```
**********************************************************
   68010 Sat Nov 15 21:14:32 2014
new/usr/src/uts/common/disp/disp.c
patch setfrontbackdq
**********************************************************
______unchanged_portion_omitted_

1151 /*
1152  * setbackdq() keeps runqs balanced such that the difference in length
1153  * between the chosen runq and the next one is no more than RUNQ_MAX_DIFF.
1154  * For threads with priorities below RUNQ_MATCH_PRI levels, the runq's lengths
1155  * must match.  When per-thread TS_RUNQMATCH flag is set, setbackdq() will
1156  * try to keep runqs perfectly balanced regardless of the thread priority.
1157  */
1158 #define RUNQ_MATCH_PRI  16      /* pri below which queue lengths must match */
1159 #define RUNQ_MAX_DIFF   2       /* maximum runq length difference */
1160 #define RUNQ_LEN(cp, pri)       ((cp)->cpu_disp->disp_q[pri].dq_sruncnt)

1162 /*
1163  * Macro that evaluates to true if it is likely that the thread has cache
1164  * warmth. This is based on the amount of time that has elapsed since the
1165  * thread last ran. If that amount of time is less than "rechoose_interval"
1166  * ticks, then we decide that the thread has enough cache warmth to warrant
1167  * some affinity for t->t_cpu.
1168  */
1169 #define THREAD_HAS_CACHE_WARMTH(thread) \
1170         ((thread == curthread) ||       \
1171         ((ddi_get_lbolt() - thread->t_disp_time) <= rechoose_interval))

1173 #endif /* ! codereview */
1174 /*
1175  * Put the specified thread on the front/back of the dispatcher queue
1176  * corresponding to its current priority.
1172  * Put the specified thread on the back of the dispatcher
1173  * queue corresponding to its current priority.
1177  *
1178  * Called with the thread in transition, onproc or stopped state and locked
1179  * (transition implies locked) and at high spl.  Returns with the thread in
1180  * TS_RUN state and still locked.
1175  * Called with the thread in transition, onproc or stopped state
1176  * and locked (transition implies locked) and at high spl.
1177  * Returns with the thread in TS_RUN state and still locked.
1181  */
1182 static void
1183 setfrontbackdq(kthread_t *tp, boolean_t front)
1179 void
1180 setbackdq(kthread_t *tp)
1184 {
1185         dispq_t         *dq;
1186         disp_t          *dp;
1187         cpu_t           *cp;
1188         pri_t           tpri;
1189         boolean_t       bound;
1186         int             bound;
1190         boolean_t       self;

1192         ASSERT(THREAD_LOCK_HELD(tp));
1193         ASSERT((tp->t_schedflag & TS_ALLSTART) == 0);
1194         ASSERT(!thread_on_queue(tp));   /* make sure tp isn't on a runq */

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



```
1202                 return;
1203         }

1205         self  = (tp == curthread);
1206         bound = (tp->t_bound_cpu || tp->t_weakbound_cpu);

1204         if (tp->t_bound_cpu || tp->t_weakbound_cpu)
1205                 bound = 1;
1206         else
1207                 bound = 0;

1208         tpri = DISP_PRIO(tp);
1209         if (ncpus == 1)
1210                 cp = tp->t_cpu;
1211         else if (!bound) {
1212                 if (tpri >= kpqpri) {
1213                         setkpdq(tp, front ? SETKP_FRONT : SETKP_BACK);
1214                         setkpdq(tp, SETKP_BACK);
1214                         return;
1215                 }

1217                 cp = tp->t_cpu;

1219                 if (!front) {
1220 #endif /* ! codereview */
1221                         /*
1222                          * We'll generally let this thread continue to run where
1223                          * it last ran...but will consider migration if:
1224                          * - We thread probably doesn't have much cache warmth.
1225                          * - The CPU where it last ran is the target of an offli
1226                          *   request.
1227                          * - The thread last ran outside it's home lgroup.
1228                          */
1229                         if ((!THREAD_HAS_CACHE_WARMTH(tp)) || (cp == cpu_inmotio
1230                                 cp = disp_lowpri_cpu(cp, tp->t_lpl, tpri, NULL);
1231                         } else if (!LGRP_CONTAINS_CPU(tp->t_lpl->lpl_lgrp, cp))
1232                                 cp = disp_lowpri_cpu(cp, tp->t_lpl, tpri,
1233                                     self ? cp : NULL);
1234                         }

1218                 if ((!THREAD_HAS_CACHE_WARMTH(tp)) ||
1219                     (tp->t_cpu == cpu_inmotion)) {
1220                         cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri, NULL);
1221                 } else if (!LGRP_CONTAINS_CPU(tp->t_lpl->lpl_lgrp, tp->t_cpu)) {
1222                         cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri,
1223                             self ? tp->t_cpu : NULL);
1224                 } else {
1225                         cp = tp->t_cpu;
1236                 }

1238                 if (tp->t_cpupart == cp->cpu_part) {
1239                         if (front) {
1240                                 /*
1241                                  * We'll generally let this thread continue to r
1242                                  * where it last ran, but will consider migratio
1243                                  * - The thread last ran outside it's home lgrou
1244                                  * - The CPU where it last ran is the target of
1245                                  *   offline request (a thread_nomigrate() on th
1246                                  *   motion CPU relies on this when forcing a pr
1247                                  * - The thread isn't the highest priority threa
1248                                  *   it last ran, and it is considered not likel
1249                                  *   have significant cache warmth.
1250                                  */
1251                                 if ((!LGRP_CONTAINS_CPU(tp->t_lpl->lpl_lgrp, cp)
1252                                     (cp == cpu_inmotion)) {
1253                                         cp = disp_lowpri_cpu(cp, tp->t_lpl, tpri
```

```
1254                                        self ? cp : NULL);
1255                                } else if ((tpri < cp->cpu_disp->disp_maxrunpri)
1256                                    (!THREAD_HAS_CACHE_WARMTH(tp))) {
1257                                        cp = disp_lowpri_cpu(cp, tp->t_lpl, tpri
1258                                            NULL);
1259                                }
1260                        } else {
1261 #endif /* ! codereview */
1262                                int     qlen;

1264                                /*
1265                                 * Perform any CMT load balancing
1266                                 */
1267                                cp = cmt_balance(tp, cp);

1269                                /*
1270                                 * Balance across the run queues
1271                                 */
1272                                qlen = RUNQ_LEN(cp, tpri);
1273                                if (tpri >= RUNQ_MATCH_PRI &&
1274                                    !(tp->t_schedflag & TS_RUNQMATCH))
1275                                        qlen -= RUNQ_MAX_DIFF;
1276                                if (qlen > 0) {
1277                                        cpu_t *newcp;

1279                                        if (tp->t_lpl->lpl_lgrpid == LGRP_ROOTID
1280                                                newcp = cp->cpu_next_part;
1281                                        } else if ((newcp = cp->cpu_next_lpl) ==
1282                                                newcp = cp->cpu_next_part;
1283                                        }

1285                                        if (RUNQ_LEN(newcp, tpri) < qlen) {
1286                                                DTRACE_PROBE3(runq__balance,
1287                                                    kthread_t *, tp,
1288                                                    cpu_t *, cp, cpu_t *, newcp)
1289                                                cp = newcp;
1290                                        }
1291                                }
1292                        }
1293 #endif /* ! codereview */
1294                } else {
1295                        /*
1296                         * Migrate to a cpu in the new partition.
1297                         */
1298                        cp = disp_lowpri_cpu(tp->t_cpupart->cp_cpulist,
1299                            tp->t_lpl, tp->t_pri, NULL);
1300                }

1302 #endif /* ! codereview */
1303                ASSERT((cp->cpu_flags & CPU_QUIESCED) == 0);
1304        } else {
1305                /*
1306                 * It is possible that t_weakbound_cpu != t_bound_cpu (for
1307                 * a short time until weak binding that existed when the
1308                 * strong binding was established has dropped) so we must
1309                 * favour weak binding over strong.
1310                 */
1311                cp = tp->t_weakbound_cpu ?
1312                    tp->t_weakbound_cpu : tp->t_bound_cpu;
1313        }

1315 #endif /* ! codereview */
1316        /*
1317         * A thread that is ONPROC may be temporarily placed on the run queue
1318         * but then chosen to run again by disp.  If the thread we're placing on
1319         * the queue is in TS_ONPROC state, don't set its t_waitrq until a
```



```
1320         * replacement process is actually scheduled in swtch().  In this
1321         * situation, curthread is the only thread that could be in the ONPROC
1322         * state.
1323         */
1324        if ((!self) && (tp->t_waitrq == 0)) {
1325                hrtime_t curtime;

1327                curtime = gethrtime_unscaled();
1328                (void) cpu_update_pct(tp, curtime);
1329                tp->t_waitrq = curtime;
1330        } else {
1331                (void) cpu_update_pct(tp, gethrtime_unscaled());
1332        }

1334        dp = cp->cpu_disp;
1335        disp_lock_enter_high(&dp->disp_lock);

1337        DTRACE_SCHED3(enqueue, kthread_t *, tp, disp_t *, dp, int, front);
1338        if (front) {
1339                TRACE_2(TR_FAC_DISP, TR_FRONTQ, "frontq:pri %d tid %p", tpri,
1340                    tp);
1341        } else {
1229        DTRACE_SCHED3(enqueue, kthread_t *, tp, disp_t *, dp, int, 0);
1342                TRACE_3(TR_FAC_DISP, TR_BACKQ, "setbackdq:pri %d cpu %p tid %p",
1343                    tpri, cp, tp);
1344        }
1345 #endif /* ! codereview */

1347 #ifndef NPROBE
1348        /* Kernel probe */
1349        if (tnf_tracing_active)
1350                tnf_thread_queue(tp, cp, tpri);
1351 #endif /* NPROBE */

1353        ASSERT(tpri >= 0 && tpri < dp->disp_npri);

1355        THREAD_RUN(tp, &dp->disp_lock);         /* set t_state to TS_RUN */
1356        tp->t_disp_queue = dp;
1357        tp->t_link = NULL;

1359        dq = &dp->disp_q[tpri];
1360        dp->disp_nrunnable++;
1361        if (!bound)
1362                dp->disp_steal = 0;
1363        membar_enter();

1365        if (dq->dq_sruncnt++ != 0) {
1366                if (front) {
1367                        ASSERT(dq->dq_last != NULL);
1368                        tp->t_link = dq->dq_first;
1369                        dq->dq_first = tp;
1370                } else {
1371 #endif /* ! codereview */
1372                        ASSERT(dq->dq_first != NULL);
1373                        dq->dq_last->t_link = tp;
1374                        dq->dq_last = tp;
1375                }
1376 #endif /* ! codereview */
1377        } else {
1378                ASSERT(dq->dq_first == NULL);
1379                ASSERT(dq->dq_last == NULL);
1380                dq->dq_first = dq->dq_last = tp;
1381                BT_SET(dp->disp_qactmap, tpri);
1382                if (tpri > dp->disp_maxrunpri) {
1383                        dp->disp_maxrunpri = tpri;
1384                        membar_enter();
```

```
1385                        cpu_resched(cp, tpri);
1386                }
1387        }

1389        if (!bound && tpri > dp->disp_max_unbound_pri) {
1390                if (self && dp->disp_max_unbound_pri == -1 && cp == CPU) {
1391                        /*
1392                         * If there are no other unbound threads on the
1393                         * run queue, don't allow other CPUs to steal
1394                         * this thread while we are in the middle of a
1395                         * context switch. We may just switch to it
1396                         * again right away. CPU_DISP_DONTSTEAL is cleared
1397                         * in swtch and swtch_to.
1398                         */
1399                        cp->cpu_disp_flags |= CPU_DISP_DONTSTEAL;
1400                }
1401                dp->disp_max_unbound_pri = tpri;
1402        }

1404 #endif /* ! codereview */
1405        (*disp_enq_thread)(cp, bound);
1406 }

1408 /*
1409  * Put the specified thread on the back of the dispatcher
1410  * queue corresponding to its current priority.
1411  *
1412  * Called with the thread in transition, onproc or stopped state
1413  * and locked (transition implies locked) and at high spl.
1414  * Returns with the thread in TS_RUN state and still locked.
1415  */
1416 void
1417 setbackdq(kthread_t *tp)
1418 {
1419        setfrontbackdq(tp, B_FALSE);
1420 }

1422 /*
1423 #endif /* ! codereview */
1424  * Put the specified thread on the front of the dispatcher
1425  * queue corresponding to its current priority.
1426  *
1427  * Called with the thread in transition, onproc or stopped state
1428  * and locked (transition implies locked) and at high spl.
1429  * Returns with the thread in TS_RUN state and still locked.
1430  */
1431 void
1432 setfrontdq(kthread_t *tp)
1433 {
1434        setfrontbackdq(tp, B_TRUE);
1232        disp_t          *dp;
1233        dispq_t         *dq;
1234        cpu_t           *cp;
1235        pri_t           tpri;
1236        int             bound;

1238        ASSERT(THREAD_LOCK_HELD(tp));
1239        ASSERT((tp->t_schedflag & TS_ALLSTART) == 0);
1240        ASSERT(!thread_on_queue(tp));   /* make sure tp isn't on a runq */

1242        /*
1243         * If thread is "swapped" or on the swap queue don't
1244         * queue it, but wake sched.
1245         */
1246        if ((tp->t_schedflag & (TS_LOAD | TS_ON_SWAPQ)) != TS_LOAD) {
1247                disp_swapped_setrun(tp);
```



```
1248                return;
1249        }

1251        if (tp->t_bound_cpu || tp->t_weakbound_cpu)
1252                bound = 1;
1253        else
1254                bound = 0;

1256        tpri = DISP_PRIO(tp);
1257        if (ncpus == 1)
1258                cp = tp->t_cpu;
1259        else if (!bound) {
1260                if (tpri >= kpqpri) {
1261                        setkpdq(tp, SETKP_FRONT);
1262                        return;
1263                }
1264                cp = tp->t_cpu;
1265                if (tp->t_cpupart == cp->cpu_part) {
1266                        /*
1267                         * We'll generally let this thread continue to run
1268                         * where it last ran, but will consider migration if:
1269                         * - The thread last ran outside it's home lgroup.
1270                         * - The CPU where it last ran is the target of an
1271                         *   offline request (a thread_nomigrate() on the in
1272                         *   motion CPU relies on this when forcing a preempt).
1273                         * - The thread isn't the highest priority thread where
1274                         *   it last ran, and it is considered not likely to
1275                         *   have significant cache warmth.
1276                         */
1277                        if ((!LGRP_CONTAINS_CPU(tp->t_lpl->lpl_lgrp, cp)) ||
1278                            (cp == cpu_inmotion)) {
1279                                cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri,
1280                                    (tp == curthread) ? cp : NULL);
1281                        } else if ((tpri < cp->cpu_disp->disp_maxrunpri) &&
1282                            (!THREAD_HAS_CACHE_WARMTH(tp))) {
1283                                cp = disp_lowpri_cpu(tp->t_cpu, tp->t_lpl, tpri,
1284                                    NULL);
1285                        }
1286                } else {
1287                        /*
1288                         * Migrate to a cpu in the new partition.
1289                         */
1290                        cp = disp_lowpri_cpu(tp->t_cpupart->cp_cpulist,
1291                            tp->t_lpl, tp->t_pri, NULL);
1292                }
1293                ASSERT((cp->cpu_flags & CPU_QUIESCED) == 0);
1294        } else {
1295                /*
1296                 * It is possible that t_weakbound_cpu != t_bound_cpu (for
1297                 * a short time until weak binding that existed when the
1298                 * strong binding was established has dropped) so we must
1299                 * favour weak binding over strong.
1300                 */
1301                cp = tp->t_weakbound_cpu ?
1302                    tp->t_weakbound_cpu : tp->t_bound_cpu;
1303        }

1305        /*
1306         * A thread that is ONPROC may be temporarily placed on the run queue
1307         * but then chosen to run again by disp.  If the thread we're placing on
1308         * the queue is in TS_ONPROC state, don't set its t_waitrq until a
1309         * replacement process is actually scheduled in swtch().  In this
1310         * situation, curthread is the only thread that could be in the ONPROC
1311         * state.
1312         */
1313        if ((tp != curthread) && (tp->t_waitrq == 0)) {
```

```
1314                    hrtime_t curtime;

1316                    curtime = gethrtime_unscaled();
1317                    (void) cpu_update_pct(tp, curtime);
1318                    tp->t_waitrq = curtime;
1319            } else {
1320                    (void) cpu_update_pct(tp, gethrtime_unscaled());
1321            }

1323            dp = cp->cpu_disp;
1324            disp_lock_enter_high(&dp->disp_lock);

1326            TRACE_2(TR_FAC_DISP, TR_FRONTQ, "frontq:pri %d tid %p", tpri, tp);
1327            DTRACE_SCHED3(enqueue, kthread_t *, tp, disp_t *, dp, int, 1);

1329    #ifndef NPROBE
1330            /* Kernel probe */
1331            if (tnf_tracing_active)
1332                    tnf_thread_queue(tp, cp, tpri);
1333    #endif /* NPROBE */

1335            ASSERT(tpri >= 0 && tpri < dp->disp_npri);

1337            THREAD_RUN(tp, &dp->disp_lock);         /* set TS_RUN state and lock */
1338            tp->t_disp_queue = dp;

1340            dq = &dp->disp_q[tpri];
1341            dp->disp_nrunnable++;
1342            if (!bound)
1343                    dp->disp_steal = 0;
1344            membar_enter();

1346            if (dq->dq_sruncnt++ != 0) {
1347                    ASSERT(dq->dq_last != NULL);
1348                    tp->t_link = dq->dq_first;
1349                    dq->dq_first = tp;
1350            } else {
1351                    ASSERT(dq->dq_last == NULL);
1352                    ASSERT(dq->dq_first == NULL);
1353                    tp->t_link = NULL;
1354                    dq->dq_first = dq->dq_last = tp;
1355                    BT_SET(dp->disp_qactmap, tpri);
1356                    if (tpri > dp->disp_maxrunpri) {
1357                            dp->disp_maxrunpri = tpri;
1358                            membar_enter();
1359                            cpu_resched(cp, tpri);
1360                    }
1361            }

1363            if (!bound && tpri > dp->disp_max_unbound_pri) {
1364                    if (tp == curthread && dp->disp_max_unbound_pri == -1 &&
1365                        cp == CPU) {
1366                            /*
1367                             * If there are no other unbound threads on the
1368                             * run queue, don't allow other CPUs to steal
1369                             * this thread while we are in the middle of a
1370                             * context switch. We may just switch to it
1371                             * again right away. CPU_DISP_DONTSTEAL is cleared
1372                             * in swtch and swtch_to.
1373                             */
1374                            cp->cpu_disp_flags |= CPU_DISP_DONTSTEAL;
1375                    }
1376                    dp->disp_max_unbound_pri = tpri;
1377            }
1378            (*disp_enq_thread)(cp, bound);
1435    }
```

**_____unchanged_portion_omitted_**