != NULL);
                lwp->lwp_state = LWP_SYS;
                /*
                 * Set up the current cred to use during this trap. u_cred
                 * no longer exists.  t_cred is used instead.
                 * The current process credential applies to the thread for
                 * the entire trap.  If trapping from the kernel, this
                 * should already be set up.
                 */
                if (curthread->t_cred != p->p_cred) {
                        cred_t *oldcred = curthread->t_cred;
                        /*
                         * DTrace accesses t_cred in probe context.  t_cred
                         * must always be either NULL, or point to a valid,
                         * allocated cred structure.
                         */
                        curthread->t_cred = crgetcred();
                        crfree(oldcred);
                }
                type |= T_USER;
                ASSERT((type == (T_SYS_RTT_PAGE | T_USER)) ||
                    (type == (T_SYS_RTT_ALIGN | T_USER)) ||
                    lwp->lwp_regs == rp);
                mpcb = lwptompcb(lwp);
                switch (type) {
                case T_WIN_OVERFLOW + T_USER:
                case T_WIN_UNDERFLOW + T_USER:
                case T_SYS_RTT_PAGE + T_USER:
                case T_DATA_MMU_MISS + T_USER:
                        mstate = LMS_DFAULT;
                        break;
                case T_INSTR_MMU_MISS + T_USER:
                        mstate = LMS_TFAULT;
                        break;
                default:
                        mstate = LMS_TRAP;
                        break;
                }
                /* Kernel probe */
                TNF_PROBE_1(thread_state, "thread", /* CSTYLED */,
                    tnf_microstate, state, (char)mstate);
                mstate = new_mstate(curthread, mstate);
                siginfo.si_signo = 0;
                stepped =
                    lwp->lwp_pcb.pcb_step != STEP_NONE &&
                    ((oldpc = rp->r_pc), prundostep()) &&
                    mmu_btop((uintptr_t)addr) == mmu_btop((uintptr_t)oldpc);
                /* this assignment must not precede call to prundostep() */
                oldpc = rp->r_pc;
        }

        TRACE_1(TR_FAC_TRAP, TR_C_TRAP_HANDLER_ENTER,
            "C_trap_handler_enter:type %x", type);

#ifdef  F_DEFERRED
        /*
         * Take any pending floating point exceptions now.
         * If the floating point unit has an exception to handle,
         * just return to user-level to let the signal handler run.
         * The instruction that got us to trap() will be reexecuted on
         * return from the signal handler and we will trap to here again.
         * This is necessary to disambiguate simultaneous traps which
         * happen when a floating-point exception is pending and a
```

```
239          * machine fault is incurred.
240          */
241         if (type & USER) {
242                 /*
243                  * FP_TRAPPED is set only by sendsig() when it copies
244                  * out the floating-point queue for the signal handler.
245                  * It is set there so we can test it here and in syscall().
246                  */
247                 mpcb->mpcb_flags &= ~FP_TRAPPED;
248                 syncfpu();
249                 if (mpcb->mpcb_flags & FP_TRAPPED) {
250                         /*
251                          * trap() has have been called recursively and may
252                          * have stopped the process, so do single step
253                          * support for /proc.
254                          */
255                         mpcb->mpcb_flags &= ~FP_TRAPPED;
256                         goto out;
257                 }
258         }
259 #endif
260         switch (type) {
261                 case T_DATA_MMU_MISS:
262                 case T_INSTR_MMU_MISS + T_USER:
263                 case T_DATA_MMU_MISS + T_USER:
264                 case T_DATA_PROT + T_USER:
265                 case T_AST + T_USER:
266                 case T_SYS_RTT_PAGE + T_USER:
267                 case T_FLUSH_PCB + T_USER:
268                 case T_FLUSHW + T_USER:
269                         break;

271                 default:
272                         FTRACE_3("trap(): type=0x%lx, regs=0x%lx, addr=0x%lx",
273                             (ulong_t)type, (ulong_t)rp, (ulong_t)addr);
274                         break;
275         }

277         switch (type) {

279         default:
280                 /*
281                  * Check for user software trap.
282                  */
283                 if (type & T_USER) {
284                         if (tudebug)
285                                 showregs(type, rp, (caddr_t)0, 0);
286                         if ((type & ~T_USER) >= T_SOFTWARE_TRAP) {
287                                 bzero(&siginfo, sizeof (siginfo));
288                                 siginfo.si_signo = SIGILL;
289                                 siginfo.si_code  = ILL_ILLTRP;
290                                 siginfo.si_addr  = (caddr_t)rp->r_pc;
291                                 siginfo.si_trapno = type &~ T_USER;
292                                 fault = FLTILL;
293                                 break;
294                         }
295                 }
296                 addr = (caddr_t)rp->r_pc;
297                 (void) die(type, rp, addr, 0);
298                 /*NOTREACHED*/

300         case T_ALIGNMENT:       /* supv alignment error */
301                 if (nfload(rp, NULL))
302                         goto cleanup;

304                 if (curthread->t_lofault) {
```



```
305                         if (lodebug) {
306                                 showregs(type, rp, addr, 0);
307                                 traceback((caddr_t)rp->r_sp);
308                         }
309                         rp->r_g1 = EFAULT;
310                         rp->r_pc = curthread->t_lofault;
311                         rp->r_npc = rp->r_pc + 4;
312                         goto cleanup;
313                 }
314                 (void) die(type, rp, addr, 0);
315                 /*NOTREACHED*/

317         case T_INSTR_EXCEPTION:         /* sys instruction access exception */
318                 addr = (caddr_t)rp->r_pc;
319                 (void) die(type, rp, addr, mmu_fsr);
320                 /*NOTREACHED*/

322         case T_INSTR_MMU_MISS:          /* sys instruction mmu miss */
323                 addr = (caddr_t)rp->r_pc;
324                 (void) die(type, rp, addr, 0);
325                 /*NOTREACHED*/

327         case T_DATA_EXCEPTION:          /* system data access exception */
328                 switch (X_FAULT_TYPE(mmu_fsr)) {
329                 case FT_RANGE:
330                         /*
331                          * This happens when we attempt to dereference an
332                          * address in the address hole.  If t_ontrap is set,
333                          * then break and fall through to T_DATA_MMU_MISS /
334                          * T_DATA_PROT case below.  If lofault is set, then
335                          * honour it (perhaps the user gave us a bogus
336                          * address in the hole to copyin from or copyout to?)
337                          */

339                         if (curthread->t_ontrap != NULL)
340                                 break;

342                         addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
343                         if (curthread->t_lofault) {
344                                 if (lodebug) {
345                                         showregs(type, rp, addr, 0);
346                                         traceback((caddr_t)rp->r_sp);
347                                 }
348                                 rp->r_g1 = EFAULT;
349                                 rp->r_pc = curthread->t_lofault;
350                                 rp->r_npc = rp->r_pc + 4;
351                                 goto cleanup;
352                         }
353                         (void) die(type, rp, addr, mmu_fsr);
354                         /*NOTREACHED*/

356                 case FT_PRIV:
357                         /*
358                          * This can happen if we access ASI_USER from a kernel
359                          * thread.  To support pxfs, we need to honor lofault if
360                          * we're doing a copyin/copyout from a kernel thread.
361                          */

363                         if (nfload(rp, NULL))
364                                 goto cleanup;
365                         addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
366                         if (curthread->t_lofault) {
367                                 if (lodebug) {
368                                         showregs(type, rp, addr, 0);
369                                         traceback((caddr_t)rp->r_sp);
370                                 }
```

```
371                                 rp->r_g1 = EFAULT;
372                                 rp->r_pc = curthread->t_lofault;
373                                 rp->r_npc = rp->r_pc + 4;
374                                 goto cleanup;
375                         }
376                         (void) die(type, rp, addr, mmu_fsr);
377                         /*NOTREACHED*/
378
379                 default:
380                         if (nfload(rp, NULL))
381                                 goto cleanup;
382                         addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
383                         (void) die(type, rp, addr, mmu_fsr);
384                         /*NOTREACHED*/
385
386                 case FT_NFO:
387                         break;
388                 }
389                 /* fall into ... */
390
391         case T_DATA_MMU_MISS:           /* system data mmu miss */
392         case T_DATA_PROT:               /* system data protection fault */
393                 if (nfload(rp, &instr))
394                         goto cleanup;
395
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;
402
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");
406
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;
417
418                 mstate = new_mstate(curthread, LMS_KFAULT);
419
420                 switch (type) {
421                 case T_DATA_PROT:
422                         fault_type = F_PROT;
423                         rw = S_WRITE;
424                         break;
425                 case T_INSTR_MMU_MISS:
426                         fault_type = F_INVAL;
427                         rw = S_EXEC;
428                         break;
429                 case T_DATA_MMU_MISS:
430                 case T_DATA_EXCEPTION:
431                         /*
432                          * The hardware doesn't update the sfsr on mmu
433                          * misses so it is not easy to find out whether
434                          * the access was a read or a write so we need
435                          * to decode the actual instruction.
436                          */
```



```
437                         fault_type = F_INVAL;
438                         rw = get_accesstype(rp);
439                         break;
440                 default:
441                         cmn_err(CE_PANIC, "trap: unknown type %x", type);
442                         break;
443                 }
444                 /*
445                  * We determine if access was done to kernel or user
446                  * address space.  The addr passed into trap is really the
447                  * tag access register.
448                  */
449                 iskernel = (((uintptr_t)addr & TAGACC_CTX_MASK) == KCONTEXT);
450                 addr = (caddr_t)((uintptr_t)addr & TAGACC_VADDR_MASK);
451
452                 res = pagefault(addr, fault_type, rw, iskernel);
453                 if (!iskernel && res == FC_NOMAP &&
454                     addr < p->p_usrstack && grow(addr))
455                         res = 0;
456
457                 (void) new_mstate(curthread, mstate);
458
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;
465
466                 if (res == 0)
467                         goto cleanup;
468
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                 }
475
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;
496
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;
637
638                                 if (ismem) {
639                                         bzero(&siginfo, sizeof (siginfo));
640                                         siginfo.si_addr = addr;
641                                         (void) stop_on_fault(FLTPAGE, &siginfo);
642                                 }
643                                 goto out;
644                         }
645
646                         if (type != (T_INSTR_MMU_MISS + T_USER)) {
647                                 /*
648                                  * check for non-faulting loads, also
649                                  * fetch the instruction to check for
650                                  * flush
651                                  */
652                                 if (nfload(rp, &instr))
653                                         goto out;
654
655                                 /* skip userland prefetch instructions */
656                                 if (IS_PREFETCH(instr)) {
657                                         rp->r_pc = rp->r_npc;
658                                         rp->r_npc += 4;
659                                         goto out;
660                                         /*NOTREACHED*/
661                                 }
662
663                                 /*
664                                  * check if the instruction was a
665                                  * flush.  ABI allows users to specify
666                                  * an illegal address on the flush
667                                  * instruction so we simply return in
668                                  * this case.
669                                  *
670                                  * NB: the hardware should set a bit
671                                  * indicating this trap was caused by
672                                  * a flush instruction.  Instruction
673                                  * decoding is bugly!
674                                  */
675                                 if (IS_FLUSH(instr)) {
676                                         /* skip the flush instruction */
677                                         rp->r_pc = rp->r_npc;
678                                         rp->r_npc += 4;
679                                         goto out;
680                                         /*NOTREACHED*/
681                                 }
682                         } else if (res == FC_PROT) {
683                                 report_stack_exec(p, addr);
684                         }
685
686                         if (tudebug)
687                                 showregs(type, rp, addr, 0);
688                 }
689
690                 /*
691                  * In the case where both pagefault and grow fail,
692                  * set the code to the value provided by pagefault.
693                  */
694                 (void) instr_size(rp, &addr, rw);
695                 bzero(&siginfo, sizeof (siginfo));
696                 siginfo.si_addr = addr;
697                 if (FC_CODE(res) == FC_OBJERR) {
698                         siginfo.si_errno = FC_ERRNO(res);
699                         if (siginfo.si_errno != EINTR) {
700                                 siginfo.si_signo = SIGBUS;
```



```
701                                 siginfo.si_code = BUS_OBJERR;
702                                 fault = FLTACCESS;
703                         }
704                 } else { /* FC_NOMAP || FC_PROT */
705                         siginfo.si_signo = SIGSEGV;
706                         siginfo.si_code = (res == FC_NOMAP) ?
707                             SEGV_MAPERR : SEGV_ACCERR;
708                         fault = FLTBOUNDS;
709                 }
710                 /*
711                  * If this is the culmination of a single-step,
712                  * reset the addr, code, signal and fault to
713                  * indicate a hardware trace trap.
714                  */
715                 if (stepped) {
716                         pcb_t *pcb = &lwp->lwp_pcb;
717
718                         siginfo.si_signo = 0;
719                         fault = 0;
720                         if (pcb->pcb_step == STEP_WASACTIVE) {
721                                 pcb->pcb_step = STEP_NONE;
722                                 pcb->pcb_tracepc = NULL;
723                                 oldpc = rp->r_pc - 4;
724                         }
725                         /*
726                          * If both NORMAL_STEP and WATCH_STEP are in
727                          * effect, give precedence to WATCH_STEP.
728                          * One or the other must be set at this point.
729                          */
730                         ASSERT(pcb->pcb_flags & (NORMAL_STEP|WATCH_STEP));
731                         if ((fault = undo_watch_step(&siginfo)) == 0 &&
732                             (pcb->pcb_flags & NORMAL_STEP)) {
733                                 siginfo.si_signo = SIGTRAP;
734                                 siginfo.si_code = TRAP_TRACE;
735                                 siginfo.si_addr = (caddr_t)rp->r_pc;
736                                 fault = FLTTRACE;
737                         }
738                         pcb->pcb_flags &= ~(NORMAL_STEP|WATCH_STEP);
739                 }
740                 break;
741
742         case T_DATA_EXCEPTION + T_USER: /* user data access exception */
743
744                 if (&vis1_partial_support != NULL) {
745                         bzero(&siginfo, sizeof (siginfo));
746                         if (vis1_partial_support(rp,
747                             &siginfo, &fault) == 0)
748                                 goto out;
749                 }
750
751                 if (nfload(rp, &instr))
752                         goto out;
753                 if (IS_FLUSH(instr)) {
754                         /* skip the flush instruction */
755                         rp->r_pc = rp->r_npc;
756                         rp->r_npc += 4;
757                         goto out;
758                         /*NOTREACHED*/
759                 }
760                 bzero(&siginfo, sizeof (siginfo));
761                 siginfo.si_addr = addr;
762                 switch (X_FAULT_TYPE(mmu_fsr)) {
763                 case FT_ATOMIC_NC:
764                         if ((IS_SWAP(instr) && swap_nc(rp, instr)) ||
765                             (IS_LDSTUB(instr) && ldstub_nc(rp, instr))) {
766                                 /* skip the atomic */
```

```
767                                 rp->r_pc = rp->r_npc;
768                                 rp->r_npc += 4;
769                                 goto out;
770                         }
771                         /* fall into ... */
772                 case FT_PRIV:
773                         siginfo.si_signo = SIGSEGV;
774                         siginfo.si_code = SEGV_ACCERR;
775                         fault = FLTBOUNDS;
776                         break;
777                 case FT_SPEC_LD:
778                 case FT_ILL_ALT:
779                         siginfo.si_signo = SIGILL;
780                         siginfo.si_code = ILL_ILLADR;
781                         fault = FLTILL;
782                         break;
783                 default:
784                         siginfo.si_signo = SIGSEGV;
785                         siginfo.si_code = SEGV_MAPERR;
786                         fault = FLTBOUNDS;
787                         break;
788                 }
789                 break;

791         case T_SYS_RTT_ALIGN + T_USER:  /* user alignment error */
792         case T_ALIGNMENT + T_USER:      /* user alignment error */
793                 if (tudebug)
794                         showregs(type, rp, addr, 0);
795                 /*
796                  * If the user has to do unaligned references
797                  * the ugly stuff gets done here.
798                  */
799                 alignfaults++;
800                 if (&vis1_partial_support != NULL) {
801                         bzero(&siginfo, sizeof (siginfo));
802                         if (vis1_partial_support(rp,
803                             &siginfo, &fault) == 0)
804                                 goto out;
805                 }

807                 bzero(&siginfo, sizeof (siginfo));
808                 if (type == T_SYS_RTT_ALIGN + T_USER) {
809                         if (nfload(rp, NULL))
810                                 goto out;
811                         /*
812                          * Can't do unaligned stack access
813                          */
814                         siginfo.si_signo = SIGBUS;
815                         siginfo.si_code = BUS_ADRALN;
816                         siginfo.si_addr = addr;
817                         fault = FLTACCESS;
818                         break;
819                 }

821                 /*
822                  * Try to fix alignment before non-faulting load test.
823                  */
824                 if (p->p_fixalignment) {
825                         if (do_unaligned(rp, &badaddr) == SIMU_SUCCESS) {
826                                 rp->r_pc = rp->r_npc;
827                                 rp->r_npc += 4;
828                                 goto out;
829                         }
830                         if (nfload(rp, NULL))
831                                 goto out;
832                         siginfo.si_signo = SIGSEGV;
```



```
833                         siginfo.si_code = SEGV_MAPERR;
834                         siginfo.si_addr = badaddr;
835                         fault = FLTBOUNDS;
836                 } else {
837                         if (nfload(rp, NULL))
838                                 goto out;
839                         siginfo.si_signo = SIGBUS;
840                         siginfo.si_code = BUS_ADRALN;
841                         if (rp->r_pc & 3) {     /* offending address, if pc */
842                                 siginfo.si_addr = (caddr_t)rp->r_pc;
843                         } else {
844                                 if (calc_memaddr(rp, &badaddr) == SIMU_UNALIGN)
845                                         siginfo.si_addr = badaddr;
846                                 else
847                                         siginfo.si_addr = (caddr_t)rp->r_pc;
848                         }
849                         fault = FLTACCESS;
850                 }
851                 break;

853         case T_PRIV_INSTR + T_USER:     /* privileged instruction fault */
854                 if (tudebug)
855                         showregs(type, rp, (caddr_t)0, 0);

857                 bzero(&siginfo, sizeof (siginfo));
858 #ifdef  sun4v
859                 /*
860                  * If this instruction fault is a non-privileged %tick
861                  * or %stick trap, and %tick/%stick user emulation is
862                  * enabled as a result of an OS suspend, then simulate
863                  * the register read. We rely on simulate_rdtick to fail
864                  * if the instruction is not a %tick or %stick read,
865                  * causing us to fall through to the normal privileged
866                  * instruction handling.
867                  */
868                 if (tick_stick_emulation_active &&
869                     (X_FAULT_TYPE(mmu_fsr) == FT_NEW_PRVACT) &&
870                     simulate_rdtick(rp) == SIMU_SUCCESS) {
871                         /* skip the successfully simulated instruction */
872                         rp->r_pc = rp->r_npc;
873                         rp->r_npc += 4;
874                         goto out;
875                 }
876 #endif
877                 siginfo.si_signo = SIGILL;
878                 siginfo.si_code = ILL_PRVOPC;
879                 siginfo.si_addr = (caddr_t)rp->r_pc;
880                 fault = FLTILL;
881                 break;

883         case T_UNIMP_INSTR:             /* priv illegal instruction fault */
884                 if (fpras_implemented) {
885                         /*
886                          * Call fpras_chktrap indicating that
887                          * we've come from a trap handler and pass
888                          * the regs.  That function may choose to panic
889                          * (in which case it won't return) or it may
890                          * determine that a reboot is desired.  In the
891                          * latter case it must alter pc/npc to skip
892                          * the illegal instruction and continue at
893                          * a controlled address.
894                          */
895                         if (&fpras_chktrap) {
896                                 if (fpras_chktrap(rp))
897                                         goto cleanup;
898                         }
```

```
899                 }
900 #if defined(SF_ERRATA_23) || defined(SF_ERRATA_30) /* call ... illegal-insn */
901                 instr = *(int *)rp->r_pc;
902                 if ((instr & 0xc0000000) == 0x40000000) {
903                         long pc;

905                         rp->r_o7 = (long long)rp->r_pc;
906                         pc = rp->r_pc + ((instr & 0x3fffffff) << 2);
907                         rp->r_pc = rp->r_npc;
908                         rp->r_npc = pc;
909                         ill_calls++;
910                         goto cleanup;
911                 }
912 #endif /* SF_ERRATA_23 || SF_ERRATA_30 */
913                 /*
914                  * It's not an fpras failure and it's not SF_ERRATA_23 - die
915                  */
916                 addr = (caddr_t)rp->r_pc;
917                 (void) die(type, rp, addr, 0);
918                 /*NOTREACHED*/

920         case T_UNIMP_INSTR + T_USER:    /* illegal instruction fault */
921 #if defined(SF_ERRATA_23) || defined(SF_ERRATA_30) /* call ... illegal-insn */
922                 instr = fetch_user_instr((caddr_t)rp->r_pc);
923                 if ((instr & 0xc0000000) == 0x40000000) {
924                         long pc;

926                         rp->r_o7 = (long long)rp->r_pc;
927                         pc = rp->r_pc + ((instr & 0x3fffffff) << 2);
928                         rp->r_pc = rp->r_npc;
929                         rp->r_npc = pc;
930                         ill_calls++;
931                         goto out;
932                 }
933 #endif /* SF_ERRATA_23 || SF_ERRATA_30 */
934                 if (tudebug)
935                         showregs(type, rp, (caddr_t)0, 0);
936                 bzero(&siginfo, sizeof (siginfo));
937                 /*
938                  * Try to simulate the instruction.
939                  */
940                 switch (simulate_unimp(rp, &badaddr)) {
941                 case SIMU_RETRY:
942                         goto out;       /* regs are already set up */
943                         /*NOTREACHED*/

945                 case SIMU_SUCCESS:
946                         /* skip the successfully simulated instruction */
947                         rp->r_pc = rp->r_npc;
948                         rp->r_npc += 4;
949                         goto out;
950                         /*NOTREACHED*/

952                 case SIMU_FAULT:
953                         siginfo.si_signo = SIGSEGV;
954                         siginfo.si_code = SEGV_MAPERR;
955                         siginfo.si_addr = badaddr;
956                         fault = FLTBOUNDS;
957                         break;

959                 case SIMU_DZERO:
960                         siginfo.si_signo = SIGFPE;
961                         siginfo.si_code = FPE_INTDIV;
962                         siginfo.si_addr = (caddr_t)rp->r_pc;
963                         fault = FLTIZDIV;
964                         break;
```



```
966                 case SIMU_UNALIGN:
967                         siginfo.si_signo = SIGBUS;
968                         siginfo.si_code = BUS_ADRALN;
969                         siginfo.si_addr = badaddr;
970                         fault = FLTACCESS;
971                         break;

973                 case SIMU_ILLEGAL:
974                 default:
975                         siginfo.si_signo = SIGILL;
976                         op3 = (instr >> 19) & 0x3F;
977                         if ((IS_FLOAT(instr) && (op3 == IOP_V8_STQFA) ||
978                             (op3 == IOP_V8_STDFA)))
979                                 siginfo.si_code = ILL_ILLADR;
980                         else
981                                 siginfo.si_code = ILL_ILLOPC;
982                         siginfo.si_addr = (caddr_t)rp->r_pc;
983                         fault = FLTILL;
984                         break;
985                 }
986                 break;

988         case T_UNIMP_LDD + T_USER:
989         case T_UNIMP_STD + T_USER:
990                 if (tudebug)
991                         showregs(type, rp, (caddr_t)0, 0);
992                 switch (simulate_lddstd(rp, &badaddr)) {
993                 case SIMU_SUCCESS:
994                         /* skip the successfully simulated instruction */
995                         rp->r_pc = rp->r_npc;
996                         rp->r_npc += 4;
997                         goto out;
998                         /*NOTREACHED*/

1000                case SIMU_FAULT:
1001                        if (nfload(rp, NULL))
1002                                goto out;
1003                        siginfo.si_signo = SIGSEGV;
1004                        siginfo.si_code = SEGV_MAPERR;
1005                        siginfo.si_addr = badaddr;
1006                        fault = FLTBOUNDS;
1007                        break;

1009                case SIMU_UNALIGN:
1010                        if (nfload(rp, NULL))
1011                                goto out;
1012                        siginfo.si_signo = SIGBUS;
1013                        siginfo.si_code = BUS_ADRALN;
1014                        siginfo.si_addr = badaddr;
1015                        fault = FLTACCESS;
1016                        break;

1018                case SIMU_ILLEGAL;
1019                default:
1020                        siginfo.si_signo = SIGILL;
1021                        siginfo.si_code = ILL_ILLOPC;
1022                        siginfo.si_addr = (caddr_t)rp->r_pc;
1023                        fault = FLTILL;
1024                        break;
1025                }
1026                break;

1028        case T_UNIMP_LDD:
1029        case T_UNIMP_STD:
1030                if (simulate_lddstd(rp, &badaddr) == SIMU_SUCCESS) {
```

```
1031                        /* skip the successfully simulated instruction */
1032                        rp->r_pc = rp->r_npc;
1033                        rp->r_npc += 4;
1034                        goto cleanup;
1035                        /*NOTREACHED*/
1036                }
1037                /*
1038                 * A third party driver executed an {LDD,STD,LDDA,STDA}
1039                 * that we couldn't simulate.
1040                 */
1041                if (nfload(rp, NULL))
1042                        goto cleanup;

1044                if (curthread->t_lofault) {
1045                        if (lodebug) {
1046                                showregs(type, rp, addr, 0);
1047                                traceback((caddr_t)rp->r_sp);
1048                        }
1049                        rp->r_g1 = EFAULT;
1050                        rp->r_pc = curthread->t_lofault;
1051                        rp->r_npc = rp->r_pc + 4;
1052                        goto cleanup;
1053                }
1054                (void) die(type, rp, addr, 0);
1055                /*NOTREACHED*/

1057        case T_IDIV0 + T_USER:          /* integer divide by zero */
1058        case T_DIV0 + T_USER:           /* integer divide by zero */
1059                if (tudebug && tudebugfpe)
1060                        showregs(type, rp, (caddr_t)0, 0);
1061                bzero(&siginfo, sizeof (siginfo));
1062                siginfo.si_signo = SIGFPE;
1063                siginfo.si_code = FPE_INTDIV;
1064                siginfo.si_addr = (caddr_t)rp->r_pc;
1065                fault = FLTIZDIV;
1066                break;

1068        case T_INT_OVERFLOW + T_USER:   /* integer overflow */
1069                if (tudebug && tudebugfpe)
1070                        showregs(type, rp, (caddr_t)0, 0);
1071                bzero(&siginfo, sizeof (siginfo));
1072                siginfo.si_signo = SIGFPE;
1073                siginfo.si_code  = FPE_INTOVF;
1074                siginfo.si_addr  = (caddr_t)rp->r_pc;
1075                fault = FLTIOVF;
1076                break;

1078        case T_BREAKPOINT + T_USER:     /* breakpoint trap (t 1) */
1079                if (tudebug && tudebugbpt)
1080                        showregs(type, rp, (caddr_t)0, 0);
1081                bzero(&siginfo, sizeof (siginfo));
1082                siginfo.si_signo = SIGTRAP;
1083                siginfo.si_code = TRAP_BRKPT;
1084                siginfo.si_addr = (caddr_t)rp->r_pc;
1085                fault = FLTBPT;
1086                break;

1088        case T_TAG_OVERFLOW + T_USER:   /* tag overflow (taddcctv, tsubcctv) */
1089                if (tudebug)
1090                        showregs(type, rp, (caddr_t)0, 0);
1091                bzero(&siginfo, sizeof (siginfo));
1092                siginfo.si_signo = SIGEMT;
1093                siginfo.si_code = EMT_TAGOVF;
1094                siginfo.si_addr = (caddr_t)rp->r_pc;
1095                fault = FLTACCESS;
1096                break;
```



```
1098        case T_FLUSH_PCB + T_USER:      /* finish user window overflow */
1099        case T_FLUSHW + T_USER:         /* finish user window flush */
1100                /*
1101                 * This trap is entered from sys_rtt in locore.s when,
1102                 * upon return to user is is found that there are user
1103                 * windows in pcb_wbuf.  This happens because they could
1104                 * not be saved on the user stack, either because it
1105                 * wasn't resident or because it was misaligned.
1106                 */
1107        {
1108                int error;
1109                caddr_t sp;

1111                error = flush_user_windows_to_stack(&sp);
1112                /*
1113                 * Possible errors:
1114                 *      error copying out
1115                 *      unaligned stack pointer
1116                 * The first is given to us as the return value
1117                 * from flush_user_windows_to_stack().  The second
1118                 * results in residual windows in the pcb.
1119                 */
1120                if (error != 0) {
1121                        /*
1122                         * EINTR comes from a signal during copyout;
1123                         * we should not post another signal.
1124                         */
1125                        if (error != EINTR) {
1126                                /*
1127                                 * Zap the process with a SIGSEGV - process
1128                                 * may be managing its own stack growth by
1129                                 * taking SIGSEGVs on a different signal stack.
1130                                 */
1131                                bzero(&siginfo, sizeof (siginfo));
1132                                siginfo.si_signo = SIGSEGV;
1133                                siginfo.si_code  = SEGV_MAPERR;
1134                                siginfo.si_addr  = sp;
1135                                fault = FLTBOUNDS;
1136                        }
1137                        break;
1138                } else if (mpcb->mpcb_wbcnt) {
1139                        bzero(&siginfo, sizeof (siginfo));
1140                        siginfo.si_signo = SIGILL;
1141                        siginfo.si_code  = ILL_BADSTK;
1142                        siginfo.si_addr  = (caddr_t)rp->r_pc;
1143                        fault = FLTILL;
1144                        break;
1145                }
1146        }

1148                /*
1149                 * T_FLUSHW is used when handling a ta 0x3 -- the old flush
1150                 * window trap -- which is implemented by executing the
1151                 * flushw instruction. The flushw can trap if any of the
1152                 * stack pages are not writable for whatever reason. In this
1153                 * case only, we advance the pc to the next instruction so
1154                 * that the user thread doesn't needlessly execute the trap
1155                 * again. Normally this wouldn't be a problem -- we'll
1156                 * usually only end up here if this is the first touch to a
1157                 * stack page -- since the second execution won't trap, but
1158                 * if there's a watchpoint on the stack page the user thread
1159                 * would spin, continuously executing the trap instruction.
1160                 */
1161                if (type == T_FLUSHW + T_USER) {
1162                        rp->r_pc = rp->r_npc;
```

```
1163                        rp->r_npc += 4;
1164                }
1165                goto out;

1167        case T_AST + T_USER:            /* profiling or resched pseudo trap */
1168                if (lwp->lwp_pcb.pcb_flags & CPC_OVERFLOW) {
1169                        lwp->lwp_pcb.pcb_flags &= ~CPC_OVERFLOW;
1170                        if (kcpc_overflow_ast()) {
1171                                /*
1172                                 * Signal performance counter overflow
1173                                 */
1174                                if (tudebug)
1175                                        showregs(type, rp, (caddr_t)0, 0);
1176                                bzero(&siginfo, sizeof (siginfo));
1177                                siginfo.si_signo = SIGEMT;
1178                                siginfo.si_code = EMT_CPCOVF;
1179                                siginfo.si_addr = (caddr_t)rp->r_pc;
1180                                /* for trap_cleanup(), below */
1181                                oldpc = rp->r_pc - 4;
1182                                fault = FLTCPCOVF;
1183                        }
1184                }

1186                /*
1187                 * The CPC_OVERFLOW check above may already have populated
1188                 * siginfo and set fault, so the checks below must not
1189                 * touch these and the functions they call must use
1190                 * trapsig() directly.
1191                 */

1193                if (lwp->lwp_pcb.pcb_flags & ASYNC_HWERR) {
1194                        lwp->lwp_pcb.pcb_flags &= ~ASYNC_HWERR;
1195                        trap_async_hwerr();
1196                }

1198                if (lwp->lwp_pcb.pcb_flags & ASYNC_BERR) {
1199                        lwp->lwp_pcb.pcb_flags &= ~ASYNC_BERR;
1200                        trap_async_berr_bto(ASYNC_BERR, rp);
1201                }

1203                if (lwp->lwp_pcb.pcb_flags & ASYNC_BTO) {
1204                        lwp->lwp_pcb.pcb_flags &= ~ASYNC_BTO;
1205                        trap_async_berr_bto(ASYNC_BTO, rp);
1206                }

1208                break;
1209        }

1211        if (fault) {
1212                /* We took a fault so abort single step. */
1213                lwp->lwp_pcb.pcb_flags &= ~(NORMAL_STEP|WATCH_STEP);
1214        }
1215        trap_cleanup(rp, fault, &siginfo, oldpc == rp->r_pc);

1217 out:   /* We can't get here from a system trap */
1218        ASSERT(type & T_USER);
1219        trap_rtt();
1220        (void) new_mstate(curthread, mstate);
1221        /* Kernel probe */
1222        TNF_PROBE_1(thread_state, "thread", /* CSTYLED */,
1223                tnf_microstate, state, LMS_USER);

1225        TRACE_0(TR_FAC_TRAP, TR_C_TRAP_HANDLER_EXIT, "C_trap_handler_exit");
1226        return;

1228 cleanup:       /* system traps end up here */
```



```
1229        ASSERT(!(type & T_USER));

1231        TRACE_0(TR_FAC_TRAP, TR_C_TRAP_HANDLER_EXIT, "C_trap_handler_exit");
1232 }
_____unchanged_portion_omitted_

1343 /*
1344  * Called from fp_traps when a floating point trap occurs.
1345  * Note that the T_DATA_EXCEPTION case does not use X_FAULT_TYPE(mmu_fsr),
1346  * because mmu_fsr (now changed to code) is always 0.
1347  * Note that the T_UNIMP_INSTR case does not call simulate_unimp(),
1348  * because the simulator only simulates multiply and divide instructions,
1349  * which would not cause floating point traps in the first place.
1350  * XXX - Supervisor mode floating point traps?
1351  */
1352 void
1353 fpu_trap(struct regs *rp, caddr_t addr, uint32_t type, uint32_t code)
1354 {
1355        proc_t *p = ttoproc(curthread);
1356        klwp_id_t lwp = ttolwp(curthread);
1357        k_siginfo_t siginfo;
1358        uint_t op3, fault = 0;
1359        int mstate;
1360        char *badaddr;
1361        kfpu_t *fp;
1362        struct fpq *pfpq;
1363        uint32_t inst;
1364        utrap_handler_t *utrapp;

1366        CPU_STATS_ADDQ(CPU, sys, trap, 1);

1375        ASSERT(curthread->t_schedflag & TS_DONT_SWAP);

1368        if (USERMODE(rp->r_tstate)) {
1369                /*
1370                 * Set lwp_state before trying to acquire any
1371                 * adaptive lock
1372                 */
1373                ASSERT(lwp != NULL);
1374                lwp->lwp_state = LWP_SYS;
1375                /*
1376                 * Set up the current cred to use during this trap. u_cred
1377                 * no longer exists.  t_cred is used instead.
1378                 * The current process credential applies to the thread for
1379                 * the entire trap.  If trapping from the kernel, this
1380                 * should already be set up.
1381                 */
1382                if (curthread->t_cred != p->p_cred) {
1383                        cred_t *oldcred = curthread->t_cred;
1384                        /*
1385                         * DTrace accesses t_cred in probe context.  t_cred
1386                         * must always be either NULL, or point to a valid,
1387                         * allocated cred structure.
1388                         */
1389                        curthread->t_cred = crgetcred();
1390                        crfree(oldcred);
1391                }
1392                ASSERT(lwp->lwp_regs == rp);
1393                mstate = new_mstate(curthread, LMS_TRAP);
1394                siginfo.si_signo = 0;
1395                type |= T_USER;
1396        }

1398        TRACE_1(TR_FAC_TRAP, TR_C_TRAP_HANDLER_ENTER,
1399            "C_fpu_trap_handler_enter:type %x", type);
```

```
1401         if (tudebug && tudebugfpe)
1402                 showregs(type, rp, addr, 0);

1404         bzero(&siginfo, sizeof (siginfo));
1405         siginfo.si_code = code;
1406         siginfo.si_addr = addr;

1408         switch (type) {

1410         case T_FP_EXCEPTION_IEEE + T_USER:      /* FPU arithmetic exception */
1411                 /*
1412                  * FPU arithmetic exception - fake up a fpq if we
1413                  *      came here directly from _fp_ieee_exception,
1414                  *      which is indicated by a zero fpu_qcnt.
1415                  */
1416                 fp = lwptofpu(curthread->t_lwp);
1417                 utrapp = curthread->t_procp->p_utraps;
1418                 if (fp->fpu_qcnt == 0) {
1419                         inst = fetch_user_instr((caddr_t)rp->r_pc);
1420                         lwp->lwp_state = LWP_SYS;
1421                         pfpq = &fp->fpu_q->FQu.fpq;
1422                         pfpq->fpq_addr = (uint32_t *)rp->r_pc;
1423                         pfpq->fpq_instr = inst;
1424                         fp->fpu_qcnt = 1;
1425                         fp->fpu_q_entrysize = sizeof (struct fpq);
1426 #ifdef SF_V9_TABLE_28
1427                         /*
1428                          * Spitfire and blackbird followed the SPARC V9 manual
1429                          * paragraph 3 of section 5.1.7.9 FSR_current_exception
1430                          * (cexc) for setting fsr.cexc bits on underflow and
1431                          * overflow traps when the fsr.tem.inexact bit is set,
1432                          * instead of following Table 28. Bugid 1263234.
1433                          */
1434                         {
1435                                 extern int spitfire_bb_fsr_bug;

1437                                 if (spitfire_bb_fsr_bug &&
1438                                     (fp->fpu_fsr & FSR_TEM_NX)) {
1439                                         if (((fp->fpu_fsr & FSR_TEM_OF) == 0) &&
1440                                             (fp->fpu_fsr & FSR_CEXC_OF)) {
1441                                                 fp->fpu_fsr &= ~FSR_CEXC_OF;
1442                                                 fp->fpu_fsr |= FSR_CEXC_NX;
1443                                                 _fp_write_pfsr(&fp->fpu_fsr);
1444                                                 siginfo.si_code = FPE_FLTRES;
1445                                         }
1446                                         if (((fp->fpu_fsr & FSR_TEM_UF) == 0) &&
1447                                             (fp->fpu_fsr & FSR_CEXC_UF)) {
1448                                                 fp->fpu_fsr &= ~FSR_CEXC_UF;
1449                                                 fp->fpu_fsr |= FSR_CEXC_NX;
1450                                                 _fp_write_pfsr(&fp->fpu_fsr);
1451                                                 siginfo.si_code = FPE_FLTRES;
1452                                         }
1453                                 }
1454                         }
1455 #endif /* SF_V9_TABLE_28 */
1456                         rp->r_pc = rp->r_npc;
1457                         rp->r_npc += 4;
1458                 } else if (utrapp && utrapp[UT_FP_EXCEPTION_IEEE_754]) {
1459                         /*
1460                          * The user had a trap handler installed.  Jump to
1461                          * the trap handler instead of signalling the process.
1462                          */
1463                         rp->r_pc = (long)utrapp[UT_FP_EXCEPTION_IEEE_754];
1464                         rp->r_npc = rp->r_pc + 4;
1465                         break;
1466                 }
```



```
1467                 siginfo.si_signo = SIGFPE;
1468                 fault = FLTFPE;
1469                 break;

1471         case T_DATA_EXCEPTION + T_USER:         /* user data access exception */
1472                 siginfo.si_signo = SIGSEGV;
1473                 fault = FLTBOUNDS;
1474                 break;

1476         case T_LDDF_ALIGN + T_USER: /* 64 bit user lddfa alignment error */
1477         case T_STDF_ALIGN + T_USER: /* 64 bit user stdfa alignment error */
1478                 alignfaults++;
1479                 lwp->lwp_state = LWP_SYS;
1480                 if (&vis1_partial_support != NULL) {
1481                         bzero(&siginfo, sizeof (siginfo));
1482                         if (vis1_partial_support(rp,
1483                             &siginfo, &fault) == 0)
1484                                 goto out;
1485                 }
1486                 if (do_unaligned(rp, &badaddr) == SIMU_SUCCESS) {
1487                         rp->r_pc = rp->r_npc;
1488                         rp->r_npc += 4;
1489                         goto out;
1490                 }
1491                 fp = lwptofpu(curthread->t_lwp);
1492                 fp->fpu_qcnt = 0;
1493                 siginfo.si_signo = SIGSEGV;
1494                 siginfo.si_code = SEGV_MAPERR;
1495                 siginfo.si_addr = badaddr;
1496                 fault = FLTBOUNDS;
1497                 break;

1499         case T_ALIGNMENT + T_USER:              /* user alignment error */
1500                 /*
1501                  * If the user has to do unaligned references
1502                  * the ugly stuff gets done here.
1503                  * Only handles vanilla loads and stores.
1504                  */
1505                 alignfaults++;
1506                 if (p->p_fixalignment) {
1507                         if (do_unaligned(rp, &badaddr) == SIMU_SUCCESS) {
1508                                 rp->r_pc = rp->r_npc;
1509                                 rp->r_npc += 4;
1510                                 goto out;
1511                         }
1512                         siginfo.si_signo = SIGSEGV;
1513                         siginfo.si_code = SEGV_MAPERR;
1514                         siginfo.si_addr = badaddr;
1515                         fault = FLTBOUNDS;
1516                 } else {
1517                         siginfo.si_signo = SIGBUS;
1518                         siginfo.si_code = BUS_ADRALN;
1519                         if (rp->r_pc & 3) {     /* offending address, if pc */
1520                                 siginfo.si_addr = (caddr_t)rp->r_pc;
1521                         } else {
1522                                 if (calc_memaddr(rp, &badaddr) == SIMU_UNALIGN)
1523                                         siginfo.si_addr = badaddr;
1524                                 else
1525                                         siginfo.si_addr = (caddr_t)rp->r_pc;
1526                         }
1527                         fault = FLTACCESS;
1528                 }
1529                 break;

1531         case T_UNIMP_INSTR + T_USER:            /* illegal instruction fault */
1532                 siginfo.si_signo = SIGILL;
```

```
1533                 inst = fetch_user_instr((caddr_t)rp->r_pc);
1534                 op3 = (inst >> 19) & 0x3F;
1535                 if ((op3 == IOP_V8_STQFA) || (op3 == IOP_V8_STDFA))
1536                         siginfo.si_code = ILL_ILLADR;
1537                 else
1538                         siginfo.si_code = ILL_ILLTRP;
1539                 fault = FLTILL;
1540                 break;

1542         default:
1543                 (void) die(type, rp, addr, 0);
1544                 /*NOTREACHED*/
1545         }

1547         /*
1548          * We can't get here from a system trap
1549          * Never restart any instruction which got here from an fp trap.
1550          */
1551         ASSERT(type & T_USER);

1553         trap_cleanup(rp, fault, &siginfo, 0);
1554 out:
1555         trap_rtt();
1556         (void) new_mstate(curthread, mstate);
1557 }
```

_____***unchanged_portion_omitted_***