```
**********************************************************
   34820 Tue Nov  4 16:25:24 2014
new/usr/src/uts/common/cpr/cpr_main.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
```

```
/*
 * 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 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * This module contains the guts of checkpoint-resume mechanism.
 * All code in this module is platform independent.
 */

#include <sys/types.h>
#include <sys/errno.h>
#include <sys/callb.h>
#include <sys/processor.h>
#include <sys/machsystm.h>
#include <sys/clock.h>
#include <sys/vfs.h>
#include <sys/kmem.h>
#include <nfs/lm.h>
#include <sys/systm.h>
#include <sys/cpr.h>
#include <sys/bootconf.h>
#include <sys/cyclic.h>
#include <sys/filio.h>
#include <sys/fs/ufs_filio.h>
#include <sys/epm.h>
#include <sys/modctl.h>
#include <sys/reboot.h>
#include <sys/kdi.h>
#include <sys/promif.h>
#include <sys/srn.h>
#include <sys/cpr_impl.h>

#define PPM(dip) ((dev_info_t *)DEVI(dip)->devi_pm_ppm)

extern struct cpr_terminator cpr_term;

extern int cpr_alloc_statefile(int);
extern void cpr_start_kernel_threads(void);
extern void cpr_abbreviate_devpath(char *, char *);
extern void cpr_convert_promtime(cpr_time_t *);
```



```
extern void cpr_send_notice(void);
extern void cpr_set_bitmap_size(void);
extern void cpr_stat_init();
extern void cpr_statef_close(void);
extern void flush_windows(void);
extern void (*srn_signal)(int, int);
extern void init_cpu_syscall(struct cpu *);
extern void i_cpr_pre_resume_cpus();
extern void i_cpr_post_resume_cpus();
extern int cpr_is_ufs(struct vfs *);

extern int pm_powering_down;
extern kmutex_t srn_clone_lock;
extern int srn_inuse;

static int cpr_suspend(int);
static int cpr_resume(int);
static void cpr_suspend_init(int);
#if defined(__x86)
static int cpr_suspend_cpus(void);
static void cpr_resume_cpus(void);
#endif
static int cpr_all_online(void);
static void cpr_restore_offline(void);

cpr_time_t wholecycle_tv;
int cpr_suspend_succeeded;
pfn_t curthreadpfn;
int curthreadremapped;

extern cpuset_t cpu_ready_set;
extern void *(*cpu_pause_func)(void *);

extern processorid_t i_cpr_bootcpuid(void);
extern cpu_t *i_cpr_bootcpu(void);
extern void tsc_adjust_delta(hrtime_t tdelta);
extern void tsc_resume(void);
extern int tsc_resume_in_cyclic;

/*
 * Set this variable to 1, to have device drivers resume in an
 * uniprocessor environment. This is to allow drivers that assume
 * that they resume on a UP machine to continue to work. Should be
 * deprecated once the broken drivers are fixed
 */
int cpr_resume_uniproc = 0;

/*
 * save or restore abort_enable;  this prevents a drop
 * to kadb or prom during cpr_resume_devices() when
 * there is no kbd present;  see abort_sequence_enter()
 */
static void
cpr_sae(int stash)
{
        static int saved_ae = -1;

        if (stash) {
                saved_ae = abort_enable;
                abort_enable = 0;
        } else if (saved_ae != -1) {
                abort_enable = saved_ae;
                saved_ae = -1;
        }
}
```

_____unchanged_portion_omitted_

```
384 int
385 cpr_suspend_cpus(void)
386 {
387         int     ret = 0;
388         extern void *i_cpr_save_context(void *arg);
389
390         mutex_enter(&cpu_lock);
391
392         /*
393          * the machine could not have booted without a bootcpu
394          */
395         ASSERT(i_cpr_bootcpu() != NULL);
396
397         /*
398          * bring all the offline cpus online
399          */
400         if ((ret = cpr_all_online())) {
401                 mutex_exit(&cpu_lock);
402                 return (ret);
403         }
404
405         /*
406          * Set the affinity to be the boot processor
407          * This is cleared in either cpr_resume_cpus() or cpr_unpause_cpus()
408          */
409         affinity_set(i_cpr_bootcpuid());
410
411         ASSERT(CPU->cpu_id == 0);
412
413         PMD(PMD_SX, ("curthread running on bootcpu\n"))
414
415         /*
416          * pause all other running CPUs and save the CPU state at the sametime
417          */
418         pause_cpus(NULL, i_cpr_save_context);
419         cpu_pause_func = i_cpr_save_context;
420         pause_cpus(NULL);
421
422         mutex_exit(&cpu_lock);
423
424         return (0);
425 }
_____unchanged_portion_omitted_

764 void
765 cpr_resume_cpus(void)
766 {
767         /*
768          * this is a cut down version of start_other_cpus()
769          * just do the initialization to wake the other cpus
770          */
771
772 #if defined(__x86)
773         /*
774          * Initialize our syscall handlers
775          */
776         init_cpu_syscall(CPU);
777
778 #endif
779
780         i_cpr_pre_resume_cpus();
781
782         /*
783          * Restart the paused cpus
784          */
```



```
785         mutex_enter(&cpu_lock);
786         start_cpus();
787         mutex_exit(&cpu_lock);
788
789         i_cpr_post_resume_cpus();
790
791         mutex_enter(&cpu_lock);
792         /*
793          * Restore this cpu to use the regular cpu_pause(), so that
794          * online and offline will work correctly
795          */
796         cpu_pause_func = NULL;
797
798         /*
799          * clear the affinity set in cpr_suspend_cpus()
800          */
801         affinity_clear();
802
803         /*
804          * offline all the cpus that were brought online during suspend
805          */
806         cpr_restore_offline();
807
808         mutex_exit(&cpu_lock);
809 }
810
811 void
812 cpr_unpause_cpus(void)
813 {
814         /*
815          * Now restore the system back to what it was before we suspended
816          */
817
818         PMD(PMD_SX, ("cpr_unpause_cpus: restoring system\n"))
819
820         mutex_enter(&cpu_lock);
821
822         /*
823          * Restore this cpu to use the regular cpu_pause(), so that
824          * online and offline will work correctly
825          */
826         cpu_pause_func = NULL;
827
828         /*
829          * Restart the paused cpus
830          */
831         start_cpus();
832
833         /*
834          * clear the affinity set in cpr_suspend_cpus()
835          */
836         affinity_clear();
837
838         /*
839          * offline all the cpus that were brought online during suspend
840          */
841         cpr_restore_offline();
842
843         mutex_exit(&cpu_lock);
844 }
845
846 /*
847  * Bring the system back up from a checkpoint, at this point
848  * the VM has been minimally restored by boot, the following
849  * are executed sequentially:
850  *
```

```
 838  *    - machdep setup and enable interrupts (mp startup if it's mp)
 839  *    - resume all devices
 840  *    - restart daemons
 841  *    - put all threads back on run queue
 842  */
 843 static int
 844 cpr_resume(int sleeptype)
 845 {
 846         cpr_time_t pwron_tv, *ctp;
 847         char *str;
 848         int rc = 0;

 850         /*
 851          * The following switch is used to resume the system
 852          * that was suspended to a different level.
 853          */
 854         CPR_DEBUG(CPR_DEBUG1, "\nEntering cpr_resume...\n");
 855         PMD(PMD_SX, ("cpr_resume %x\n", sleeptype))

 857         /*
 858          * Note:
 859          *
 860          * The rollback labels rb_xyz do not represent the cpr resume
 861          * state when event 'xyz' has happened. Instead they represent
 862          * the state during cpr suspend when event 'xyz' was being
 863          * entered (and where cpr suspend failed). The actual call that
 864          * failed may also need to be partially rolled back, since they
 865          * aren't atomic in most cases.  In other words, rb_xyz means
 866          * "roll back all cpr suspend events that happened before 'xyz',
 867          * and the one that caused the failure, if necessary."
 868          */
 869         switch (CPR->c_substate) {
 870 #if defined(__sparc)
 871         case C_ST_DUMP:
 872                 /*
 873                  * This is most likely a full-fledged cpr_resume after
 874                  * a complete and successful cpr suspend. Just roll back
 875                  * everything.
 876                  */
 877                 ASSERT(sleeptype == CPR_TODISK);
 878                 break;

 880         case C_ST_REUSABLE:
 881         case C_ST_DUMP_NOSPC:
 882         case C_ST_SETPROPS_0:
 883         case C_ST_SETPROPS_1:
 884                 /*
 885                  * C_ST_REUSABLE and C_ST_DUMP_NOSPC are the only two
 886                  * special switch cases here. The other two do not have
 887                  * any state change during cpr_suspend() that needs to
 888                  * be rolled back. But these are exit points from
 889                  * cpr_suspend, so theoretically (or in the future), it
 890                  * is possible that a need for roll back of a state
 891                  * change arises between these exit points.
 892                  */
 893                 ASSERT(sleeptype == CPR_TODISK);
 894                 goto rb_dump;
 895 #endif

 897         case C_ST_NODUMP:
 898                 PMD(PMD_SX, ("cpr_resume: NODUMP\n"))
 899                 goto rb_nodump;

 901         case C_ST_STOP_KERNEL_THREADS:
 902                 PMD(PMD_SX, ("cpr_resume: STOP_KERNEL_THREADS\n"))
 903                 goto rb_stop_kernel_threads;
```



```
 905         case C_ST_SUSPEND_DEVICES:
 906                 PMD(PMD_SX, ("cpr_resume: SUSPEND_DEVICES\n"))
 907                 goto rb_suspend_devices;

 909 #if defined(__sparc)
 910         case C_ST_STATEF_ALLOC:
 911                 ASSERT(sleeptype == CPR_TODISK);
 912                 goto rb_statef_alloc;

 914         case C_ST_DISABLE_UFS_LOGGING:
 915                 ASSERT(sleeptype == CPR_TODISK);
 916                 goto rb_disable_ufs_logging;
 917 #endif

 919         case C_ST_PM_REATTACH_NOINVOL:
 920                 PMD(PMD_SX, ("cpr_resume: REATTACH_NOINVOL\n"))
 921                 goto rb_pm_reattach_noinvol;

 923         case C_ST_STOP_USER_THREADS:
 924                 PMD(PMD_SX, ("cpr_resume: STOP_USER_THREADS\n"))
 925                 goto rb_stop_user_threads;

 927 #if defined(__sparc)
 928         case C_ST_MP_OFFLINE:
 929                 PMD(PMD_SX, ("cpr_resume: MP_OFFLINE\n"))
 930                 goto rb_mp_offline;
 931 #endif

 933 #if defined(__x86)
 934         case C_ST_MP_PAUSED:
 935                 PMD(PMD_SX, ("cpr_resume: MP_PAUSED\n"))
 936                 goto rb_mp_paused;
 937 #endif


 940         default:
 941                 PMD(PMD_SX, ("cpr_resume: others\n"))
 942                 goto rb_others;
 943         }

 945 rb_all:
 946         /*
 947          * perform platform-dependent initialization
 948          */
 949         if (cpr_suspend_succeeded)
 950                 i_cpr_machdep_setup();

 952         /*
 953          * system did not really go down if we jump here
 954          */
 955 rb_dump:
 956         /*
 957          * IMPORTANT:  SENSITIVE RESUME SEQUENCE
 958          *
 959          * DO NOT ADD ANY INITIALIZATION STEP BEFORE THIS POINT!!
 960          */
 961 rb_nodump:
 962         /*
 963          * If we did suspend to RAM, we didn't generate a dump
 964          */
 965         PMD(PMD_SX, ("cpr_resume: CPR DMA callback\n"))
 966         (void) callb_execute_class(CB_CL_CPR_DMA, CB_CODE_CPR_RESUME);
 967         if (cpr_suspend_succeeded) {
 968                 PMD(PMD_SX, ("cpr_resume: CPR RPC callback\n"))
 969                 (void) callb_execute_class(CB_CL_CPR_RPC, CB_CODE_CPR_RESUME);
```

```
 970         }

 972         prom_resume_prepost();
 973 #if !defined(__sparc)
 974         /*
 975          * Need to sync the software clock with the hardware clock.
 976          * On Sparc, this occurs in the sparc-specific cbe.  However
 977          * on x86 this needs to be handled _before_ we bring other cpu's
 978          * back online.  So we call a resume function in timestamp.c
 979          */
 980         if (tsc_resume_in_cyclic == 0)
 981                 tsc_resume();

 983 #endif

 985 #if defined(__sparc)
 986         if (cpr_suspend_succeeded && (boothowto & RB_DEBUG))
 987                 kdi_dvec_cpr_restart();
 988 #endif


 991 #if defined(__x86)
 992 rb_mp_paused:
 993         PT(PT_RMPO);
 994         PMD(PMD_SX, ("resume aux cpus\n"))

 996         if (cpr_suspend_succeeded) {
 997                 cpr_resume_cpus();
 998         } else {
 999                 cpr_unpause_cpus();
1000         }
1001 #endif

1003         /*
1004          * let the tmp callout catch up.
1005          */
1006         PMD(PMD_SX, ("cpr_resume: CPR CALLOUT callback\n"))
1007         (void) callb_execute_class(CB_CL_CPR_CALLOUT, CB_CODE_CPR_RESUME);

1009         i_cpr_enable_intr();

1011         mutex_enter(&cpu_lock);
1012         PMD(PMD_SX, ("cpr_resume: cyclic resume\n"))
1013         cyclic_resume();
1014         mutex_exit(&cpu_lock);

1016         PMD(PMD_SX, ("cpr_resume: handle xc\n"))
1017         i_cpr_handle_xc(0);     /* turn it off to allow xc assertion */

1019         PMD(PMD_SX, ("cpr_resume: CPR POST KERNEL callback\n"))
1020         (void) callb_execute_class(CB_CL_CPR_POST_KERNEL, CB_CODE_CPR_RESUME);

1022         /*
1023          * statistics gathering
1024          */
1025         if (cpr_suspend_succeeded) {
1026                 /*
1027                  * Prevent false alarm in tod_validate() due to tod
1028                  * value change between suspend and resume
1029                  */
1030                 cpr_tod_status_set(TOD_CPR_RESUME_DONE);

1032                 cpr_convert_promtime(&pwron_tv);

1034                 ctp = &cpr_term.tm_shutdown;
1035                 if (sleeptype == CPR_TODISK)
```



```
1036                         CPR_STAT_EVENT_END_TMZ("  write statefile", ctp);
1037                 CPR_STAT_EVENT_END_TMZ("Suspend Total", ctp);

1039                 CPR_STAT_EVENT_START_TMZ("Resume Total", &pwron_tv);

1041                 str = "  prom time";
1042                 CPR_STAT_EVENT_START_TMZ(str, &pwron_tv);
1043                 ctp = &cpr_term.tm_cprboot_start;
1044                 CPR_STAT_EVENT_END_TMZ(str, ctp);

1046                 str = "  read statefile";
1047                 CPR_STAT_EVENT_START_TMZ(str, ctp);
1048                 ctp = &cpr_term.tm_cprboot_end;
1049                 CPR_STAT_EVENT_END_TMZ(str, ctp);
1050         }

1052 rb_stop_kernel_threads:
1053         /*
1054          * Put all threads back to where they belong; get the kernel
1055          * daemons straightened up too. Note that the callback table
1056          * locked during cpr_stop_kernel_threads() is released only
1057          * in cpr_start_kernel_threads(). Ensure modunloading is
1058          * disabled before starting kernel threads, we don't want
1059          * modunload thread to start changing device tree underneath.
1060          */
1061         PMD(PMD_SX, ("cpr_resume: modunload disable\n"))
1062         modunload_disable();
1063         PMD(PMD_SX, ("cpr_resume: start kernel threads\n"))
1064         cpr_start_kernel_threads();

1066 rb_suspend_devices:
1067         CPR_DEBUG(CPR_DEBUG1, "resuming devices...");
1068         CPR_STAT_EVENT_START("  start drivers");

1070         PMD(PMD_SX,
1071             ("cpr_resume: rb_suspend_devices: cpr_resume_uniproc = %d\n",
1072             cpr_resume_uniproc))

1074 #if defined(__x86)
1075         /*
1076          * If cpr_resume_uniproc is set, then pause all the other cpus
1077          * apart from the current cpu, so that broken drivers that think
1078          * that they are on a uniprocessor machine will resume
1079          */
1080         if (cpr_resume_uniproc) {
1081                 mutex_enter(&cpu_lock);
1082                 pause_cpus(NULL, NULL);
1097                 pause_cpus(NULL);
1083                 mutex_exit(&cpu_lock);
1084         }
1085 #endif

1087         /*
1088          * The policy here is to continue resume everything we can if we did
1089          * not successfully finish suspend; and panic if we are coming back
1090          * from a fully suspended system.
1091          */
1092         PMD(PMD_SX, ("cpr_resume: resume devices\n"))
1093         rc = cpr_resume_devices(ddi_root_node(), 0);

1095         cpr_sae(0);

1097         str = "Failed to resume one or more devices.";

1099         if (rc) {
1100                 if (CPR->c_substate == C_ST_DUMP ||
```

```
1101                    (sleeptype == CPR_TORAM &&
1102                    CPR->c_substate == C_ST_NODUMP)) {
1103                        if (cpr_test_point == FORCE_SUSPEND_TO_RAM) {
1104                                PMD(PMD_SX, ("cpr_resume: resume device "
1105                                    "warn\n"))
1106                                cpr_err(CE_WARN, str);
1107                        } else {
1108                                PMD(PMD_SX, ("cpr_resume: resume device "
1109                                    "panic\n"))
1110                                cpr_err(CE_PANIC, str);
1111                        }
1112                } else {
1113                        PMD(PMD_SX, ("cpr_resume: resume device warn\n"))
1114                        cpr_err(CE_WARN, str);
1115                }
1116        }

1118        CPR_STAT_EVENT_END("  start drivers");
1119        CPR_DEBUG(CPR_DEBUG1, "done\n");

1121 #if defined(__x86)
1122        /*
1123         * If cpr_resume_uniproc is set, then unpause all the processors
1124         * that were paused before resuming the drivers
1125         */
1126        if (cpr_resume_uniproc) {
1127                mutex_enter(&cpu_lock);
1128                start_cpus();
1129                mutex_exit(&cpu_lock);
1130        }
1131 #endif

1133        /*
1134         * If we had disabled modunloading in this cpr resume cycle (i.e. we
1135         * resumed from a state earlier than C_ST_SUSPEND_DEVICES), re-enable
1136         * modunloading now.
1137         */
1138        if (CPR->c_substate != C_ST_SUSPEND_DEVICES) {
1139                PMD(PMD_SX, ("cpr_resume: modload enable\n"))
1140                modunload_enable();
1141        }

1143        /*
1144         * Hooks needed by lock manager prior to resuming.
1145         * Refer to code for more comments.
1146         */
1147        PMD(PMD_SX, ("cpr_resume: lock mgr\n"))
1148        cpr_lock_mgr(lm_cprresume);

1150 #if defined(__sparc)
1151        /*
1152         * This is a partial (half) resume during cpr suspend, we
1153         * haven't yet given up on the suspend. On return from here,
1154         * cpr_suspend() will try to reallocate and retry the suspend.
1155         */
1156        if (CPR->c_substate == C_ST_DUMP_NOSPC) {
1157                return (0);
1158        }

1160        if (sleeptype == CPR_TODISK) {
1161 rb_statef_alloc:
1162                cpr_statef_close();

1164 rb_disable_ufs_logging:
1165                /*
1166                 * if ufs logging was disabled, re-enable
```



```
1167                 */
1168                (void) cpr_ufs_logging(1);
1169        }
1170 #endif

1172 rb_pm_reattach_noinvol:
1173        /*
1174         * When pm_reattach_noinvol() succeeds, modunload_thread will
1175         * remain disabled until after cpr suspend passes the
1176         * C_ST_STOP_KERNEL_THREADS state. If any failure happens before
1177         * cpr suspend reaches this state, we'll need to enable modunload
1178         * thread during rollback.
1179         */
1180        if (CPR->c_substate == C_ST_DISABLE_UFS_LOGGING ||
1181            CPR->c_substate == C_ST_STATEF_ALLOC ||
1182            CPR->c_substate == C_ST_SUSPEND_DEVICES ||
1183            CPR->c_substate == C_ST_STOP_KERNEL_THREADS) {
1184                PMD(PMD_SX, ("cpr_resume: reattach noinvol fini\n"))
1185                pm_reattach_noinvol_fini();
1186        }

1188        PMD(PMD_SX, ("cpr_resume: CPR POST USER callback\n"))
1189        (void) callb_execute_class(CB_CL_CPR_POST_USER, CB_CODE_CPR_RESUME);
1190        PMD(PMD_SX, ("cpr_resume: CPR PROMPRINTF callback\n"))
1191        (void) callb_execute_class(CB_CL_CPR_PROMPRINTF, CB_CODE_CPR_RESUME);

1193        PMD(PMD_SX, ("cpr_resume: restore direct levels\n"))
1194        pm_restore_direct_levels();

1196 rb_stop_user_threads:
1197        CPR_DEBUG(CPR_DEBUG1, "starting user threads...");
1198        PMD(PMD_SX, ("cpr_resume: starting user threads\n"))
1199        cpr_start_user_threads();
1200        CPR_DEBUG(CPR_DEBUG1, "done\n");
1201        /*
1202         * Ask Xorg to resume the frame buffer, and wait for it to happen
1203         */
1204        mutex_enter(&srn_clone_lock);
1205        if (srn_signal) {
1206                PMD(PMD_SX, ("cpr_suspend: (*srn_signal)(..., "
1207                    "SRN_NORMAL_RESUME)\n"))
1208                srn_inuse = 1;          /* because (*srn_signal) cv_waits */
1209                (*srn_signal)(SRN_TYPE_APM, SRN_NORMAL_RESUME);
1210                srn_inuse = 0;
1211        } else {
1212                PMD(PMD_SX, ("cpr_suspend: srn_signal NULL\n"))
1213        }
1214        mutex_exit(&srn_clone_lock);

1216 #if defined(__sparc)
1217 rb_mp_offline:
1218        if (cpr_mp_online())
1219                cpr_err(CE_WARN, "Failed to online all the processors.");
1220 #endif

1222 rb_others:
1223        PMD(PMD_SX, ("cpr_resume: dep thread\n"))
1224        pm_dispatch_to_dep_thread(PM_DEP_WK_CPR_RESUME, NULL, NULL,
1225            PM_DEP_WAIT, NULL, 0);

1227        PMD(PMD_SX, ("cpr_resume: CPR PM callback\n"))
1228        (void) callb_execute_class(CB_CL_CPR_PM, CB_CODE_CPR_RESUME);

1230        if (cpr_suspend_succeeded) {
1231                cpr_stat_record_events();
1232        }
```

```
#if defined(__sparc)
        if (sleeptype == CPR_TODISK && !cpr_reusable_mode)
                cpr_clear_definfo();
#endif

        i_cpr_free_cpus();
        CPR_DEBUG(CPR_DEBUG1, "Sending SIGTHAW...");
        PMD(PMD_SX, ("cpr_resume: SIGTHAW\n"))
        cpr_signal_user(SIGTHAW);
        CPR_DEBUG(CPR_DEBUG1, "done\n");

        CPR_STAT_EVENT_END("Resume Total");

        CPR_STAT_EVENT_START_TMZ("WHOLE CYCLE", &wholecycle_tv);
        CPR_STAT_EVENT_END("WHOLE CYCLE");

        if (cpr_debug & CPR_DEBUG1)
                cmn_err(CE_CONT, "\nThe system is back where you left!\n");

        CPR_STAT_EVENT_START("POST CPR DELAY");

#ifdef CPR_STAT
        ctp = &cpr_term.tm_shutdown;
        CPR_STAT_EVENT_START_TMZ("PWROFF TIME", ctp);
        CPR_STAT_EVENT_END_TMZ("PWROFF TIME", &pwron_tv);

        CPR_STAT_EVENT_PRINT();
#endif /* CPR_STAT */

        PMD(PMD_SX, ("cpr_resume returns %x\n", rc))
        return (rc);
}
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   52319 Tue Nov  4 16:25:25 2014
new/usr/src/uts/common/disp/cmt.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_

 320 /*
 321  * Promote PG above it's current parent.
 322  * This is only legal if PG has an equal or greater number of CPUs than its
 323  * parent.
 324  *
 325  * This routine operates on the CPU specific processor group data (for the CPUs
 326  * in the PG being promoted), and may be invoked from a context where one CPU's
 327  * PG data is under construction. In this case the argument "pgdata", if not
 328  * NULL, is a reference to the CPU's under-construction PG data.
 329  */
 330 static void
 331 cmt_hier_promote(pg_cmt_t *pg, cpu_pg_t *pgdata)
 332 {
 333         pg_cmt_t        *parent;
 334         group_t         *children;
 335         cpu_t           *cpu;
 336         group_iter_t    iter;
 337         pg_cpu_itr_t    cpu_iter;
 338         int             r;
 339         int             err;
 340         int             nchildren;

 342         ASSERT(MUTEX_HELD(&cpu_lock));

 344         parent = pg->cmt_parent;
 345         if (parent == NULL) {
 346                 /*
 347                  * Nothing to do
 348                  */
 349                 return;
 350         }

 352         ASSERT(PG_NUM_CPUS((pg_t *)pg) >= PG_NUM_CPUS((pg_t *)parent));

 354         /*
 355          * We're changing around the hierarchy, which is actively traversed
 356          * by the dispatcher. Pause CPUS to ensure exclusivity.
 357          */
 358         pause_cpus(NULL, NULL);
 358         pause_cpus(NULL);

 360         /*
 361          * If necessary, update the parent's sibling set, replacing parent
 362          * with PG.
 363          */
 364         if (parent->cmt_siblings) {
 365                 if (group_remove(parent->cmt_siblings, parent, GRP_NORESIZE)
 366                     != -1) {
 367                         r = group_add(parent->cmt_siblings, pg, GRP_NORESIZE);
 368                         ASSERT(r != -1);
 369                 }
 370         }

 372         /*
 373          * If the parent is at the top of the hierarchy, replace it's entry
 374          * in the root lgroup's group of top level PGs.
 375          */
 376         if (parent->cmt_parent == NULL &&
 377             parent->cmt_siblings != &cmt_root->cl_pgs) {
```



```
 378                 if (group_remove(&cmt_root->cl_pgs, parent, GRP_NORESIZE)
 379                     != -1) {
 380                         r = group_add(&cmt_root->cl_pgs, pg, GRP_NORESIZE);
 381                         ASSERT(r != -1);
 382                 }
 383         }

 385         /*
 386          * We assume (and therefore assert) that the PG being promoted is an
 387          * only child of it's parent. Update the parent's children set
 388          * replacing PG's entry with the parent (since the parent is becoming
 389          * the child). Then have PG and the parent swap children sets and
 390          * children counts.
 391          */
 392         ASSERT(GROUP_SIZE(parent->cmt_children) <= 1);
 393         if (group_remove(parent->cmt_children, pg, GRP_NORESIZE) != -1) {
 394                 r = group_add(parent->cmt_children, parent, GRP_NORESIZE);
 395                 ASSERT(r != -1);
 396         }

 398         children = pg->cmt_children;
 399         pg->cmt_children = parent->cmt_children;
 400         parent->cmt_children = children;

 402         nchildren = pg->cmt_nchildren;
 403         pg->cmt_nchildren = parent->cmt_nchildren;
 404         parent->cmt_nchildren = nchildren;

 406         /*
 407          * Update the sibling references for PG and it's parent
 408          */
 409         pg->cmt_siblings = parent->cmt_siblings;
 410         parent->cmt_siblings = pg->cmt_children;

 412         /*
 413          * Update any cached lineages in the per CPU pg data.
 414          */
 415         PG_CPU_ITR_INIT(pg, cpu_iter);
 416         while ((cpu = pg_cpu_next(&cpu_iter)) != NULL) {
 417                 int             idx;
 418                 int             sz;
 419                 pg_cmt_t        *cpu_pg;
 420                 cpu_pg_t        *pgd;   /* CPU's PG data */

 422                 /*
 423                  * The CPU's whose lineage is under construction still
 424                  * references the bootstrap CPU PG data structure.
 425                  */
 426                 if (pg_cpu_is_bootstrapped(cpu))
 427                         pgd = pgdata;
 428                 else
 429                         pgd = cpu->cpu_pg;

 431                 /*
 432                  * Iterate over the CPU's PGs updating the children
 433                  * of the PG being promoted, since they have a new parent.
 434                  */
 435                 group_iter_init(&iter);
 436                 while ((cpu_pg = group_iterate(&pgd->cmt_pgs, &iter)) != NULL) {
 437                         if (cpu_pg->cmt_parent == pg) {
 438                                 cpu_pg->cmt_parent = parent;
 439                         }
 440                 }

 442                 /*
 443                  * Update the CMT load balancing lineage
```

```
444                  */
445                 if ((idx = group_find(&pgd->cmt_pgs, (void *)pg)) == -1) {
446                         /*
447                          * Unless this is the CPU who's lineage is being
448                          * constructed, the PG being promoted should be
449                          * in the lineage.
450                          */
451                         ASSERT(pg_cpu_is_bootstrapped(cpu));
452                         continue;
453                 }

455                 ASSERT(idx > 0);
456                 ASSERT(GROUP_ACCESS(&pgd->cmt_pgs, idx - 1) == parent);

458                 /*
459                  * Have the child and the parent swap places in the CPU's
460                  * lineage
461                  */
462                 group_remove_at(&pgd->cmt_pgs, idx);
463                 group_remove_at(&pgd->cmt_pgs, idx - 1);
464                 err = group_add_at(&pgd->cmt_pgs, parent, idx);
465                 ASSERT(err == 0);
466                 err = group_add_at(&pgd->cmt_pgs, pg, idx - 1);
467                 ASSERT(err == 0);

469                 /*
470                  * Ensure cmt_lineage references CPU's leaf PG.
471                  * Since cmt_pgs is top-down ordered, the bottom is the last
472                  * element.
473                  */
474                 if ((sz = GROUP_SIZE(&pgd->cmt_pgs)) > 0)
475                         pgd->cmt_lineage = GROUP_ACCESS(&pgd->cmt_pgs, sz - 1);
476         }

478         /*
479          * Update the parent references for PG and it's parent
480          */
481         pg->cmt_parent = parent->cmt_parent;
482         parent->cmt_parent = pg;

484         start_cpus();
485 }
_____unchanged_portion_omitted_

1455 /*
1456  * Prune PG, and all other instances of PG's hardware sharing relationship
1457  * from the CMT PG hierarchy.
1458  *
1459  * This routine operates on the CPU specific processor group data (for the CPUs
1460  * in the PG being pruned), and may be invoked from a context where one CPU's
1461  * PG data is under construction. In this case the argument "pgdata", if not
1462  * NULL, is a reference to the CPU's under-construction PG data.
1463  */
1464 static int
1465 pg_cmt_prune(pg_cmt_t *pg_bad, pg_cmt_t **lineage, int *sz, cpu_pg_t *pgdata)
1466 {
1467         group_t         *hwset, *children;
1468         int             i, j, r, size = *sz;
1469         group_iter_t    hw_iter, child_iter;
1470         pg_cpu_itr_t    cpu_iter;
1471         pg_cmt_t        *pg, *child;
1472         cpu_t           *cpu;
1473         int             cap_needed;
1474         pghw_type_t     hw;

1476         ASSERT(MUTEX_HELD(&cpu_lock));
```



```
1478         /*
1479          * Inform pghw layer that this PG is pruned.
1480          */
1481         pghw_cmt_fini((pghw_t *)pg_bad);

1483         hw = ((pghw_t *)pg_bad)->pghw_hw;

1485         if (hw == PGHW_POW_ACTIVE) {
1486                 cmn_err(CE_NOTE, "!Active CPUPM domain groups look suspect. "
1487                     "Event Based CPUPM Unavailable");
1488         } else if (hw == PGHW_POW_IDLE) {
1489                 cmn_err(CE_NOTE, "!Idle CPUPM domain groups look suspect. "
1490                     "Dispatcher assisted CPUPM disabled.");
1491         }

1493         /*
1494          * Find and eliminate the PG from the lineage.
1495          */
1496         for (i = 0; i < size; i++) {
1497                 if (lineage[i] == pg_bad) {
1498                         for (j = i; j < size - 1; j++)
1499                                 lineage[j] = lineage[j + 1];
1500                         *sz = size - 1;
1501                         break;
1502                 }
1503         }

1505         /*
1506          * We'll prune all instances of the hardware sharing relationship
1507          * represented by pg. But before we do that (and pause CPUs) we need
1508          * to ensure the hierarchy's groups are properly sized.
1509          */
1510         hwset = pghw_set_lookup(hw);

1512         /*
1513          * Blacklist the hardware so future processor groups of this type won't
1514          * participate in CMT thread placement.
1515          *
1516          * XXX
1517          * For heterogeneous system configurations, this might be overkill.
1518          * We may only need to blacklist the illegal PGs, and other instances
1519          * of this hardware sharing relationship may be ok.
1520          */
1521         cmt_hw_blacklisted[hw] = 1;

1523         /*
1524          * For each of the PGs being pruned, ensure sufficient capacity in
1525          * the siblings set for the PG's children
1526          */
1527         group_iter_init(&hw_iter);
1528         while ((pg = group_iterate(hwset, &hw_iter)) != NULL) {
1529                 /*
1530                  * PG is being pruned, but if it is bringing up more than
1531                  * one child, ask for more capacity in the siblings group.
1532                  */
1533                 cap_needed = 0;
1534                 if (pg->cmt_children &&
1535                     GROUP_SIZE(pg->cmt_children) > 1) {
1536                         cap_needed = GROUP_SIZE(pg->cmt_children) - 1;

1538                         group_expand(pg->cmt_siblings,
1539                             GROUP_SIZE(pg->cmt_siblings) + cap_needed);

1541                         /*
1542                          * If this is a top level group, also ensure the
```

```
1543                         * capacity in the root lgrp level CMT grouping.
1544                         */
1545                        if (pg->cmt_parent == NULL &&
1546                            pg->cmt_siblings != &cmt_root->cl_pgs) {
1547                                group_expand(&cmt_root->cl_pgs,
1548                                    GROUP_SIZE(&cmt_root->cl_pgs) + cap_needed);
1549                                cmt_root->cl_npgs += cap_needed;
1550                        }
1551                }
1552        }

1554        /*
1555         * We're operating on the PG hierarchy. Pause CPUs to ensure
1556         * exclusivity with respect to the dispatcher.
1557         */
1558        pause_cpus(NULL, NULL);
1558        pause_cpus(NULL);

1560        /*
1561         * Prune all PG instances of the hardware sharing relationship
1562         * represented by pg.
1563         */
1564        group_iter_init(&hw_iter);
1565        while ((pg = group_iterate(hwset, &hw_iter)) != NULL) {

1567                /*
1568                 * Remove PG from it's group of siblings, if it's there.
1569                 */
1570                if (pg->cmt_siblings) {
1571                        (void) group_remove(pg->cmt_siblings, pg, GRP_NORESIZE);
1572                }
1573                if (pg->cmt_parent == NULL &&
1574                    pg->cmt_siblings != &cmt_root->cl_pgs) {
1575                        (void) group_remove(&cmt_root->cl_pgs, pg,
1576                            GRP_NORESIZE);
1577                }

1579                /*
1580                 * Indicate that no CMT policy will be implemented across
1581                 * this PG.
1582                 */
1583                pg->cmt_policy = CMT_NO_POLICY;

1585                /*
1586                 * Move PG's children from it's children set to it's parent's
1587                 * children set. Note that the parent's children set, and PG's
1588                 * siblings set are the same thing.
1589                 *
1590                 * Because we are iterating over the same group that we are
1591                 * operating on (removing the children), first add all of PG's
1592                 * children to the parent's children set, and once we are done
1593                 * iterating, empty PG's children set.
1594                 */
1595                if (pg->cmt_children != NULL) {
1596                        children = pg->cmt_children;

1598                        group_iter_init(&child_iter);
1599                        while ((child = group_iterate(children, &child_iter))
1600                            != NULL) {
1601                                if (pg->cmt_siblings != NULL) {
1602                                        r = group_add(pg->cmt_siblings, child,
1603                                            GRP_NORESIZE);
1604                                        ASSERT(r == 0);

1606                                        if (pg->cmt_parent == NULL &&
1607                                            pg->cmt_siblings !=
```



```
1608                                            &cmt_root->cl_pgs) {
1609                                                r = group_add(&cmt_root->cl_pgs,
1610                                                    child, GRP_NORESIZE);
1611                                                ASSERT(r == 0);
1612                                        }
1613                                }
1614                        }
1615                        group_empty(pg->cmt_children);
1616                }

1618                /*
1619                 * Reset the callbacks to the defaults
1620                 */
1621                pg_callback_set_defaults((pg_t *)pg);

1623                /*
1624                 * Update all the CPU lineages in each of PG's CPUs
1625                 */
1626                PG_CPU_ITR_INIT(pg, cpu_iter);
1627                while ((cpu = pg_cpu_next(&cpu_iter)) != NULL) {
1628                        pg_cmt_t        *cpu_pg;
1629                        group_iter_t    liter;  /* Iterator for the lineage */
1630                        cpu_pg_t        *cpd;   /* CPU's PG data */

1632                        /*
1633                         * The CPU's lineage is under construction still
1634                         * references the bootstrap CPU PG data structure.
1635                         */
1636                        if (pg_cpu_is_bootstrapped(cpu))
1637                                cpd = pgdata;
1638                        else
1639                                cpd = cpu->cpu_pg;

1641                        /*
1642                         * Iterate over the CPU's PGs updating the children
1643                         * of the PG being promoted, since they have a new
1644                         * parent and siblings set.
1645                         */
1646                        group_iter_init(&liter);
1647                        while ((cpu_pg = group_iterate(&cpd->pgs,
1648                            &liter)) != NULL) {
1649                                if (cpu_pg->cmt_parent == pg) {
1650                                        cpu_pg->cmt_parent = pg->cmt_parent;
1651                                        cpu_pg->cmt_siblings = pg->cmt_siblings;
1652                                }
1653                        }

1655                        /*
1656                         * Update the CPU's lineages
1657                         *
1658                         * Remove the PG from the CPU's group used for CMT
1659                         * scheduling.
1660                         */
1661                        (void) group_remove(&cpd->cmt_pgs, pg, GRP_NORESIZE);
1662                }
1663        }
1664        start_cpus();
1665        return (0);
1666 }

1668 /*
1669  * Disable CMT scheduling
1670  */
1671 static void
1672 pg_cmt_disable(void)
1673 {
```

```
1674         cpu_t           *cpu;

1676         ASSERT(MUTEX_HELD(&cpu_lock));

1678         pause_cpus(NULL, NULL);
1678         pause_cpus(NULL);
1679         cpu = cpu_list;

1681         do {
1682                 if (cpu->cpu_pg)
1683                         group_empty(&cpu->cpu_pg->cmt_pgs);
1684         } while ((cpu = cpu->cpu_next) != cpu_list);

1686         cmt_sched_disabled = 1;
1687         start_cpus();
1688         cmn_err(CE_NOTE, "!CMT thread placement optimizations unavailable");
1689 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   30551 Tue Nov  4 16:25:25 2014
new/usr/src/uts/common/disp/cpupart.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_


319 static int
320 cpupart_move_cpu(cpu_t *cp, cpupart_t *newpp, int forced)
321 {
322         cpupart_t *oldpp;
323         cpu_t   *ncp, *newlist;
324         kthread_t *t;
325         int     move_threads = 1;
326         lgrp_id_t lgrpid;
327         proc_t  *p;
328         int lgrp_diff_lpl;
329         lpl_t   *cpu_lpl;
330         int     ret;
331         boolean_t unbind_all_threads = (forced != 0);

333         ASSERT(MUTEX_HELD(&cpu_lock));
334         ASSERT(newpp != NULL);

336         oldpp = cp->cpu_part;
337         ASSERT(oldpp != NULL);
338         ASSERT(oldpp->cp_ncpus > 0);

340         if (newpp == oldpp) {
341                 /*
342                  * Don't need to do anything.
343                  */
344                 return (0);
345         }

347         cpu_state_change_notify(cp->cpu_id, CPU_CPUPART_OUT);

349         if (!disp_bound_partition(cp, 0)) {
350                 /*
351                  * Don't need to move threads if there are no threads in
352                  * the partition.  Note that threads can't enter the
353                  * partition while we're holding cpu_lock.
354                  */
355                 move_threads = 0;
356         } else if (oldpp->cp_ncpus == 1) {
357                 /*
358                  * The last CPU is removed from a partition which has threads
359                  * running in it. Some of these threads may be bound to this
360                  * CPU.
361                  *
362                  * Attempt to unbind threads from the CPU and from the processor
363                  * set. Note that no threads should be bound to this CPU since
364                  * cpupart_move_threads will refuse to move bound threads to
365                  * other CPUs.
366                  */
367                 (void) cpu_unbind(oldpp->cp_cpulist->cpu_id, B_FALSE);
368                 (void) cpupart_unbind_threads(oldpp, B_FALSE);

370                 if (!disp_bound_partition(cp, 0)) {
371                         /*
372                          * No bound threads in this partition any more
373                          */
374                         move_threads = 0;
375                 } else {
376                         /*
```



```
377                          * There are still threads bound to the partition
378                          */
379                         cpu_state_change_notify(cp->cpu_id, CPU_CPUPART_IN);
380                         return (EBUSY);
381                 }
382         }

384         /*
385          * If forced flag is set unbind any threads from this CPU.
386          * Otherwise unbind soft-bound threads only.
387          */
388         if ((ret = cpu_unbind(cp->cpu_id, unbind_all_threads)) != 0) {
389                 cpu_state_change_notify(cp->cpu_id, CPU_CPUPART_IN);
390                 return (ret);
391         }

393         /*
394          * Stop further threads weak binding to this cpu.
395          */
396         cpu_inmotion = cp;
397         membar_enter();

399         /*
400          * Notify the Processor Groups subsystem that the CPU
401          * will be moving cpu partitions. This is done before
402          * CPUs are paused to provide an opportunity for any
403          * needed memory allocations.
404          */
405         pg_cpupart_out(cp, oldpp);
406         pg_cpupart_in(cp, newpp);

408 again:
409         if (move_threads) {
410                 int loop_count;
411                 /*
412                  * Check for threads strong or weak bound to this CPU.
413                  */
414                 for (loop_count = 0; disp_bound_threads(cp, 0); loop_count++) {
415                         if (loop_count >= 5) {
416                                 cpu_state_change_notify(cp->cpu_id,
417                                     CPU_CPUPART_IN);
418                                 pg_cpupart_out(cp, newpp);
419                                 pg_cpupart_in(cp, oldpp);
420                                 cpu_inmotion = NULL;
421                                 return (EBUSY); /* some threads still bound */
422                         }
423                         delay(1);
424                 }
425         }

427         /*
428          * Before we actually start changing data structures, notify
429          * the cyclic subsystem that we want to move this CPU out of its
430          * partition.
431          */
432         if (!cyclic_move_out(cp)) {
433                 /*
434                  * This CPU must be the last CPU in a processor set with
435                  * a bound cyclic.
436                  */
437                 cpu_state_change_notify(cp->cpu_id, CPU_CPUPART_IN);
438                 pg_cpupart_out(cp, newpp);
439                 pg_cpupart_in(cp, oldpp);
440                 cpu_inmotion = NULL;
441                 return (EBUSY);
442         }
```

```
444         pause_cpus(cp, NULL);
444         pause_cpus(cp);

446         if (move_threads) {
447                 /*
448                  * The thread on cpu before the pause thread may have read
449                  * cpu_inmotion before we raised the barrier above.  Check
450                  * again.
451                  */
452                 if (disp_bound_threads(cp, 1)) {
453                         start_cpus();
454                         goto again;
455                 }

457         }

459         /*
460          * Now that CPUs are paused, let the PG subsystem perform
461          * any necessary data structure updates.
462          */
463         pg_cpupart_move(cp, oldpp, newpp);

465         /* save this cpu's lgroup -- it'll be the same in the new partition */
466         lgrpid = cp->cpu_lpl->lpl_lgrpid;

468         cpu_lpl = cp->cpu_lpl;
469         /*
470          * let the lgroup framework know cp has left the partition
471          */
472         lgrp_config(LGRP_CONFIG_CPUPART_DEL, (uintptr_t)cp, lgrpid);

474         /* move out of old partition */
475         oldpp->cp_ncpus--;
476         if (oldpp->cp_ncpus > 0) {

478                 ncp = cp->cpu_prev_part->cpu_next_part = cp->cpu_next_part;
479                 cp->cpu_next_part->cpu_prev_part = cp->cpu_prev_part;
480                 if (oldpp->cp_cpulist == cp) {
481                         oldpp->cp_cpulist = ncp;
482                 }
483         } else {
484                 ncp = oldpp->cp_cpulist = NULL;
485                 cp_numparts_nonempty--;
486                 ASSERT(cp_numparts_nonempty != 0);
487         }
488         oldpp->cp_gen++;

490         /* move into new partition */
491         newlist = newpp->cp_cpulist;
492         if (newlist == NULL) {
493                 newpp->cp_cpulist = cp->cpu_next_part = cp->cpu_prev_part = cp;
494                 cp_numparts_nonempty++;
495                 ASSERT(cp_numparts_nonempty != 0);
496         } else {
497                 cp->cpu_next_part = newlist;
498                 cp->cpu_prev_part = newlist->cpu_prev_part;
499                 newlist->cpu_prev_part->cpu_next_part = cp;
500                 newlist->cpu_prev_part = cp;
501         }
502         cp->cpu_part = newpp;
503         newpp->cp_ncpus++;
504         newpp->cp_gen++;

506         ASSERT(bitset_is_null(&newpp->cp_haltset));
507         ASSERT(bitset_is_null(&oldpp->cp_haltset));
```



```
509         /*
510          * let the lgroup framework know cp has entered the partition
511          */
512         lgrp_config(LGRP_CONFIG_CPUPART_ADD, (uintptr_t)cp, lgrpid);

514         /*
515          * If necessary, move threads off processor.
516          */
517         if (move_threads) {
518                 ASSERT(ncp != NULL);

520                 /*
521                  * Walk thru the active process list to look for
522                  * threads that need to have a new home lgroup,
523                  * or the last CPU they run on is the same CPU
524                  * being moved out of the partition.
525                  */

527                 for (p = practive; p != NULL; p = p->p_next) {

529                         t = p->p_tlist;

531                         if (t == NULL)
532                                 continue;

534                         lgrp_diff_lpl = 0;

536                         do {

538                                 ASSERT(t->t_lpl != NULL);

540                                 /*
541                                  * Update the count of how many threads are
542                                  * in this CPU's lgroup but have a different lpl
543                                  */

545                                 if (t->t_lpl != cpu_lpl &&
546                                     t->t_lpl->lpl_lgrpid == lgrpid)
547                                         lgrp_diff_lpl++;
548                                 /*
549                                  * If the lgroup that t is assigned to no
550                                  * longer has any CPUs in t's partition,
551                                  * we'll have to choose a new lgroup for t.
552                                  */

554                                 if (!LGRP_CPUS_IN_PART(t->t_lpl->lpl_lgrpid,
555                                     t->t_cpupart)) {
556                                         lgrp_move_thread(t,
557                                             lgrp_choose(t, t->t_cpupart), 0);
558                                 }

560                                 /*
561                                  * make sure lpl points to our own partition
562                                  */
563                                 ASSERT(t->t_lpl >= t->t_cpupart->cp_lgrploads &&
564                                     (t->t_lpl < t->t_cpupart->cp_lgrploads +
565                                     t->t_cpupart->cp_nlgrploads));

567                                 ASSERT(t->t_lpl->lpl_ncpu > 0);

569                                 /* Update CPU last ran on if it was this CPU */
570                                 if (t->t_cpu == cp && t->t_cpupart == oldpp &&
571                                     t->t_bound_cpu != cp) {
572                                         t->t_cpu = disp_lowpri_cpu(ncp,
573                                             t->t_lpl, t->t_pri, NULL);
```

```
574                                 }
575                                 t = t->t_forw;
576                         } while (t != p->p_tlist);

578                         /*
579                          * Didn't find any threads in the same lgroup as this
580                          * CPU with a different lpl, so remove the lgroup from
581                          * the process lgroup bitmask.
582                          */

584                         if (lgrp_diff_lpl)
585                                 klgrpset_del(p->p_lgrpset, lgrpid);
586                 }

588                 /*
589                  * Walk thread list looking for threads that need to be
590                  * rehomed, since there are some threads that are not in
591                  * their process's p_tlist.
592                  */

594                 t = curthread;

596                 do {
597                         ASSERT(t != NULL && t->t_lpl != NULL);

599                         /*
600                          * If the lgroup that t is assigned to no
601                          * longer has any CPUs in t's partition,
602                          * we'll have to choose a new lgroup for t.
603                          * Also, choose best lgroup for home when
604                          * thread has specified lgroup affinities,
605                          * since there may be an lgroup with more
606                          * affinity available after moving CPUs
607                          * around.
608                          */
609                         if (!LGRP_CPUS_IN_PART(t->t_lpl->lpl_lgrpid,
610                             t->t_cpupart) || t->t_lgrp_affinity) {
611                                 lgrp_move_thread(t,
612                                     lgrp_choose(t, t->t_cpupart), 1);
613                         }

615                         /* make sure lpl points to our own partition */
616                         ASSERT((t->t_lpl >= t->t_cpupart->cp_lgrploads) &&
617                             (t->t_lpl < t->t_cpupart->cp_lgrploads +
618                             t->t_cpupart->cp_nlgrploads));

620                         ASSERT(t->t_lpl->lpl_ncpu > 0);

622                         /* Update CPU last ran on if it was this CPU */
623                         if (t->t_cpu == cp && t->t_cpupart == oldpp &&
624                             t->t_bound_cpu != cp) {
625                                 t->t_cpu = disp_lowpri_cpu(ncp, t->t_lpl,
626                                     t->t_pri, NULL);
627                         }

629                         t = t->t_next;
630                 } while (t != curthread);

632                 /*
633                  * Clear off the CPU's run queue, and the kp queue if the
634                  * partition is now empty.
635                  */
636                 disp_cpu_inactive(cp);

638                 /*
639                  * Make cp switch to a thread from the new partition.
```



```
640                  */
641                 cp->cpu_runrun = 1;
642                 cp->cpu_kprunrun = 1;
643         }

645         cpu_inmotion = NULL;
646         start_cpus();

648         /*
649          * Let anyone interested know that cpu has been added to the set.
650          */
651         cpu_state_change_notify(cp->cpu_id, CPU_CPUPART_IN);

653         /*
654          * Now let the cyclic subsystem know that it can reshuffle cyclics
655          * bound to the new processor set.
656          */
657         cyclic_move_in(cp);

659         return (0);
660 }
```

*unchanged_portion_omitted_*

```
812 /*
813  * Create a new partition.  On MP systems, this also allocates a
814  * kpreempt disp queue for that partition.
815  */
816 int
817 cpupart_create(psetid_t *psid)
818 {
819         cpupart_t       *pp;

821         ASSERT(pool_lock_held());

823         pp = kmem_zalloc(sizeof (cpupart_t), KM_SLEEP);
824         pp->cp_nlgrploads = lgrp_plat_max_lgrps();
825         pp->cp_lgrploads = kmem_zalloc(sizeof (lpl_t) * pp->cp_nlgrploads,
826             KM_SLEEP);

828         mutex_enter(&cpu_lock);
829         if (cp_numparts == cp_max_numparts) {
830                 mutex_exit(&cpu_lock);
831                 kmem_free(pp->cp_lgrploads, sizeof (lpl_t) * pp->cp_nlgrploads);
832                 pp->cp_lgrploads = NULL;
833                 kmem_free(pp, sizeof (cpupart_t));
834                 return (ENOMEM);
835         }
836         cp_numparts++;
837         /* find the next free partition ID */
838         while (cpupart_find(CPTOPS(cp_id_next)) != NULL)
839                 cp_id_next++;
840         pp->cp_id = cp_id_next++;
841         pp->cp_ncpus = 0;
842         pp->cp_cpulist = NULL;
843         pp->cp_attr = 0;
844         klgrpset_clear(pp->cp_lgrpset);
845         pp->cp_kp_queue.disp_maxrunpri = -1;
846         pp->cp_kp_queue.disp_max_unbound_pri = -1;
847         pp->cp_kp_queue.disp_cpu = NULL;
848         pp->cp_gen = 0;
849         DISP_LOCK_INIT(&pp->cp_kp_queue.disp_lock);
850         *psid = CPTOPS(pp->cp_id);
851         disp_kp_alloc(&pp->cp_kp_queue, v.v_nglobpris);
852         cpupart_kstat_create(pp);
853         cpupart_lpl_initialize(pp);
```

```
855         bitset_init(&pp->cp_cmt_pgs);

857         /*
858          * Initialize and size the partition's bitset of halted CPUs.
859          */
860         bitset_init_fanout(&pp->cp_haltset, cp_haltset_fanout);
861         bitset_resize(&pp->cp_haltset, max_ncpus);

863         /*
864          * Pause all CPUs while changing the partition list, to make sure
865          * the clock thread (which traverses the list without holding
866          * cpu_lock) isn't running.
867          */
868         pause_cpus(NULL, NULL);
868         pause_cpus(NULL);
869         pp->cp_next = cp_list_head;
870         pp->cp_prev = cp_list_head->cp_prev;
871         cp_list_head->cp_prev->cp_next = pp;
872         cp_list_head->cp_prev = pp;
873         start_cpus();
874         mutex_exit(&cpu_lock);

876         return (0);
877 }
_____unchanged_portion_omitted_

949 /*
950  * Destroy a partition.
951  */
952 int
953 cpupart_destroy(psetid_t psid)
954 {
955         cpu_t   *cp, *first_cp;
956         cpupart_t *pp, *newpp;
957         int     err = 0;

959         ASSERT(pool_lock_held());
960         mutex_enter(&cpu_lock);

962         pp = cpupart_find(psid);
963         if (pp == NULL || pp == &cp_default) {
964                 mutex_exit(&cpu_lock);
965                 return (EINVAL);
966         }

968         /*
969          * Unbind all the threads currently bound to the partition.
970          */
971         err = cpupart_unbind_threads(pp, B_TRUE);
972         if (err) {
973                 mutex_exit(&cpu_lock);
974                 return (err);
975         }

977         newpp = &cp_default;
978         while ((cp = pp->cp_cpulist) != NULL) {
979                 if (err = cpupart_move_cpu(cp, newpp, 0)) {
980                         mutex_exit(&cpu_lock);
981                         return (err);
982                 }
983         }

985         ASSERT(bitset_is_null(&pp->cp_cmt_pgs));
986         ASSERT(bitset_is_null(&pp->cp_haltset));
```



```
988         /*
989          * Teardown the partition's group of active CMT PGs and halted
990          * CPUs now that they have all left.
991          */
992         bitset_fini(&pp->cp_cmt_pgs);
993         bitset_fini(&pp->cp_haltset);

995         /*
996          * Reset the pointers in any offline processors so they won't
997          * try to rejoin the destroyed partition when they're turned
998          * online.
999          */
1000        first_cp = cp = CPU;
1001        do {
1002                if (cp->cpu_part == pp) {
1003                        ASSERT(cp->cpu_flags & CPU_OFFLINE);
1004                        cp->cpu_part = newpp;
1005                }
1006                cp = cp->cpu_next;
1007        } while (cp != first_cp);

1009        /*
1010         * Pause all CPUs while changing the partition list, to make sure
1011         * the clock thread (which traverses the list without holding
1012         * cpu_lock) isn't running.
1013         */
1014        pause_cpus(NULL, NULL);
1014        pause_cpus(NULL);
1015        pp->cp_prev->cp_next = pp->cp_next;
1016        pp->cp_next->cp_prev = pp->cp_prev;
1017        if (cp_list_head == pp)
1018                cp_list_head = pp->cp_next;
1019        start_cpus();

1021        if (cp_id_next > pp->cp_id)
1022                cp_id_next = pp->cp_id;

1024        if (pp->cp_kstat)
1025                kstat_delete(pp->cp_kstat);

1027        cp_numparts--;

1029        disp_kp_free(&pp->cp_kp_queue);

1031        cpupart_lpl_teardown(pp);

1033        kmem_free(pp, sizeof (cpupart_t));
1034        mutex_exit(&cpu_lock);

1036        return (err);
1037 }
______unchanged_portion_omitted_
```

```
**********************************************************
   70612 Tue Nov  4 16:25:25 2014
new/usr/src/uts/common/disp/disp.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_


 312 /*
 313  * For each CPU, allocate new dispatch queues
 314  * with the stated number of priorities.
 315  */
 316 static void
 317 cpu_dispqalloc(int numpris)
 318 {
 319         cpu_t   *cpup;
 320         struct disp_queue_info  *disp_mem;
 321         int i, num;

 323         ASSERT(MUTEX_HELD(&cpu_lock));

 325         disp_mem = kmem_zalloc(NCPU *
 326             sizeof (struct disp_queue_info), KM_SLEEP);

 328         /*
 329          * This routine must allocate all of the memory before stopping
 330          * the cpus because it must not sleep in kmem_alloc while the
 331          * CPUs are stopped.  Locks they hold will not be freed until they
 332          * are restarted.
 333          */
 334         i = 0;
 335         cpup = cpu_list;
 336         do {
 337                 disp_dq_alloc(&disp_mem[i], numpris, cpup->cpu_disp);
 338                 i++;
 339                 cpup = cpup->cpu_next;
 340         } while (cpup != cpu_list);
 341         num = i;

 343         pause_cpus(NULL, NULL);
 343         pause_cpus(NULL);
 344         for (i = 0; i < num; i++)
 345                 disp_dq_assign(&disp_mem[i], numpris);
 346         start_cpus();

 348         /*
 349          * I must free all of the memory after starting the cpus because
 350          * I can not risk sleeping in kmem_free while the cpus are stopped.
 351          */
 352         for (i = 0; i < num; i++)
 353                 disp_dq_free(&disp_mem[i]);

 355         kmem_free(disp_mem, NCPU * sizeof (struct disp_queue_info));
 356 }
______unchanged_portion_omitted_
```

```
**********************************************************
   94655 Tue Nov  4 16:25:25 2014
new/usr/src/uts/common/os/cpu.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
```

```
/*
 * 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) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

/*
 * Architecture-independent CPU control functions.
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/var.h>
#include <sys/thread.h>
#include <sys/cpuvar.h>
#include <sys/cpu_event.h>
#include <sys/kstat.h>
#include <sys/uadmin.h>
#include <sys/systm.h>
#include <sys/errno.h>
#include <sys/cmn_err.h>
#include <sys/procset.h>
#include <sys/processor.h>
#include <sys/debug.h>
#include <sys/cpupart.h>
#include <sys/lgrp.h>
#include <sys/pset.h>
#include <sys/pghw.h>
#include <sys/kmem.h>
#include <sys/kmem_impl.h>      /* to set per-cpu kmem_cache offset */
#include <sys/atomic.h>
#include <sys/callb.h>
#include <sys/vtrace.h>
#include <sys/cyclic.h>
#include <sys/bitmap.h>
#include <sys/nvpair.h>
#include <sys/pool_pset.h>
#include <sys/msacct.h>
#include <sys/time.h>
#include <sys/archsystm.h>
#include <sys/sdt.h>
#if defined(__x86) || defined(__amd64)
```



```
#include <sys/x86_archext.h>
#endif
#include <sys/callo.h>

extern int      mp_cpu_start(cpu_t *);
extern int      mp_cpu_stop(cpu_t *);
extern int      mp_cpu_poweron(cpu_t *);
extern int      mp_cpu_poweroff(cpu_t *);
extern int      mp_cpu_configure(int);
extern int      mp_cpu_unconfigure(int);
extern void     mp_cpu_faulted_enter(cpu_t *);
extern void     mp_cpu_faulted_exit(cpu_t *);

extern int cmp_cpu_to_chip(processorid_t cpuid);
#ifdef __sparcv9
extern char *cpu_fru_fmri(cpu_t *cp);
#endif

static void cpu_add_active_internal(cpu_t *cp);
static void cpu_remove_active(cpu_t *cp);
static void cpu_info_kstat_create(cpu_t *cp);
static void cpu_info_kstat_destroy(cpu_t *cp);
static void cpu_stats_kstat_create(cpu_t *cp);
static void cpu_stats_kstat_destroy(cpu_t *cp);

static int cpu_sys_stats_ks_update(kstat_t *ksp, int rw);
static int cpu_vm_stats_ks_update(kstat_t *ksp, int rw);
static int cpu_stat_ks_update(kstat_t *ksp, int rw);
static int cpu_state_change_hooks(int, cpu_setup_t, cpu_setup_t);

/*
 * cpu_lock protects ncpus, ncpus_online, cpu_flag, cpu_list, cpu_active,
 * max_cpu_seqid_ever, and dispatch queue reallocations.  The lock ordering with
 * respect to related locks is:
 *
 *      cpu_lock --> thread_free_lock  --->  p_lock  --->  thread_lock()
 *
 * Warning:  Certain sections of code do not use the cpu_lock when
 * traversing the cpu_list (e.g. mutex_vector_enter(), clock()).  Since
 * all cpus are paused during modifications to this list, a solution
 * to protect the list is too either disable kernel preemption while
 * walking the list, *or* recheck the cpu_next pointer at each
 * iteration in the loop.  Note that in no cases can any cached
 * copies of the cpu pointers be kept as they may become invalid.
 */
kmutex_t        cpu_lock;
cpu_t           *cpu_list;              /* list of all CPUs */
cpu_t           *clock_cpu_list;        /* used by clock to walk CPUs */
cpu_t           *cpu_active;            /* list of active CPUs */
static cpuset_t cpu_available;          /* set of available CPUs */
cpuset_t        cpu_seqid_inuse;        /* which cpu_seqids are in use */

cpu_t           **cpu_seq;              /* ptrs to CPUs, indexed by seq_id */

/*
 * max_ncpus keeps the max cpus the system can have. Initially
 * it's NCPU, but since most archs scan the devtree for cpus
 * fairly early on during boot, the real max can be known before
 * ncpus is set (useful for early NCPU based allocations).
 */
int max_ncpus = NCPU;
/*
 * platforms that set max_ncpus to maxiumum number of cpus that can be
 * dynamically added will set boot_max_ncpus to the number of cpus found
 * at device tree scan time during boot.
 */
```

```
128 int boot_max_ncpus = -1;
129 int boot_ncpus = -1;
130 /*
131  * Maximum possible CPU id.  This can never be >= NCPU since NCPU is
132  * used to size arrays that are indexed by CPU id.
133  */
134 processorid_t max_cpuid = NCPU - 1;

136 /*
137  * Maximum cpu_seqid was given. This number can only grow and never shrink. It
138  * can be used to optimize NCPU loops to avoid going through CPUs which were
139  * never on-line.
140  */
141 processorid_t max_cpu_seqid_ever = 0;

143 int ncpus = 1;
144 int ncpus_online = 1;

146 /*
147  * CPU that we're trying to offline.  Protected by cpu_lock.
148  */
149 cpu_t *cpu_inmotion;

151 /*
152  * Can be raised to suppress further weakbinding, which are instead
153  * satisfied by disabling preemption.  Must be raised/lowered under cpu_lock,
154  * while individual thread weakbinding synchronization is done under thread
155  * lock.
156  */
157 int weakbindingbarrier;

159 /*
160  * Variables used in pause_cpus().
161  */
162 static volatile char safe_list[NCPU];

164 static struct _cpu_pause_info {
165         int             cp_spl;         /* spl saved in pause_cpus() */
166         volatile int    cp_go;          /* Go signal sent after all ready */
167         int             cp_count;       /* # of CPUs to pause */
168         ksema_t         cp_sem;         /* synch pause_cpus & cpu_pause */
169         kthread_id_t    cp_paused;
170         void            *(*cp_func)(void *);
171 #endif /* ! codereview */
172 } cpu_pause_info;

174 static kmutex_t pause_free_mutex;
175 static kcondvar_t pause_free_cv;

170 void *(*cpu_pause_func)(void *) = NULL;


178 static struct cpu_sys_stats_ks_data {
179         kstat_named_t cpu_ticks_idle;
180         kstat_named_t cpu_ticks_user;
181         kstat_named_t cpu_ticks_kernel;
182         kstat_named_t cpu_ticks_wait;
183         kstat_named_t cpu_nsec_idle;
184         kstat_named_t cpu_nsec_user;
185         kstat_named_t cpu_nsec_kernel;
186         kstat_named_t cpu_nsec_dtrace;
187         kstat_named_t cpu_nsec_intr;
188         kstat_named_t cpu_load_intr;
189         kstat_named_t wait_ticks_io;
190         kstat_named_t dtrace_probes;
191         kstat_named_t bread;
```



```
192         kstat_named_t bwrite;
193         kstat_named_t lread;
194         kstat_named_t lwrite;
195         kstat_named_t phread;
196         kstat_named_t phwrite;
197         kstat_named_t pswitch;
198         kstat_named_t trap;
199         kstat_named_t intr;
200         kstat_named_t syscall;
201         kstat_named_t sysread;
202         kstat_named_t syswrite;
203         kstat_named_t sysfork;
204         kstat_named_t sysvfork;
205         kstat_named_t sysexec;
206         kstat_named_t readch;
207         kstat_named_t writech;
208         kstat_named_t rcvint;
209         kstat_named_t xmtint;
210         kstat_named_t mdmint;
211         kstat_named_t rawch;
212         kstat_named_t canch;
213         kstat_named_t outch;
214         kstat_named_t msg;
215         kstat_named_t sema;
216         kstat_named_t namei;
217         kstat_named_t ufsiget;
218         kstat_named_t ufsdirblk;
219         kstat_named_t ufsipage;
220         kstat_named_t ufsinopage;
221         kstat_named_t procovf;
222         kstat_named_t intrthread;
223         kstat_named_t intrblk;
224         kstat_named_t intrunpin;
225         kstat_named_t idlethread;
226         kstat_named_t inv_swtch;
227         kstat_named_t nthreads;
228         kstat_named_t cpumigrate;
229         kstat_named_t xcalls;
230         kstat_named_t mutex_adenters;
231         kstat_named_t rw_rdfails;
232         kstat_named_t rw_wrfails;
233         kstat_named_t modload;
234         kstat_named_t modunload;
235         kstat_named_t bawrite;
236         kstat_named_t iowait;
237 } cpu_sys_stats_ks_data_template = {
_____unchanged_portion_omitted_

751 /*
752  * This routine is called to place the CPUs in a safe place so that
753  * one of them can be taken off line or placed on line.  What we are
754  * trying to do here is prevent a thread from traversing the list
755  * of active CPUs while we are changing it or from getting placed on
756  * the run queue of a CPU that has just gone off line.  We do this by
757  * creating a thread with the highest possible prio for each CPU and
758  * having it call this routine.  The advantage of this method is that
759  * we can eliminate all checks for CPU_ACTIVE in the disp routines.
760  * This makes disp faster at the expense of making p_online() slower
761  * which is a good trade off.
762  */
763 static void
764 cpu_pause(int index)
765 {
766         int s;
767         struct _cpu_pause_info *cpi = &cpu_pause_info;
768         volatile char *safe = &safe_list[index];
```

```
 769         long    lindex = index;

 771         ASSERT((curthread->t_bound_cpu != NULL) || (*safe == PAUSE_DIE));

 773         while (*safe != PAUSE_DIE) {
 774                 *safe = PAUSE_READY;
 775                 membar_enter();         /* make sure stores are flushed */
 776                 sema_v(&cpi->cp_sem);   /* signal requesting thread */

 778                 /*
 779                  * Wait here until all pause threads are running.  That
 780                  * indicates that it's safe to do the spl.  Until
 781                  * cpu_pause_info.cp_go is set, we don't want to spl
 782                  * because that might block clock interrupts needed
 783                  * to preempt threads on other CPUs.
 784                  */
 785                 while (cpi->cp_go == 0)
 786                         ;
 787                 /*
 788                  * Even though we are at the highest disp prio, we need
 789                  * to block out all interrupts below LOCK_LEVEL so that
 790                  * an intr doesn't come in, wake up a thread, and call
 791                  * setbackdq/setfrontdq.
 792                  */
 793                 s = splhigh();
 794                 /*
 795                  * if cp_func has been set then call it using index as the
 796                  * argument, currently only used by cpr_suspend_cpus().
 797                  * This function is used as the code to execute on the
 798                  * "paused" cpu's when a machine comes out of a sleep state
 799                  * and CPU's were powered off.  (could also be used for
 800                  * hotplugging CPU's).
 790                  * if cpu_pause_func() has been set then call it using
 791                  * index as the argument, currently only used by
 792                  * cpr_suspend_cpus().  This function is used as the
 793                  * code to execute on the "paused" cpu's when a machine
 794                  * comes out of a sleep state and CPU's were powered off.
 795                  * (could also be used for hotplugging CPU's).
 801                  */
 802                 if (cpi->cp_func != NULL)
 803                         (*cpi->cp_func)((void *)lindex);
 797                 if (cpu_pause_func != NULL)
 798                         (*cpu_pause_func)((void *)lindex);

 805                 mach_cpu_pause(safe);

 807                 splx(s);
 808                 /*
 809                  * Waiting is at an end. Switch out of cpu_pause
 810                  * loop and resume useful work.
 811                  */
 812                 swtch();
 813         }

 815         mutex_enter(&pause_free_mutex);
 816         *safe = PAUSE_DEAD;
 817         cv_broadcast(&pause_free_cv);
 818         mutex_exit(&pause_free_mutex);
 819 }
```

unchanged_portion_omitted_

```
 974 /*
 975  * Pause all of the CPUs except the one we are on by creating a high
 976  * priority thread bound to those CPUs.
 977  *
```



```
 978  * Note that one must be extremely careful regarding code
 979  * executed while CPUs are paused.  Since a CPU may be paused
 980  * while a thread scheduling on that CPU is holding an adaptive
 981  * lock, code executed with CPUs paused must not acquire adaptive
 982  * (or low-level spin) locks.  Also, such code must not block,
 983  * since the thread that is supposed to initiate the wakeup may
 984  * never run.
 985  *
 986  * With a few exceptions, the restrictions on code executed with CPUs
 987  * paused match those for code executed at high-level interrupt
 988  * context.
 989  */
 990 void
 991 pause_cpus(cpu_t *off_cp, void *(*func)(void *))
 986 pause_cpus(cpu_t *off_cp)
 992 {
 993         processorid_t   cpu_id;
 994         int             i;
 995         struct _cpu_pause_info  *cpi = &cpu_pause_info;

 997         ASSERT(MUTEX_HELD(&cpu_lock));
 998         ASSERT(cpi->cp_paused == NULL);
 999         cpi->cp_count = 0;
1000         cpi->cp_go = 0;
1001         for (i = 0; i < NCPU; i++)
1002                 safe_list[i] = PAUSE_IDLE;
1003         kpreempt_disable();

1005         cpi->cp_func = func;

1007 #endif /* ! codereview */
1008         /*
1009          * If running on the cpu that is going offline, get off it.
1010          * This is so that it won't be necessary to rechoose a CPU
1011          * when done.
1012          */
1013         if (CPU == off_cp)
1014                 cpu_id = off_cp->cpu_next_part->cpu_id;
1015         else
1016                 cpu_id = CPU->cpu_id;
1017         affinity_set(cpu_id);

1019         /*
1020          * Start the pause threads and record how many were started
1021          */
1022         cpi->cp_count = cpu_pause_start(cpu_id);

1024         /*
1025          * Now wait for all CPUs to be running the pause thread.
1026          */
1027         while (cpi->cp_count > 0) {
1028                 /*
1029                  * Spin reading the count without grabbing the disp
1030                  * lock to make sure we don't prevent the pause
1031                  * threads from getting the lock.
1032                  */
1033                 while (sema_held(&cpi->cp_sem))
1034                         ;
1035                 if (sema_tryp(&cpi->cp_sem))
1036                         --cpi->cp_count;
1037         }
1038         cpi->cp_go = 1;                 /* all have reached cpu_pause */

1040         /*
1041          * Now wait for all CPUs to spl. (Transition from PAUSE_READY
1042          * to PAUSE_WAIT.)
```

```
1043          */
1044         for (i = 0; i < NCPU; i++) {
1045                 while (safe_list[i] != PAUSE_WAIT)
1046                         ;
1047         }
1048         cpi->cp_spl = splhigh();        /* block dispatcher on this CPU */
1049         cpi->cp_paused = curthread;
1050 }

1052 /*
1053  * Check whether the current thread has CPUs paused
1054  */
1055 int
1056 cpus_paused(void)
1057 {
1058         if (cpu_pause_info.cp_paused != NULL) {
1059                 ASSERT(cpu_pause_info.cp_paused == curthread);
1060                 return (1);
1061         }
1062         return (0);
1063 }

1065 static cpu_t *
1066 cpu_get_all(processorid_t cpun)
1067 {
1068         ASSERT(MUTEX_HELD(&cpu_lock));

1070         if (cpun >= NCPU || cpun < 0 || !CPU_IN_SET(cpu_available, cpun))
1071                 return (NULL);
1072         return (cpu[cpun]);
1073 }

1075 /*
1076  * Check whether cpun is a valid processor id and whether it should be
1077  * visible from the current zone. If it is, return a pointer to the
1078  * associated CPU structure.
1079  */
1080 cpu_t *
1081 cpu_get(processorid_t cpun)
1082 {
1083         cpu_t *c;

1085         ASSERT(MUTEX_HELD(&cpu_lock));
1086         c = cpu_get_all(cpun);
1087         if (c != NULL && !INGLOBALZONE(curproc) && pool_pset_enabled() &&
1088             zone_pset_get(curproc->p_zone) != cpupart_query_cpu(c))
1089                 return (NULL);
1090         return (c);
1091 }

1093 /*
1094  * The following functions should be used to check CPU states in the kernel.
1095  * They should be invoked with cpu_lock held.  Kernel subsystems interested
1096  * in CPU states should *not* use cpu_get_state() and various P_ONLINE/etc
1097  * states.  Those are for user-land (and system call) use only.
1098  */

1100 /*
1101  * Determine whether the CPU is online and handling interrupts.
1102  */
1103 int
1104 cpu_is_online(cpu_t *cpu)
1105 {
1106         ASSERT(MUTEX_HELD(&cpu_lock));
1107         return (cpu_flagged_online(cpu->cpu_flags));
1108 }
```



```
1110 /*
1111  * Determine whether the CPU is offline (this includes spare and faulted).
1112  */
1113 int
1114 cpu_is_offline(cpu_t *cpu)
1115 {
1116         ASSERT(MUTEX_HELD(&cpu_lock));
1117         return (cpu_flagged_offline(cpu->cpu_flags));
1118 }

1120 /*
1121  * Determine whether the CPU is powered off.
1122  */
1123 int
1124 cpu_is_poweredoff(cpu_t *cpu)
1125 {
1126         ASSERT(MUTEX_HELD(&cpu_lock));
1127         return (cpu_flagged_poweredoff(cpu->cpu_flags));
1128 }

1130 /*
1131  * Determine whether the CPU is handling interrupts.
1132  */
1133 int
1134 cpu_is_nointr(cpu_t *cpu)
1135 {
1136         ASSERT(MUTEX_HELD(&cpu_lock));
1137         return (cpu_flagged_nointr(cpu->cpu_flags));
1138 }

1140 /*
1141  * Determine whether the CPU is active (scheduling threads).
1142  */
1143 int
1144 cpu_is_active(cpu_t *cpu)
1145 {
1146         ASSERT(MUTEX_HELD(&cpu_lock));
1147         return (cpu_flagged_active(cpu->cpu_flags));
1148 }

1150 /*
1151  * Same as above, but these require cpu_flags instead of cpu_t pointers.
1152  */
1153 int
1154 cpu_flagged_online(cpu_flag_t cpu_flags)
1155 {
1156         return (cpu_flagged_active(cpu_flags) &&
1157             (cpu_flags & CPU_ENABLE));
1158 }

1160 int
1161 cpu_flagged_offline(cpu_flag_t cpu_flags)
1162 {
1163         return (((cpu_flags & CPU_POWEROFF) == 0) &&
1164             ((cpu_flags & (CPU_READY | CPU_OFFLINE)) != CPU_READY));
1165 }

1167 int
1168 cpu_flagged_poweredoff(cpu_flag_t cpu_flags)
1169 {
1170         return ((cpu_flags & CPU_POWEROFF) == CPU_POWEROFF);
1171 }

1173 int
1174 cpu_flagged_nointr(cpu_flag_t cpu_flags)
```

```
1175 {
1176         return (cpu_flagged_active(cpu_flags) &&
1177             (cpu_flags & CPU_ENABLE) == 0);
1178 }

1180 int
1181 cpu_flagged_active(cpu_flag_t cpu_flags)
1182 {
1183         return (((cpu_flags & (CPU_POWEROFF | CPU_FAULTED | CPU_SPARE)) == 0) &&
1184             ((cpu_flags & (CPU_READY | CPU_OFFLINE)) == CPU_READY));
1185 }

1187 /*
1188  * Bring the indicated CPU online.
1189  */
1190 int
1191 cpu_online(cpu_t *cp)
1192 {
1193         int     error = 0;

1195         /*
1196          * Handle on-line request.
1197          *      This code must put the new CPU on the active list before
1198          *      starting it because it will not be paused, and will start
1199          *      using the active list immediately.  The real start occurs
1200          *      when the CPU_QUIESCED flag is turned off.
1201          */

1203         ASSERT(MUTEX_HELD(&cpu_lock));

1205         /*
1206          * Put all the cpus into a known safe place.
1207          * No mutexes can be entered while CPUs are paused.
1208          */
1209         error = mp_cpu_start(cp);       /* arch-dep hook */
1210         if (error == 0) {
1211                 pg_cpupart_in(cp, cp->cpu_part);
1212                 pause_cpus(NULL, NULL);
1000                 pause_cpus(NULL);
1213                 cpu_add_active_internal(cp);
1214                 if (cp->cpu_flags & CPU_FAULTED) {
1215                         cp->cpu_flags &= ~CPU_FAULTED;
1216                         mp_cpu_faulted_exit(cp);
1217                 }
1218                 cp->cpu_flags &= ~(CPU_QUIESCED | CPU_OFFLINE | CPU_FROZEN |
1219                     CPU_SPARE);
1220                 CPU_NEW_GENERATION(cp);
1221                 start_cpus();
1222                 cpu_stats_kstat_create(cp);
1223                 cpu_create_intrstat(cp);
1224                 lgrp_kstat_create(cp);
1225                 cpu_state_change_notify(cp->cpu_id, CPU_ON);
1226                 cpu_intr_enable(cp);    /* arch-dep hook */
1227                 cpu_state_change_notify(cp->cpu_id, CPU_INTR_ON);
1228                 cpu_set_state(cp);
1229                 cyclic_online(cp);
1230                 /*
1231                  * This has to be called only after cyclic_online(). This
1232                  * function uses cyclics.
1233                  */
1234                 callout_cpu_online(cp);
1235                 poke_cpu(cp->cpu_id);
1236         }

1238         return (error);
1239 }
```



```
1241 /*
1242  * Take the indicated CPU offline.
1243  */
1244 int
1245 cpu_offline(cpu_t *cp, int flags)
1246 {
1247         cpupart_t *pp;
1248         int     error = 0;
1249         cpu_t   *ncp;
1250         int     intr_enable;
1251         int     cyclic_off = 0;
1252         int     callout_off = 0;
1253         int     loop_count;
1254         int     no_quiesce = 0;
1255         int     (*bound_func)(struct cpu *, int);
1256         kthread_t *t;
1257         lpl_t   *cpu_lpl;
1258         proc_t  *p;
1259         int     lgrp_diff_lpl;
1260         boolean_t unbind_all_threads = (flags & CPU_FORCED) != 0;

1262         ASSERT(MUTEX_HELD(&cpu_lock));

1264         /*
1265          * If we're going from faulted or spare to offline, just
1266          * clear these flags and update CPU state.
1267          */
1268         if (cp->cpu_flags & (CPU_FAULTED | CPU_SPARE)) {
1269                 if (cp->cpu_flags & CPU_FAULTED) {
1270                         cp->cpu_flags &= ~CPU_FAULTED;
1271                         mp_cpu_faulted_exit(cp);
1272                 }
1273                 cp->cpu_flags &= ~CPU_SPARE;
1274                 cpu_set_state(cp);
1275                 return (0);
1276         }

1278         /*
1279          * Handle off-line request.
1280          */
1281         pp = cp->cpu_part;
1282         /*
1283          * Don't offline last online CPU in partition
1284          */
1285         if (ncpus_online <= 1 || pp->cp_ncpus <= 1 || cpu_intr_count(cp) < 2)
1286                 return (EBUSY);
1287         /*
1288          * Unbind all soft-bound threads bound to our CPU and hard bound threads
1289          * if we were asked to.
1290          */
1291         error = cpu_unbind(cp->cpu_id, unbind_all_threads);
1292         if (error != 0)
1293                 return (error);
1294         /*
1295          * We shouldn't be bound to this CPU ourselves.
1296          */
1297         if (curthread->t_bound_cpu == cp)
1298                 return (EBUSY);

1300         /*
1301          * Tell interested parties that this CPU is going offline.
1302          */
1303         CPU_NEW_GENERATION(cp);
1304         cpu_state_change_notify(cp->cpu_id, CPU_OFF);
```

```
1306         /*
1307          * Tell the PG subsystem that the CPU is leaving the partition
1308          */
1309         pg_cpupart_out(cp, pp);

1311         /*
1312          * Take the CPU out of interrupt participation so we won't find
1313          * bound kernel threads.  If the architecture cannot completely
1314          * shut off interrupts on the CPU, don't quiesce it, but don't
1315          * run anything but interrupt thread... this is indicated by
1316          * the CPU_OFFLINE flag being on but the CPU_QUIESCE flag being
1317          * off.
1318          */
1319         intr_enable = cp->cpu_flags & CPU_ENABLE;
1320         if (intr_enable)
1321                 no_quiesce = cpu_intr_disable(cp);

1323         /*
1324          * Record that we are aiming to offline this cpu.  This acts as
1325          * a barrier to further weak binding requests in thread_nomigrate
1326          * and also causes cpu_choose, disp_lowpri_cpu and setfrontdq to
1327          * lean away from this cpu.  Further strong bindings are already
1328          * avoided since we hold cpu_lock.  Since threads that are set
1329          * runnable around now and others coming off the target cpu are
1330          * directed away from the target, existing strong and weak bindings
1331          * (especially the latter) to the target cpu stand maximum chance of
1332          * being able to unbind during the short delay loop below (if other
1333          * unbound threads compete they may not see cpu in time to unbind
1334          * even if they would do so immediately.
1335          */
1336         cpu_inmotion = cp;
1337         membar_enter();

1339         /*
1340          * Check for kernel threads (strong or weak) bound to that CPU.
1341          * Strongly bound threads may not unbind, and we'll have to return
1342          * EBUSY.  Weakly bound threads should always disappear - we've
1343          * stopped more weak binding with cpu_inmotion and existing
1344          * bindings will drain imminently (they may not block).  Nonetheless
1345          * we will wait for a fixed period for all bound threads to disappear.
1346          * Inactive interrupt threads are OK (they'll be in TS_FREE
1347          * state).  If test finds some bound threads, wait a few ticks
1348          * to give short-lived threads (such as interrupts) chance to
1349          * complete.  Note that if no_quiesce is set, i.e. this cpu
1350          * is required to service interrupts, then we take the route
1351          * that permits interrupt threads to be active (or bypassed).
1352          */
1353         bound_func = no_quiesce ? disp_bound_threads : disp_bound_anythreads;

1355 again:  for (loop_count = 0; (*bound_func)(cp, 0); loop_count++) {
1356                 if (loop_count >= 5) {
1357                         error = EBUSY;  /* some threads still bound */
1358                         break;
1359                 }

1361                 /*
1362                  * If some threads were assigned, give them
1363                  * a chance to complete or move.
1364                  *
1365                  * This assumes that the clock_thread is not bound
1366                  * to any CPU, because the clock_thread is needed to
1367                  * do the delay(hz/100).
1368                  *
1369                  * Note: we still hold the cpu_lock while waiting for
1370                  * the next clock tick.  This is OK since it isn't
1371                  * needed for anything else except processor_bind(2),
```



```
1372                  * and system initialization.  If we drop the lock,
1373                  * we would risk another p_online disabling the last
1374                  * processor.
1375                  */
1376                 delay(hz/100);
1377         }

1379         if (error == 0 && callout_off == 0) {
1380                 callout_cpu_offline(cp);
1381                 callout_off = 1;
1382         }

1384         if (error == 0 && cyclic_off == 0) {
1385                 if (!cyclic_offline(cp)) {
1386                         /*
1387                          * We must have bound cyclics...
1388                          */
1389                         error = EBUSY;
1390                         goto out;
1391                 }
1392                 cyclic_off = 1;
1393         }

1395         /*
1396          * Call mp_cpu_stop() to perform any special operations
1397          * needed for this machine architecture to offline a CPU.
1398          */
1399         if (error == 0)
1400                 error = mp_cpu_stop(cp);        /* arch-dep hook */

1402         /*
1403          * If that all worked, take the CPU offline and decrement
1404          * ncpus_online.
1405          */
1406         if (error == 0) {
1407                 /*
1408                  * Put all the cpus into a known safe place.
1409                  * No mutexes can be entered while CPUs are paused.
1410                  */
1411                 pause_cpus(cp, NULL);
1199                 pause_cpus(cp);
1412                 /*
1413                  * Repeat the operation, if necessary, to make sure that
1414                  * all outstanding low-level interrupts run to completion
1415                  * before we set the CPU_QUIESCED flag.  It's also possible
1416                  * that a thread has weak bound to the cpu despite our raising
1417                  * cpu_inmotion above since it may have loaded that
1418                  * value before the barrier became visible (this would have
1419                  * to be the thread that was on the target cpu at the time
1420                  * we raised the barrier).
1421                  */
1422                 if ((!no_quiesce && cp->cpu_intr_actv != 0) ||
1423                     (*bound_func)(cp, 1)) {
1424                         start_cpus();
1425                         (void) mp_cpu_start(cp);
1426                         goto again;
1427                 }
1428                 ncp = cp->cpu_next_part;
1429                 cpu_lpl = cp->cpu_lpl;
1430                 ASSERT(cpu_lpl != NULL);

1432                 /*
1433                  * Remove the CPU from the list of active CPUs.
1434                  */
1435                 cpu_remove_active(cp);
```

```
1437                 /*
1438                  * Walk the active process list and look for threads
1439                  * whose home lgroup needs to be updated, or
1440                  * the last CPU they run on is the one being offlined now.
1441                  */

1443                 ASSERT(curthread->t_cpu != cp);
1444                 for (p = practive; p != NULL; p = p->p_next) {

1446                         t = p->p_tlist;

1448                         if (t == NULL)
1449                                 continue;

1451                         lgrp_diff_lpl = 0;

1453                         do {
1454                                 ASSERT(t->t_lpl != NULL);
1455                                 /*
1456                                  * Taking last CPU in lpl offline
1457                                  * Rehome thread if it is in this lpl
1458                                  * Otherwise, update the count of how many
1459                                  * threads are in this CPU's lgroup but have
1460                                  * a different lpl.
1461                                  */

1463                                 if (cpu_lpl->lpl_ncpu == 0) {
1464                                         if (t->t_lpl == cpu_lpl)
1465                                                 lgrp_move_thread(t,
1466                                                     lgrp_choose(t,
1467                                                     t->t_cpupart), 0);
1468                                         else if (t->t_lpl->lpl_lgrpid ==
1469                                             cpu_lpl->lpl_lgrpid)
1470                                                 lgrp_diff_lpl++;
1471                                 }
1472                                 ASSERT(t->t_lpl->lpl_ncpu > 0);

1474                                 /*
1475                                  * Update CPU last ran on if it was this CPU
1476                                  */
1477                                 if (t->t_cpu == cp && t->t_bound_cpu != cp)
1478                                         t->t_cpu = disp_lowpri_cpu(ncp,
1479                                             t->t_lpl, t->t_pri, NULL);
1480                                 ASSERT(t->t_cpu != cp || t->t_bound_cpu == cp ||
1481                                     t->t_weakbound_cpu == cp);

1483                                 t = t->t_forw;
1484                         } while (t != p->p_tlist);

1486                         /*
1487                          * Didn't find any threads in the same lgroup as this
1488                          * CPU with a different lpl, so remove the lgroup from
1489                          * the process lgroup bitmask.
1490                          */

1492                         if (lgrp_diff_lpl == 0)
1493                                 klgrpset_del(p->p_lgrpset, cpu_lpl->lpl_lgrpid);
1494                 }

1496                 /*
1497                  * Walk thread list looking for threads that need to be
1498                  * rehomed, since there are some threads that are not in
1499                  * their process's p_tlist.
1500                  */

1502                 t = curthread;
```



```
1503                 do {
1504                         ASSERT(t != NULL && t->t_lpl != NULL);

1506                         /*
1507                          * Rehome threads with same lpl as this CPU when this
1508                          * is the last CPU in the lpl.
1509                          */

1511                         if ((cpu_lpl->lpl_ncpu == 0) && (t->t_lpl == cpu_lpl))
1512                                 lgrp_move_thread(t,
1513                                     lgrp_choose(t, t->t_cpupart), 1);

1515                         ASSERT(t->t_lpl->lpl_ncpu > 0);

1517                         /*
1518                          * Update CPU last ran on if it was this CPU
1519                          */

1521                         if (t->t_cpu == cp && t->t_bound_cpu != cp) {
1522                                 t->t_cpu = disp_lowpri_cpu(ncp,
1523                                     t->t_lpl, t->t_pri, NULL);
1524                         }
1525                         ASSERT(t->t_cpu != cp || t->t_bound_cpu == cp ||
1526                             t->t_weakbound_cpu == cp);
1527                         t = t->t_next;

1529                 } while (t != curthread);
1530                 ASSERT((cp->cpu_flags & (CPU_FAULTED | CPU_SPARE)) == 0);
1531                 cp->cpu_flags |= CPU_OFFLINE;
1532                 disp_cpu_inactive(cp);
1533                 if (!no_quiesce)
1534                         cp->cpu_flags |= CPU_QUIESCED;
1535                 ncpus_online--;
1536                 cpu_set_state(cp);
1537                 cpu_inmotion = NULL;
1538                 start_cpus();
1539                 cpu_stats_kstat_destroy(cp);
1540                 cpu_delete_intrstat(cp);
1541                 lgrp_kstat_destroy(cp);
1542         }

1544 out:
1545         cpu_inmotion = NULL;

1547         /*
1548          * If we failed, re-enable interrupts.
1549          * Do this even if cpu_intr_disable returned an error, because
1550          * it may have partially disabled interrupts.
1551          */
1552         if (error && intr_enable)
1553                 cpu_intr_enable(cp);

1555         /*
1556          * If we failed, but managed to offline the cyclic subsystem on this
1557          * CPU, bring it back online.
1558          */
1559         if (error && cyclic_off)
1560                 cyclic_online(cp);

1562         /*
1563          * If we failed, but managed to offline callouts on this CPU,
1564          * bring it back online.
1565          */
1566         if (error && callout_off)
1567                 callout_cpu_online(cp);
```

```
1569         /*
1570          * If we failed, tell the PG subsystem that the CPU is back
1571          */
1572         pg_cpupart_in(cp, pp);

1574         /*
1575          * If we failed, we need to notify everyone that this CPU is back on.
1576          */
1577         if (error != 0) {
1578                 CPU_NEW_GENERATION(cp);
1579                 cpu_state_change_notify(cp->cpu_id, CPU_ON);
1580                 cpu_state_change_notify(cp->cpu_id, CPU_INTR_ON);
1581         }

1583         return (error);
1584 }
______unchanged_portion_omitted_

1731 /*
1732  * Insert a CPU into the list of available CPUs.
1733  */
1734 void
1735 cpu_add_unit(cpu_t *cp)
1736 {
1737         int seqid;

1739         ASSERT(MUTEX_HELD(&cpu_lock));
1740         ASSERT(cpu_list != NULL);       /* list started in cpu_list_init */

1742         lgrp_config(LGRP_CONFIG_CPU_ADD, (uintptr_t)cp, 0);

1744         /*
1745          * Note: most users of the cpu_list will grab the
1746          * cpu_lock to insure that it isn't modified.  However,
1747          * certain users can't or won't do that.  To allow this
1748          * we pause the other cpus.  Users who walk the list
1749          * without cpu_lock, must disable kernel preemption
1750          * to insure that the list isn't modified underneath
1751          * them.  Also, any cached pointers to cpu structures
1752          * must be revalidated by checking to see if the
1753          * cpu_next pointer points to itself.  This check must
1754          * be done with the cpu_lock held or kernel preemption
1755          * disabled.  This check relies upon the fact that
1756          * old cpu structures are not free'ed or cleared after
1757          * then are removed from the cpu_list.
1758          *
1759          * Note that the clock code walks the cpu list dereferencing
1760          * the cpu_part pointer, so we need to initialize it before
1761          * adding the cpu to the list.
1762          */
1763         cp->cpu_part = &cp_default;
1764         (void) pause_cpus(NULL, NULL);
1552         (void) pause_cpus(NULL);
1765         cp->cpu_next = cpu_list;
1766         cp->cpu_prev = cpu_list->cpu_prev;
1767         cpu_list->cpu_prev->cpu_next = cp;
1768         cpu_list->cpu_prev = cp;
1769         start_cpus();

1771         for (seqid = 0; CPU_IN_SET(cpu_seqid_inuse, seqid); seqid++)
1772                 continue;
1773         CPUSET_ADD(cpu_seqid_inuse, seqid);
1774         cp->cpu_seqid = seqid;

1776         if (seqid > max_cpu_seqid_ever)
1777                 max_cpu_seqid_ever = seqid;
```



```
1779         ASSERT(ncpus < max_ncpus);
1780         ncpus++;
1781         cp->cpu_cache_offset = KMEM_CPU_CACHE_OFFSET(cp->cpu_seqid);
1782         cpu[cp->cpu_id] = cp;
1783         CPUSET_ADD(cpu_available, cp->cpu_id);
1784         cpu_seq[cp->cpu_seqid] = cp;

1786         /*
1787          * allocate a pause thread for this CPU.
1788          */
1789         cpu_pause_alloc(cp);

1791         /*
1792          * So that new CPUs won't have NULL prev_onln and next_onln pointers,
1793          * link them into a list of just that CPU.
1794          * This is so that disp_lowpri_cpu will work for thread_create in
1795          * pause_cpus() when called from the startup thread in a new CPU.
1796          */
1797         cp->cpu_next_onln = cp;
1798         cp->cpu_prev_onln = cp;
1799         cpu_info_kstat_create(cp);
1800         cp->cpu_next_part = cp;
1801         cp->cpu_prev_part = cp;

1803         init_cpu_mstate(cp, CMS_SYSTEM);

1805         pool_pset_mod = gethrtime();
1806 }

1808 /*
1809  * Do the opposite of cpu_add_unit().
1810  */
1811 void
1812 cpu_del_unit(int cpuid)
1813 {
1814         struct cpu      *cp, *cpnext;

1816         ASSERT(MUTEX_HELD(&cpu_lock));
1817         cp = cpu[cpuid];
1818         ASSERT(cp != NULL);

1820         ASSERT(cp->cpu_next_onln == cp);
1821         ASSERT(cp->cpu_prev_onln == cp);
1822         ASSERT(cp->cpu_next_part == cp);
1823         ASSERT(cp->cpu_prev_part == cp);

1825         /*
1826          * Tear down the CPU's physical ID cache, and update any
1827          * processor groups
1828          */
1829         pg_cpu_fini(cp, NULL);
1830         pghw_physid_destroy(cp);

1832         /*
1833          * Destroy kstat stuff.
1834          */
1835         cpu_info_kstat_destroy(cp);
1836         term_cpu_mstate(cp);
1837         /*
1838          * Free up pause thread.
1839          */
1840         cpu_pause_free(cp);
1841         CPUSET_DEL(cpu_available, cp->cpu_id);
1842         cpu[cp->cpu_id] = NULL;
1843         cpu_seq[cp->cpu_seqid] = NULL;
```

```
1845         /*
1846          * The clock thread and mutex_vector_enter cannot hold the
1847          * cpu_lock while traversing the cpu list, therefore we pause
1848          * all other threads by pausing the other cpus. These, and any
1849          * other routines holding cpu pointers while possibly sleeping
1850          * must be sure to call kpreempt_disable before processing the
1851          * list and be sure to check that the cpu has not been deleted
1852          * after any sleeps (check cp->cpu_next != NULL). We guarantee
1853          * to keep the deleted cpu structure around.
1854          *
1855          * Note that this MUST be done AFTER cpu_available
1856          * has been updated so that we don't waste time
1857          * trying to pause the cpu we're trying to delete.
1858          */
1859         (void) pause_cpus(NULL, NULL);
1647         (void) pause_cpus(NULL);

1861         cpnext = cp->cpu_next;
1862         cp->cpu_prev->cpu_next = cp->cpu_next;
1863         cp->cpu_next->cpu_prev = cp->cpu_prev;
1864         if (cp == cpu_list)
1865                 cpu_list = cpnext;

1867         /*
1868          * Signals that the cpu has been deleted (see above).
1869          */
1870         cp->cpu_next = NULL;
1871         cp->cpu_prev = NULL;

1873         start_cpus();

1875         CPUSET_DEL(cpu_seqid_inuse, cp->cpu_seqid);
1876         ncpus--;
1877         lgrp_config(LGRP_CONFIG_CPU_DEL, (uintptr_t)cp, 0);

1879         pool_pset_mod = gethrtime();
1880 }
```

_____***unchanged_portion_omitted_***

```
1922 /*
1923  * Add a CPU to the list of active CPUs.
1924  *      This is called from machine-dependent layers when a new CPU is started.
1925  */
1926 void
1927 cpu_add_active(cpu_t *cp)
1928 {
1929         pg_cpupart_in(cp, cp->cpu_part);

1931         pause_cpus(NULL, NULL);
1719         pause_cpus(NULL);
1932         cpu_add_active_internal(cp);
1933         start_cpus();

1935         cpu_stats_kstat_create(cp);
1936         cpu_create_intrstat(cp);
1937         lgrp_kstat_create(cp);
1938         cpu_state_change_notify(cp->cpu_id, CPU_INIT);
1939 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   30588 Tue Nov  4 16:25:25 2014
new/usr/src/uts/common/os/cpu_event.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_

 368 static void
 369 cpu_idle_insert_callback(cpu_idle_cb_impl_t *cip)
 370 {
 371         int unlock = 0, unpause = 0;
 372         int i, cnt_new = 0, cnt_old = 0;
 373         char *buf_new = NULL, *buf_old = NULL;

 375         ASSERT(MUTEX_HELD(&cpu_idle_cb_lock));

 377         /*
 378          * Expand array if it's full.
 379          * Memory must be allocated out of pause/start_cpus() scope because
 380          * kmem_zalloc() can't be called with KM_SLEEP flag within that scope.
 381          */
 382         if (cpu_idle_cb_curr == cpu_idle_cb_max) {
 383                 cnt_new = cpu_idle_cb_max + CPU_IDLE_ARRAY_CAPACITY_INC;
 384                 buf_new = (char *)kmem_zalloc(cnt_new *
 385                     sizeof (cpu_idle_cb_item_t), KM_SLEEP);
 386         }

 388         /* Try to acquire cpu_lock if not held yet. */
 389         if (!MUTEX_HELD(&cpu_lock)) {
 390                 mutex_enter(&cpu_lock);
 391                 unlock = 1;
 392         }
 393         /*
 394          * Pause all other CPUs (and let them run pause thread).
 395          * It's guaranteed that no other threads will access cpu_idle_cb_array
 396          * after pause_cpus().
 397          */
 398         if (!cpus_paused()) {
 399                 pause_cpus(NULL, NULL);
 399                 pause_cpus(NULL);
 400                 unpause = 1;
 401         }

 403         /* Copy content to new buffer if needed. */
 404         if (buf_new != NULL) {
 405                 buf_old = (char *)cpu_idle_cb_array;
 406                 cnt_old = cpu_idle_cb_max;
 407                 if (buf_old != NULL) {
 408                         ASSERT(cnt_old != 0);
 409                         bcopy(cpu_idle_cb_array, buf_new,
 410                             sizeof (cpu_idle_cb_item_t) * cnt_old);
 411                 }
 412                 cpu_idle_cb_array = (cpu_idle_cb_item_t *)buf_new;
 413                 cpu_idle_cb_max = cnt_new;
 414         }

 416         /* Insert into array according to priority. */
 417         ASSERT(cpu_idle_cb_curr < cpu_idle_cb_max);
 418         for (i = cpu_idle_cb_curr; i > 0; i--) {
 419                 if (cpu_idle_cb_array[i - 1].impl->priority >= cip->priority) {
 420                         break;
 421                 }
 422                 cpu_idle_cb_array[i] = cpu_idle_cb_array[i - 1];
 423         }
 424         cpu_idle_cb_array[i].arg = cip->argument;
 425         cpu_idle_cb_array[i].enter = cip->callback->idle_enter;
```



```
 426         cpu_idle_cb_array[i].exit = cip->callback->idle_exit;
 427         cpu_idle_cb_array[i].impl = cip;
 428         cpu_idle_cb_curr++;

 430         /* Resume other CPUs from paused state if needed. */
 431         if (unpause) {
 432                 start_cpus();
 433         }
 434         if (unlock) {
 435                 mutex_exit(&cpu_lock);
 436         }

 438         /* Free old resource if needed. */
 439         if (buf_old != NULL) {
 440                 ASSERT(cnt_old != 0);
 441                 kmem_free(buf_old, cnt_old * sizeof (cpu_idle_cb_item_t));
 442         }
 443 }

 445 static void
 446 cpu_idle_remove_callback(cpu_idle_cb_impl_t *cip)
 447 {
 448         int i, found = 0;
 449         int unlock = 0, unpause = 0;
 450         cpu_idle_cb_state_t *sp;

 452         ASSERT(MUTEX_HELD(&cpu_idle_cb_lock));

 454         /* Try to acquire cpu_lock if not held yet. */
 455         if (!MUTEX_HELD(&cpu_lock)) {
 456                 mutex_enter(&cpu_lock);
 457                 unlock = 1;
 458         }
 459         /*
 460          * Pause all other CPUs.
 461          * It's guaranteed that no other threads will access cpu_idle_cb_array
 462          * after pause_cpus().
 463          */
 464         if (!cpus_paused()) {
 465                 pause_cpus(NULL, NULL);
 465                 pause_cpus(NULL);
 466                 unpause = 1;
 467         }

 469         /* Remove cip from array. */
 470         for (i = 0; i < cpu_idle_cb_curr; i++) {
 471                 if (found == 0) {
 472                         if (cpu_idle_cb_array[i].impl == cip) {
 473                                 found = 1;
 474                         }
 475                 } else {
 476                         cpu_idle_cb_array[i - 1] = cpu_idle_cb_array[i];
 477                 }
 478         }
 479         ASSERT(found != 0);
 480         cpu_idle_cb_curr--;

 482         /*
 483          * Reset property ready flag for all CPUs if no registered callback
 484          * left because cpu_idle_enter/exit will stop updating property if
 485          * there's no callback registered.
 486          */
 487         if (cpu_idle_cb_curr == 0) {
 488                 for (sp = cpu_idle_cb_state, i = 0; i < max_ncpus; i++, sp++) {
 489                         sp->v.ready = B_FALSE;
 490                 }
```

```
 491         }

 493         /* Resume other CPUs from paused state if needed. */
 494         if (unpause) {
 495                 start_cpus();
 496         }
 497         if (unlock) {
 498                 mutex_exit(&cpu_lock);
 499         }
 500 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   21527 Tue Nov  4 16:25:26 2014
new/usr/src/uts/common/os/cpu_pm.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_

 172 int
 173 cpupm_set_policy(cpupm_policy_t new_policy)
 174 {
 175         static int      gov_init = 0;
 176         int             result = 0;

 178         mutex_enter(&cpu_lock);
 179         if (new_policy == cpupm_policy) {
 180                 mutex_exit(&cpu_lock);
 181                 return (result);
 182         }

 184         /*
 185          * Pausing CPUs causes a high priority thread to be scheduled
 186          * on all other CPUs (besides the current one). This locks out
 187          * other CPUs from making CPUPM state transitions.
 188          */
 189         switch (new_policy) {
 190         case CPUPM_POLICY_DISABLED:
 191                 pause_cpus(NULL, NULL);
 191                 pause_cpus(NULL);
 192                 cpupm_policy = CPUPM_POLICY_DISABLED;
 193                 start_cpus();

 195                 result = cmt_pad_disable(PGHW_POW_ACTIVE);

 197                 /*
 198                  * Once PAD has been enabled, it should always be possible
 199                  * to disable it.
 200                  */
 201                 ASSERT(result == 0);

 203                 /*
 204                  * Bring all the active power domains to the maximum
 205                  * performance state.
 206                  */
 207                 cpupm_state_change_global(CPUPM_DTYPE_ACTIVE,
 208                     CPUPM_STATE_MAX_PERF);

 210                 break;
 211         case CPUPM_POLICY_ELASTIC:

 213                 result = cmt_pad_enable(PGHW_POW_ACTIVE);
 214                 if (result < 0) {
 215                         /*
 216                          * Failed to enable PAD across the active power
 217                          * domains, which may well be because none were
 218                          * enumerated.
 219                          */
 220                         break;
 221                 }

 223                 /*
 224                  * Initialize the governor parameters the first time through.
 225                  */
 226                 if (gov_init == 0) {
 227                         cpupm_governor_initialize();
 228                         gov_init = 1;
 229                 }
```



```
 231                 pause_cpus(NULL, NULL);
 231                 pause_cpus(NULL);
 232                 cpupm_policy = CPUPM_POLICY_ELASTIC;
 233                 start_cpus();

 235                 break;
 236         default:
 237                 cmn_err(CE_WARN, "Attempt to set unknown CPUPM policy %d\n",
 238                     new_policy);
 239                 ASSERT(0);
 240                 break;
 241         }
 242         mutex_exit(&cpu_lock);

 244         return (result);
 245 }
______unchanged_portion_omitted_
```

```
**********************************************************
  119448 Tue Nov  4 16:25:26 2014
new/usr/src/uts/common/os/lgrp.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_

1225 /*
1226  * Called to indicate that the lgrp with platform handle "hand" now
1227  * contains the memory identified by "mnode".
1228  *
1229  * LOCKING for this routine is a bit tricky. Usually it is called without
1230  * cpu_lock and it must must grab cpu_lock here to prevent racing with other
1231  * callers. During DR of the board containing the caged memory it may be called
1232  * with cpu_lock already held and CPUs paused.
1233  *
1234  * If the insertion is part of the DR copy-rename and the inserted mnode (and
1235  * only this mnode) is already present in the lgrp_root->lgrp_mnodes set, we are
1236  * dealing with the special case of DR copy-rename described in
1237  * lgrp_mem_rename().
1238  */
1239 void
1240 lgrp_mem_init(int mnode, lgrp_handle_t hand, boolean_t is_copy_rename)
1241 {
1242         klgrpset_t      changed;
1243         int             count;
1244         int             i;
1245         lgrp_t          *my_lgrp;
1246         lgrp_id_t       lgrpid;
1247         mnodeset_t      mnodes_mask = ((mnodeset_t)1 << mnode);
1248         boolean_t       drop_lock = B_FALSE;
1249         boolean_t       need_synch = B_FALSE;

1251         /*
1252          * Grab CPU lock (if we haven't already)
1253          */
1254         if (!MUTEX_HELD(&cpu_lock)) {
1255                 mutex_enter(&cpu_lock);
1256                 drop_lock = B_TRUE;
1257         }

1259         /*
1260          * This routine may be called from a context where we already
1261          * hold cpu_lock, and have already paused cpus.
1262          */
1263         if (!cpus_paused())
1264                 need_synch = B_TRUE;

1266         /*
1267          * Check if this mnode is already configured and return immediately if
1268          * it is.
1269          *
1270          * NOTE: in special case of copy-rename of the only remaining mnode,
1271          * lgrp_mem_fini() refuses to remove the last mnode from the root, so we
1272          * recognize this case and continue as usual, but skip the update to
1273          * the lgrp_mnodes and the lgrp_nmnodes. This restores the inconsistency
1274          * in topology, temporarily introduced by lgrp_mem_fini().
1275          */
1276         if (! (is_copy_rename && (lgrp_root->lgrp_mnodes == mnodes_mask)) &&
1277             lgrp_root->lgrp_mnodes & mnodes_mask) {
1278                 if (drop_lock)
1279                         mutex_exit(&cpu_lock);
1280                 return;
1281         }

1283         /*
```



```
1284          * Update lgroup topology with new memory resources, keeping track of
1285          * which lgroups change
1286          */
1287         count = 0;
1288         klgrpset_clear(changed);
1289         my_lgrp = lgrp_hand_to_lgrp(hand);
1290         if (my_lgrp == NULL) {
1291                 /* new lgrp */
1292                 my_lgrp = lgrp_create();
1293                 lgrpid = my_lgrp->lgrp_id;
1294                 my_lgrp->lgrp_plathand = hand;
1295                 my_lgrp->lgrp_latency = lgrp_plat_latency(hand, hand);
1296                 klgrpset_add(my_lgrp->lgrp_leaves, lgrpid);
1297                 klgrpset_add(my_lgrp->lgrp_set[LGRP_RSRC_MEM], lgrpid);

1299                 if (need_synch)
1300                         pause_cpus(NULL, NULL);
1300                         pause_cpus(NULL);
1301                 count = lgrp_leaf_add(my_lgrp, lgrp_table, lgrp_alloc_max + 1,
1302                     &changed);
1303                 if (need_synch)
1304                         start_cpus();
1305         } else if (my_lgrp->lgrp_latency == 0 && lgrp_plat_latency(hand, hand)
1306             > 0) {
1307                 /*
1308                  * Leaf lgroup was created, but latency wasn't available
1309                  * then.  So, set latency for it and fill in rest of lgroup
1310                  * topology  now that we know how far it is from other leaf
1311                  * lgroups.
1312                  */
1313                 klgrpset_clear(changed);
1314                 lgrpid = my_lgrp->lgrp_id;
1315                 if (!klgrpset_ismember(my_lgrp->lgrp_set[LGRP_RSRC_MEM],
1316                     lgrpid))
1317                         klgrpset_add(my_lgrp->lgrp_set[LGRP_RSRC_MEM], lgrpid);
1318                 if (need_synch)
1319                         pause_cpus(NULL, NULL);
1319                         pause_cpus(NULL);
1320                 count = lgrp_leaf_add(my_lgrp, lgrp_table, lgrp_alloc_max + 1,
1321                     &changed);
1322                 if (need_synch)
1323                         start_cpus();
1324         } else if (!klgrpset_ismember(my_lgrp->lgrp_set[LGRP_RSRC_MEM],
1325             my_lgrp->lgrp_id)) {
1326                 /*
1327                  * Add new lgroup memory resource to existing lgroup
1328                  */
1329                 lgrpid = my_lgrp->lgrp_id;
1330                 klgrpset_add(my_lgrp->lgrp_set[LGRP_RSRC_MEM], lgrpid);
1331                 klgrpset_add(changed, lgrpid);
1332                 count++;
1333                 for (i = 0; i <= lgrp_alloc_max; i++) {
1334                         lgrp_t          *lgrp;

1336                         lgrp = lgrp_table[i];
1337                         if (!LGRP_EXISTS(lgrp) ||
1338                             !lgrp_rsets_member(lgrp->lgrp_set, lgrpid))
1339                                 continue;

1341                         klgrpset_add(lgrp->lgrp_set[LGRP_RSRC_MEM], lgrpid);
1342                         klgrpset_add(changed, lgrp->lgrp_id);
1343                         count++;
1344                 }
1345         }

1347         /*
```

```
1348          * Add memory node to lgroup and remove lgroup from ones that need
1349          * to be updated
1350          */
1351         if (!(my_lgrp->lgrp_mnodes & mnodes_mask)) {
1352                 my_lgrp->lgrp_mnodes |= mnodes_mask;
1353                 my_lgrp->lgrp_nmnodes++;
1354         }
1355         klgrpset_del(changed, lgrpid);

1357         /*
1358          * Update memory node information for all lgroups that changed and
1359          * contain new memory node as a resource
1360          */
1361         if (count)
1362                 (void) lgrp_mnode_update(changed, NULL);

1364         if (drop_lock)
1365                 mutex_exit(&cpu_lock);
1366 }

1368 /*
1369  * Called to indicate that the lgroup associated with the platform
1370  * handle "hand" no longer contains given memory node
1371  *
1372  * LOCKING for this routine is a bit tricky. Usually it is called without
1373  * cpu_lock and it must must grab cpu_lock here to prevent racing with other
1374  * callers. During DR of the board containing the caged memory it may be called
1375  * with cpu_lock already held and CPUs paused.
1376  *
1377  * If the deletion is part of the DR copy-rename and the deleted mnode is the
1378  * only one present in the lgrp_root->lgrp_mnodes, all the topology is updated,
1379  * but lgrp_root->lgrp_mnodes is left intact. Later, lgrp_mem_init() will insert
1380  * the same mnode back into the topology. See lgrp_mem_rename() and
1381  * lgrp_mem_init() for additional details.
1382  */
1383 void
1384 lgrp_mem_fini(int mnode, lgrp_handle_t hand, boolean_t is_copy_rename)
1385 {
1386         klgrpset_t      changed;
1387         int             count;
1388         int             i;
1389         lgrp_t          *my_lgrp;
1390         lgrp_id_t       lgrpid;
1391         mnodeset_t      mnodes_mask;
1392         boolean_t       drop_lock = B_FALSE;
1393         boolean_t       need_synch = B_FALSE;

1395         /*
1396          * Grab CPU lock (if we haven't already)
1397          */
1398         if (!MUTEX_HELD(&cpu_lock)) {
1399                 mutex_enter(&cpu_lock);
1400                 drop_lock = B_TRUE;
1401         }

1403         /*
1404          * This routine may be called from a context where we already
1405          * hold cpu_lock and have already paused cpus.
1406          */
1407         if (!cpus_paused())
1408                 need_synch = B_TRUE;

1410         my_lgrp = lgrp_hand_to_lgrp(hand);

1412         /*
1413          * The lgrp *must* be pre-existing
```



```
1414          */
1415         ASSERT(my_lgrp != NULL);

1417         /*
1418          * Delete memory node from lgroups which contain it
1419          */
1420         mnodes_mask = ((mnodeset_t)1 << mnode);
1421         for (i = 0; i <= lgrp_alloc_max; i++) {
1422                 lgrp_t *lgrp = lgrp_table[i];
1423                 /*
1424                  * Skip any non-existent lgroups and any lgroups that don't
1425                  * contain leaf lgroup of memory as a memory resource
1426                  */
1427                 if (!LGRP_EXISTS(lgrp) ||
1428                     !(lgrp->lgrp_mnodes & mnodes_mask))
1429                         continue;

1431                 /*
1432                  * Avoid removing the last mnode from the root in the DR
1433                  * copy-rename case. See lgrp_mem_rename() for details.
1434                  */
1435                 if (is_copy_rename &&
1436                     (lgrp == lgrp_root) && (lgrp->lgrp_mnodes == mnodes_mask))
1437                         continue;

1439                 /*
1440                  * Remove memory node from lgroup.
1441                  */
1442                 lgrp->lgrp_mnodes &= ~mnodes_mask;
1443                 lgrp->lgrp_nmnodes--;
1444                 ASSERT(lgrp->lgrp_nmnodes >= 0);
1445         }
1446         ASSERT(lgrp_root->lgrp_nmnodes > 0);

1448         /*
1449          * Don't need to update lgroup topology if this lgroup still has memory.
1450          *
1451          * In the special case of DR copy-rename with the only mnode being
1452          * removed, the lgrp_mnodes for the root is always non-zero, but we
1453          * still need to update the lgroup topology.
1454          */
1455         if ((my_lgrp->lgrp_nmnodes > 0) &&
1456             !(is_copy_rename && (my_lgrp == lgrp_root) &&
1457             (my_lgrp->lgrp_mnodes == mnodes_mask))) {
1458                 if (drop_lock)
1459                         mutex_exit(&cpu_lock);
1460                 return;
1461         }

1463         /*
1464          * This lgroup does not contain any memory now
1465          */
1466         klgrpset_clear(my_lgrp->lgrp_set[LGRP_RSRC_MEM]);

1468         /*
1469          * Remove this lgroup from lgroup topology if it does not contain any
1470          * resources now
1471          */
1472         lgrpid = my_lgrp->lgrp_id;
1473         count = 0;
1474         klgrpset_clear(changed);
1475         if (lgrp_rsets_empty(my_lgrp->lgrp_set)) {
1476                 /*
1477                  * Delete lgroup when no more resources
1478                  */
1479                 if (need_synch)
```

```
1480                        pause_cpus(NULL, NULL);
1480                        pause_cpus(NULL);
1481                count = lgrp_leaf_delete(my_lgrp, lgrp_table,
1482                    lgrp_alloc_max + 1, &changed);
1483                ASSERT(count > 0);
1484                if (need_synch)
1485                        start_cpus();
1486        } else {
1487                /*
1488                 * Remove lgroup from memory resources of any lgroups that
1489                 * contain it as such
1490                 */
1491                for (i = 0; i <= lgrp_alloc_max; i++) {
1492                        lgrp_t          *lgrp;

1494                        lgrp = lgrp_table[i];
1495                        if (!LGRP_EXISTS(lgrp) ||
1496                            !klgrpset_ismember(lgrp->lgrp_set[LGRP_RSRC_MEM],
1497                            lgrpid))
1498                                continue;

1500                        klgrpset_del(lgrp->lgrp_set[LGRP_RSRC_MEM], lgrpid);
1501                }
1502        }
1503        if (drop_lock)
1504                mutex_exit(&cpu_lock);
1505 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   36945 Tue Nov  4 16:25:26 2014
new/usr/src/uts/common/os/lgrp_topo.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_


1448 /*
1449  * Update lgroup topology for any leaves that don't have their latency set
1450  *
1451  * This may happen on some machines when the lgroup platform support doesn't
1452  * know the latencies between nodes soon enough to provide it when the
1453  * resources are being added.  If the lgroup platform code needs to probe
1454  * memory to determine the latencies between nodes, it must wait until the
1455  * CPUs become active so at least one CPU in each node can probe memory in
1456  * each node.
1457  */
1458 int
1459 lgrp_topo_update(lgrp_t **lgrps, int lgrp_count, klgrpset_t *changed)
1460 {
1461         klgrpset_t      changes;
1462         int             count;
1463         int             i;
1464         lgrp_t          *lgrp;

1466         count = 0;
1467         if (changed)
1468                 klgrpset_clear(*changed);

1470         /*
1471          * For UMA machines, make sure that root lgroup contains all
1472          * resources.  The root lgrp should also name itself as its own leaf
1473          */
1474         if (nlgrps == 1) {
1475                 for (i = 0; i < LGRP_RSRC_COUNT; i++)
1476                         klgrpset_add(lgrp_root->lgrp_set[i],
1477                             lgrp_root->lgrp_id);
1478                 klgrpset_add(lgrp_root->lgrp_leaves, lgrp_root->lgrp_id);
1479                 return (0);
1480         }

1482         mutex_enter(&cpu_lock);
1483         pause_cpus(NULL, NULL);
1483         pause_cpus(NULL);

1485         /*
1486          * Look for any leaf lgroup without its latency set, finish adding it
1487          * to the lgroup topology assuming that it exists and has the root
1488          * lgroup as its parent, and update the memory nodes of all lgroups
1489          * that have it as a memory resource.
1490          */
1491         for (i = 0; i < lgrp_count; i++) {
1492                 lgrp = lgrps[i];

1494                 /*
1495                  * Skip non-existent and non-leaf lgroups and any lgroup
1496                  * with its latency set already
1497                  */
1498                 if (lgrp == NULL || lgrp->lgrp_id == LGRP_NONE ||
1499                     lgrp->lgrp_childcnt != 0 || lgrp->lgrp_latency != 0)
1500                         continue;

1502 #ifdef  DEBUG
1503                 if (lgrp_topo_debug > 1) {
1504                         prom_printf("\nlgrp_topo_update: updating lineage "
```



```
1505                             "of lgrp %d at 0x%p\n", lgrp->lgrp_id,
1506                             (void *)lgrp);
1507                 }
1508 #endif  /* DEBUG */

1510                 count += lgrp_leaf_add(lgrp, lgrps, lgrp_count, &changes);
1511                 if (changed)
1512                         klgrpset_or(*changed, changes);

1514                 if (!klgrpset_isempty(changes))
1515                         (void) lgrp_mnode_update(changes, NULL);

1517 #ifdef  DEBUG
1518                 if (lgrp_topo_debug > 1 && changed)
1519                         prom_printf("lgrp_topo_update: changed %d lgrps: "
1520                             "0x%llx\n",
1521                             count, (u_longlong_t)*changed);
1522 #endif  /* DEBUG */
1523         }

1525         if (lgrp_topo_levels < LGRP_TOPO_LEVELS && lgrp_topo_levels == 2) {
1526                 count += lgrp_topo_flatten(2, lgrps, lgrp_count, changed);
1527                 (void) lpl_topo_flatten(2);
1528         }

1530         start_cpus();
1531         mutex_exit(&cpu_lock);

1533         return (count);
1534 }
______unchanged_portion_omitted_
```

****
82758 Tue Nov 4 16:25:26 2014
new/usr/src/uts/common/os/mem_config.c
5285 pass in cpu_pause_func via pause_cpus
****
_____***unchanged_portion_omitted_***

```
3295 /*
3296  * Invalidate memseg pointers in cpu private vm data caches.
3297  */
3298 static void
3299 memseg_cpu_vm_flush()
3300 {
3301         cpu_t *cp;
3302         vm_cpu_data_t *vc;

3304         mutex_enter(&cpu_lock);
3305         pause_cpus(NULL, NULL);
3305         pause_cpus(NULL);

3307         cp = cpu_list;
3308         do {
3309                 vc = cp->cpu_vm_data;
3310                 vc->vc_pnum_memseg = NULL;
3311                 vc->vc_pnext_memseg = NULL;

3313         } while ((cp = cp->cpu_next) != cpu_list);

3315         start_cpus();
3316         mutex_exit(&cpu_lock);
3317 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   30035 Tue Nov  4 16:25:27 2014
new/usr/src/uts/common/sys/cpuvar.h
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_
```

```
 604 #define CPU_STATS(cp, stat)                                     \
 605         ((cp)->cpu_stats.stat)

 607 /*
 608  * Increment CPU generation value.
 609  * This macro should be called whenever CPU goes on-line or off-line.
 610  * Updates to cpu_generation should be protected by cpu_lock.
 611  */
 612 #define CPU_NEW_GENERATION(cp)  ((cp)->cpu_generation++)

 614 #endif /* _KERNEL || _KMEMUSER */

 616 /*
 617  * CPU support routines.
 618  */
 619 #if     defined(_KERNEL) && defined(__STDC__)   /* not for genassym.c */

 621 struct zone;

 623 void    cpu_list_init(cpu_t *);
 624 void    cpu_add_unit(cpu_t *);
 625 void    cpu_del_unit(int cpuid);
 626 void    cpu_add_active(cpu_t *);
 627 void    cpu_kstat_init(cpu_t *);
 628 void    cpu_visibility_add(cpu_t *, struct zone *);
 629 void    cpu_visibility_remove(cpu_t *, struct zone *);
 630 void    cpu_visibility_configure(cpu_t *, struct zone *);
 631 void    cpu_visibility_unconfigure(cpu_t *, struct zone *);
 632 void    cpu_visibility_online(cpu_t *, struct zone *);
 633 void    cpu_visibility_offline(cpu_t *, struct zone *);
 634 void    cpu_create_intrstat(cpu_t *);
 635 void    cpu_delete_intrstat(cpu_t *);
 636 int     cpu_kstat_intrstat_update(kstat_t *, int);
 637 void    cpu_intr_swtch_enter(kthread_t *);
 638 void    cpu_intr_swtch_exit(kthread_t *);

 640 void    mbox_lock_init(void);    /* initialize cross-call locks */
 641 void    mbox_init(int cpun);     /* initialize cross-calls */
 642 void    poke_cpu(int cpun);      /* interrupt another CPU (to preempt) */

 644 /*
 645  * values for safe_list.  Pause state that CPUs are in.
 646  */
 647 #define PAUSE_IDLE      0               /* normal state */
 648 #define PAUSE_READY     1               /* paused thread ready to spl */
 649 #define PAUSE_WAIT      2               /* paused thread is spl-ed high */
 650 #define PAUSE_DIE       3               /* tell pause thread to leave */
 651 #define PAUSE_DEAD      4               /* pause thread has left */

 653 void    mach_cpu_pause(volatile char *);

 655 void    pause_cpus(cpu_t *off_cp, void *(*func)(void *));
 655 void    pause_cpus(cpu_t *off_cp);
 656 void    start_cpus(void);
 657 int     cpus_paused(void);

 659 void    cpu_pause_init(void);
 660 cpu_t   *cpu_get(processorid_t cpun);   /* get the CPU struct associated */
```



```
 662 int     cpu_online(cpu_t *cp);                  /* take cpu online */
 663 int     cpu_offline(cpu_t *cp, int flags);      /* take cpu offline */
 664 int     cpu_spare(cpu_t *cp, int flags);        /* take cpu to spare */
 665 int     cpu_faulted(cpu_t *cp, int flags);      /* take cpu to faulted */
 666 int     cpu_poweron(cpu_t *cp);         /* take powered-off cpu to offline */
 667 int     cpu_poweroff(cpu_t *cp);        /* take offline cpu to powered-off */

 669 cpu_t   *cpu_intr_next(cpu_t *cp);      /* get next online CPU taking intrs */
 670 int     cpu_intr_count(cpu_t *cp);      /* count # of CPUs handling intrs */
 671 int     cpu_intr_on(cpu_t *cp);         /* CPU taking I/O interrupts? */
 672 void    cpu_intr_enable(cpu_t *cp);     /* enable I/O interrupts */
 673 int     cpu_intr_disable(cpu_t *cp);    /* disable I/O interrupts */
 674 void    cpu_intr_alloc(cpu_t *cp, int n); /* allocate interrupt threads */

 676 /*
 677  * Routines for checking CPU states.
 678  */
 679 int     cpu_is_online(cpu_t *);         /* check if CPU is online */
 680 int     cpu_is_nointr(cpu_t *);         /* check if CPU can service intrs */
 681 int     cpu_is_active(cpu_t *);         /* check if CPU can run threads */
 682 int     cpu_is_offline(cpu_t *);        /* check if CPU is offline */
 683 int     cpu_is_poweredoff(cpu_t *);     /* check if CPU is powered off */

 685 int     cpu_flagged_online(cpu_flag_t); /* flags show CPU is online */
 686 int     cpu_flagged_nointr(cpu_flag_t); /* flags show CPU not handling intrs */
 687 int     cpu_flagged_active(cpu_flag_t); /* flags show CPU scheduling threads */
 688 int     cpu_flagged_offline(cpu_flag_t); /* flags show CPU is offline */
 689 int     cpu_flagged_poweredoff(cpu_flag_t); /* flags show CPU is powered off */

 691 /*
 692  * The processor_info(2) state of a CPU is a simplified representation suitable
 693  * for use by an application program.  Kernel subsystems should utilize the
 694  * internal per-CPU state as given by the cpu_flags member of the cpu structure,
 695  * as this information may include platform- or architecture-specific state
 696  * critical to a subsystem's disposition of a particular CPU.
 697  */
 698 void    cpu_set_state(cpu_t *);         /* record/timestamp current state */
 699 int     cpu_get_state(cpu_t *);         /* get current cpu state */
 700 const char *cpu_get_state_str(cpu_t *); /* get current cpu state as string */


 703 void    cpu_set_curr_clock(uint64_t);   /* indicate the current CPU's freq */
 704 void    cpu_set_supp_freqs(cpu_t *, const char *); /* set the CPU supported */
 705                                                 /* frequencies */

 707 int     cpu_configure(int);
 708 int     cpu_unconfigure(int);
 709 void    cpu_destroy_bound_threads(cpu_t *cp);

 711 extern int cpu_bind_thread(kthread_t *tp, processorid_t bind,
 712     processorid_t *obind, int *error);
 713 extern int cpu_unbind(processorid_t cpu_id, boolean_t force);
 714 extern void thread_affinity_set(kthread_t *t, int cpu_id);
 715 extern void thread_affinity_clear(kthread_t *t);
 716 extern void affinity_set(int cpu_id);
 717 extern void affinity_clear(void);
 718 extern void init_cpu_mstate(struct cpu *, int);
 719 extern void term_cpu_mstate(struct cpu *);
 720 extern void new_cpu_mstate(int, hrtime_t);
 721 extern void get_cpu_mstate(struct cpu *, hrtime_t *);
 722 extern void thread_nomigrate(void);
 723 extern void thread_allowmigrate(void);
 724 extern void weakbinding_stop(void);
 725 extern void weakbinding_start(void);

 727 /*
```

```
 728  * The following routines affect the CPUs participation in interrupt processing,
 729  * if that is applicable on the architecture.  This only affects interrupts
 730  * which aren't directed at the processor (not cross calls).
 731  *
 732  * cpu_disable_intr returns non-zero if interrupts were previously enabled.
 733  */
 734 int     cpu_disable_intr(struct cpu *cp); /* stop issuing interrupts to cpu */
 735 void    cpu_enable_intr(struct cpu *cp); /* start issuing interrupts to cpu */

 737 /*
 738  * The mutex cpu_lock protects cpu_flags for all CPUs, as well as the ncpus
 739  * and ncpus_online counts.
 740  */
 741 extern kmutex_t cpu_lock;       /* lock protecting CPU data */

 743 /*
 744  * CPU state change events
 745  *
 746  * Various subsystems need to know when CPUs change their state. They get this
 747  * information by registering  CPU state change callbacks using
 748  * register_cpu_setup_func(). Whenever any CPU changes its state, the callback
 749  * function is called. The callback function is passed three arguments:
 750  *
 751  *   Event, described by cpu_setup_t
 752  *   CPU ID
 753  *   Transparent pointer passed when registering the callback
 754  *
 755  * The callback function is called with cpu_lock held. The return value from the
 756  * callback function is usually ignored, except for CPU_CONFIG and CPU_UNCONFIG
 757  * events. For these two events, non-zero return value indicates a failure and
 758  * prevents successful completion of the operation.
 759  *
 760  * New events may be added in the future. Callback functions should ignore any
 761  * events that they do not understand.
 762  *
 763  * The following events provide notification callbacks:
 764  *
 765  *  CPU_INIT    A new CPU is started and added to the list of active CPUs
 766  *                This event is only used during boot
 767  *
 768  *  CPU_CONFIG  A newly inserted CPU is prepared for starting running code
 769  *                This event is called by DR code
 770  *
 771  *  CPU_UNCONFIG CPU has been powered off and needs cleanup
 772  *                This event is called by DR code
 773  *
 774  *  CPU_ON      CPU is enabled but does not run anything yet
 775  *
 776  *  CPU_INTR_ON CPU is enabled and has interrupts enabled
 777  *
 778  *  CPU_OFF     CPU is going offline but can still run threads
 779  *
 780  *  CPU_CPUPART_OUT     CPU is going to move out of its partition
 781  *
 782  *  CPU_CPUPART_IN      CPU is going to move to a new partition
 783  *
 784  *  CPU_SETUP   CPU is set up during boot and can run threads
 785  */
 786 typedef enum {
 787         CPU_INIT,
 788         CPU_CONFIG,
 789         CPU_UNCONFIG,
 790         CPU_ON,
 791         CPU_OFF,
 792         CPU_CPUPART_IN,
 793         CPU_CPUPART_OUT,
```



```
 794         CPU_SETUP,
 795         CPU_INTR_ON
 796 } cpu_setup_t;
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   18246 Tue Nov  4 16:25:27 2014
new/usr/src/uts/i86pc/i86hvm/io/xpv/xpv_support.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_

 429 /*
 430  * Top level routine to direct suspend/resume of a domain.
 431  */
 432 void
 433 xen_suspend_domain(void)
 434 {
 435         extern void rtcsync(void);
 436         extern void ec_resume(void);
 437         extern kmutex_t ec_lock;
 438         struct xen_add_to_physmap xatp;
 439         ulong_t flags;
 440         int err;

 442         cmn_err(CE_NOTE, "Domain suspending for save/migrate");

 444         SUSPEND_DEBUG("xen_suspend_domain\n");

 446         /*
 447          * We only want to suspend the PV devices, since the emulated devices
 448          * are suspended by saving the emulated device state.  The PV devices
 449          * are all children of the xpvd nexus device.  So we search the
 450          * device tree for the xpvd node to use as the root of the tree to
 451          * be suspended.
 452          */
 453         if (xpvd_dip == NULL)
 454                 ddi_walk_devs(ddi_root_node(), check_xpvd, NULL);

 456         /*
 457          * suspend interrupts and devices
 458          */
 459         if (xpvd_dip != NULL)
 460                 (void) xen_suspend_devices(ddi_get_child(xpvd_dip));
 461         else
 462                 cmn_err(CE_WARN, "No PV devices found to suspend");
 463         SUSPEND_DEBUG("xenbus_suspend\n");
 464         xenbus_suspend();

 466         mutex_enter(&cpu_lock);

 468         /*
 469          * Suspend on vcpu 0
 470          */
 471         thread_affinity_set(curthread, 0);
 472         kpreempt_disable();

 474         if (ncpus > 1)
 475                 pause_cpus(NULL, NULL);
 475                 pause_cpus(NULL);
 476         /*
 477          * We can grab the ec_lock as it's a spinlock with a high SPL. Hence
 478          * any holder would have dropped it to get through pause_cpus().
 479          */
 480         mutex_enter(&ec_lock);

 482         /*
 483          * From here on in, we can't take locks.
 484          */

 486         flags = intr_clear();
```



```
 488         SUSPEND_DEBUG("HYPERVISOR_suspend\n");
 489         /*
 490          * At this point we suspend and sometime later resume.
 491          * Note that this call may return with an indication of a cancelled
 492          * for now no matter ehat the return we do a full resume of all
 493          * suspended drivers, etc.
 494          */
 495         (void) HYPERVISOR_shutdown(SHUTDOWN_suspend);

 497         /*
 498          * Point HYPERVISOR_shared_info to the proper place.
 499          */
 500         xatp.domid = DOMID_SELF;
 501         xatp.idx = 0;
 502         xatp.space = XENMAPSPACE_shared_info;
 503         xatp.gpfn = xen_shared_info_frame;
 504         if ((err = HYPERVISOR_memory_op(XENMEM_add_to_physmap, &xatp)) != 0)
 505                 panic("Could not set shared_info page. error: %d", err);

 507         SUSPEND_DEBUG("gnttab_resume\n");
 508         gnttab_resume();

 510         SUSPEND_DEBUG("ec_resume\n");
 511         ec_resume();

 513         intr_restore(flags);

 515         if (ncpus > 1)
 516                 start_cpus();

 518         mutex_exit(&ec_lock);
 519         mutex_exit(&cpu_lock);

 521         /*
 522          * Now we can take locks again.
 523          */

 525         rtcsync();

 527         SUSPEND_DEBUG("xenbus_resume\n");
 528         xenbus_resume();
 529         SUSPEND_DEBUG("xen_resume_devices\n");
 530         if (xpvd_dip != NULL)
 531                 (void) xen_resume_devices(ddi_get_child(xpvd_dip), 0);

 533         thread_affinity_clear(curthread);
 534         kpreempt_enable();

 536         SUSPEND_DEBUG("finished xen_suspend_domain\n");

 538         cmn_err(CE_NOTE, "domain restore/migrate completed");
 539 }
______unchanged_portion_omitted_
```

```
**********************************************************
   23453 Tue Nov  4 16:25:27 2014
new/usr/src/uts/i86pc/io/dr/dr_quiesce.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_

 785 int
 786 dr_suspend(dr_sr_handle_t *srh)
 787 {
 788         dr_handle_t     *handle;
 789         int             force;
 790         int             dev_errs_idx;
 791         uint64_t        dev_errs[DR_MAX_ERR_INT];
 792         int             rc = DDI_SUCCESS;

 794         handle = srh->sr_dr_handlep;

 796         force = dr_cmd_flags(handle) & SBD_FLAG_FORCE;

 798         prom_printf("\nDR: suspending user threads...\n");
 799         srh->sr_suspend_state = DR_SRSTATE_USER;
 800         if (((rc = dr_stop_user_threads(srh)) != DDI_SUCCESS) &&
 801             dr_check_user_stop_result) {
 802                 dr_resume(srh);
 803                 return (rc);
 804         }

 806         if (!force) {
 807                 struct dr_ref drc = {0};

 809                 prom_printf("\nDR: checking devices...\n");
 810                 dev_errs_idx = 0;

 812                 drc.arr = dev_errs;
 813                 drc.idx = &dev_errs_idx;
 814                 drc.len = DR_MAX_ERR_INT;

 816                 /*
 817                  * Since the root node can never go away, it
 818                  * doesn't have to be held.
 819                  */
 820                 ddi_walk_devs(ddi_root_node(), dr_check_unsafe_major, &drc);
 821                 if (dev_errs_idx) {
 822                         handle->h_err = drerr_int(ESBD_UNSAFE, dev_errs,
 823                             dev_errs_idx, 1);
 824                         dr_resume(srh);
 825                         return (DDI_FAILURE);
 826                 }
 827                 PR_QR("done\n");
 828         } else {
 829                 prom_printf("\nDR: dr_suspend invoked with force flag\n");
 830         }

 832 #ifndef SKIP_SYNC
 833         /*
 834          * This sync swap out all user pages
 835          */
 836         vfs_sync(SYNC_ALL);
 837 #endif

 839         /*
 840          * special treatment for lock manager
 841          */
 842         lm_cprsuspend();
```



```
 844 #ifndef SKIP_SYNC
 845         /*
 846          * sync the file system in case we never make it back
 847          */
 848         sync();
 849 #endif

 851         /*
 852          * now suspend drivers
 853          */
 854         prom_printf("DR: suspending drivers...\n");
 855         srh->sr_suspend_state = DR_SRSTATE_DRIVER;
 856         srh->sr_err_idx = 0;
 857         /* No parent to hold busy */
 858         if ((rc = dr_suspend_devices(ddi_root_node(), srh)) != DDI_SUCCESS) {
 859                 if (srh->sr_err_idx && srh->sr_dr_handlep) {
 860                         (srh->sr_dr_handlep)->h_err = drerr_int(ESBD_SUSPEND,
 861                             srh->sr_err_ints, srh->sr_err_idx, 1);
 862                 }
 863                 dr_resume(srh);
 864                 return (rc);
 865         }

 867         drmach_suspend_last();

 869         /*
 870          * finally, grab all cpus
 871          */
 872         srh->sr_suspend_state = DR_SRSTATE_FULL;

 874         mutex_enter(&cpu_lock);
 875         pause_cpus(NULL, NULL);
 875         pause_cpus(NULL);
 876         dr_stop_intr();

 878         return (rc);
 879 }
______unchanged_portion_omitted_
```

```
**********************************************************
    4291 Tue Nov  4 16:25:27 2014
new/usr/src/uts/i86pc/io/ppm/acpisleep.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
```

```
/*
 * 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/smp_impldefs.h>
#include <sys/promif.h>

#include <sys/kmem.h>
#include <sys/archsystm.h>
#include <sys/cpuvar.h>
#include <sys/pte.h>
#include <vm/seg_kmem.h>
#include <sys/epm.h>
#include <sys/cpr.h>
#include <sys/machsystm.h>
#include <sys/clock.h>

#include <sys/cpr_wakecode.h>
#include <sys/acpi/acpi.h>

#ifdef OLDPMCODE
#include "acpi.h"
#endif

#include        <sys/x86_archext.h>
#include        <sys/reboot.h>
#include        <sys/cpu_module.h>
#include        <sys/kdi.h>

/*
 * S3 stuff
 */

int acpi_rtc_wake = 0x0;                /* wake in N seconds */

#if 0   /* debug */
static uint8_t  branchbuf[64 * 1024];   /* for the HDT branch trace stuff */
#endif  /* debug */
```



```
extern int boothowto;

#define BOOTCPU 0       /* cpu 0 is always the boot cpu */

extern void             kernel_wc_code(void);
extern tod_ops_t        *tod_ops;
extern int flushes_require_xcalls;
extern int tsc_gethrtime_enable;

extern cpuset_t cpu_ready_set;
extern void *(*cpu_pause_func)(void *);


/*
 * This is what we've all been waiting for!
 */
int
acpi_enter_sleepstate(s3a_t *s3ap)
{
        ACPI_PHYSICAL_ADDRESS   wakephys = s3ap->s3a_wakephys;
        caddr_t                 wakevirt = rm_platter_va;
        /*LINTED*/
        wakecode_t              *wp = (wakecode_t *)wakevirt;
        uint_t                  Sx = s3ap->s3a_state;

        PT(PT_SWV);
        /* Set waking vector */
        if (AcpiSetFirmwareWakingVector(wakephys) != AE_OK) {
                PT(PT_SWV_FAIL);
                PMD(PMD_SX, ("Can't SetFirmwareWakingVector(%lx)\n",
                    (long)wakephys))
                goto insomnia;
        }

        PT(PT_EWE);
        /* Enable wake events */
        if (AcpiEnableEvent(ACPI_EVENT_POWER_BUTTON, 0) != AE_OK) {
                PT(PT_EWE_FAIL);
                PMD(PMD_SX, ("Can't EnableEvent(POWER_BUTTON)\n"))
        }
        if (acpi_rtc_wake > 0) {
                /* clear the RTC bit first */
                (void) AcpiWriteBitRegister(ACPI_BITREG_RT_CLOCK_STATUS, 1);
                PT(PT_RTCW);
                if (AcpiEnableEvent(ACPI_EVENT_RTC, 0) != AE_OK) {
                        PT(PT_RTCW_FAIL);
                        PMD(PMD_SX, ("Can't EnableEvent(RTC)\n"))
                }

                /*
                 * Set RTC to wake us in a wee while.
                 */
                mutex_enter(&tod_lock);
                PT(PT_TOD);
                TODOP_SETWAKE(tod_ops, acpi_rtc_wake);
                mutex_exit(&tod_lock);
        }

        /*
         * Prepare for sleep ... could've done this earlier?
         */
        PT(PT_SXP);
        PMD(PMD_SX, ("Calling AcpiEnterSleepStatePrep(%d) ...\n", Sx))
        if (AcpiEnterSleepStatePrep(Sx) != AE_OK) {
```

```
126                 PMD(PMD_SX, ("... failed\n!"))
127                 goto insomnia;
128         }

130         switch (s3ap->s3a_test_point) {
131         case DEVICE_SUSPEND_TO_RAM:
132         case FORCE_SUSPEND_TO_RAM:
133         case LOOP_BACK_PASS:
134                 return (0);
135         case LOOP_BACK_FAIL:
136                 return (1);
137         default:
138                 ASSERT(s3ap->s3a_test_point == LOOP_BACK_NONE);
139         }

141         /*
142          * Tell the hardware to sleep.
143          */
144         PT(PT_SXE);
145         PMD(PMD_SX, ("Calling AcpiEnterSleepState(%d) ...\n", Sx))
146         if (AcpiEnterSleepState(Sx) != AE_OK) {
147                 PT(PT_SXE_FAIL);
148                 PMD(PMD_SX, ("... failed!\n"))
149         }

151 insomnia:
152         PT(PT_INSOM);
153         /* cleanup is done in the caller */
154         return (1);
155 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   27105 Tue Nov  4 16:25:27 2014
new/usr/src/uts/i86pc/os/cpr_impl.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_

 724 /*
 725  * Stop all other cpu's before halting or rebooting. We pause the cpu's
 726  * instead of sending a cross call.
 727  * Stolen from sun4/os/mp_states.c
 728  */

 730 static int cpu_are_paused;      /* sic */

 732 void
 733 i_cpr_stop_other_cpus(void)
 734 {
 735         mutex_enter(&cpu_lock);
 736         if (cpu_are_paused) {
 737                 mutex_exit(&cpu_lock);
 738                 return;
 739         }
 740         pause_cpus(NULL, NULL);
 740         pause_cpus(NULL);
 741         cpu_are_paused = 1;

 743         mutex_exit(&cpu_lock);
 744 }
______unchanged_portion_omitted_
```

```
**********************************************************
   34387 Tue Nov  4 16:25:27 2014
new/usr/src/uts/i86pc/os/machdep.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
```

```
/*
 * 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) 1992, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright (c) 2010, Intel Corporation.
 * All rights reserved.
 */

#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/param.h>
#include <sys/segments.h>
#include <sys/sysmacros.h>
#include <sys/signal.h>
#include <sys/systm.h>
#include <sys/user.h>
#include <sys/mman.h>
#include <sys/vm.h>

#include <sys/disp.h>
#include <sys/class.h>

#include <sys/proc.h>
#include <sys/buf.h>
#include <sys/kmem.h>

#include <sys/reboot.h>
#include <sys/uadmin.h>
#include <sys/callb.h>

#include <sys/cred.h>
#include <sys/vnode.h>
#include <sys/file.h>

#include <sys/procfs.h>
#include <sys/acct.h>

#include <sys/vfs.h>
#include <sys/dnlc.h>
#include <sys/var.h>
```



```
#include <sys/cmn_err.h>
#include <sys/utsname.h>
#include <sys/debug.h>

#include <sys/dumphdr.h>
#include <sys/bootconf.h>
#include <sys/varargs.h>
#include <sys/promif.h>
#include <sys/modctl.h>

#include <sys/consdev.h>
#include <sys/frame.h>

#include <sys/sunddi.h>
#include <sys/ddidmareq.h>
#include <sys/psw.h>
#include <sys/regset.h>
#include <sys/privregs.h>
#include <sys/clock.h>
#include <sys/tss.h>
#include <sys/cpu.h>
#include <sys/stack.h>
#include <sys/trap.h>
#include <sys/pic.h>
#include <vm/hat.h>
#include <vm/anon.h>
#include <vm/as.h>
#include <vm/page.h>
#include <vm/seg.h>
#include <vm/seg_kmem.h>
#include <vm/seg_map.h>
#include <vm/seg_vn.h>
#include <vm/seg_kp.h>
#include <vm/hat_i86.h>
#include <sys/swap.h>
#include <sys/thread.h>
#include <sys/sysconf.h>
#include <sys/vm_machparam.h>
#include <sys/archsystm.h>
#include <sys/machsystm.h>
#include <sys/machlock.h>
#include <sys/x_call.h>
#include <sys/instance.h>

#include <sys/time.h>
#include <sys/smp_impldefs.h>
#include <sys/psm_types.h>
#include <sys/atomic.h>
#include <sys/panic.h>
#include <sys/cpuvar.h>
#include <sys/dtrace.h>
#include <sys/bl.h>
#include <sys/nvpair.h>
#include <sys/x86_archext.h>
#include <sys/pool_pset.h>
#include <sys/autoconf.h>
#include <sys/mem.h>
#include <sys/dumphdr.h>
#include <sys/compress.h>
#include <sys/cpu_module.h>
#if defined(__xpv)
#include <sys/hypervisor.h>
#include <sys/xpv_panic.h>
#endif

#include <sys/fastboot.h>
```

```
#include <sys/machelf.h>
#include <sys/kobj.h>
#include <sys/multiboot.h>

#ifdef  TRAPTRACE
#include <sys/traptrace.h>
#endif  /* TRAPTRACE */

#include <c2/audit.h>
#include <sys/clock_impl.h>

extern void audit_enterprom(int);
extern void audit_exitprom(int);

/*
 * Tunable to enable apix PSM; if set to 0, pcplusmp PSM will be used.
 */
int     apix_enable = 1;

int     apic_nvidia_io_max = 0; /* no. of NVIDIA i/o apics */

/*
 * Occassionally the kernel knows better whether to power-off or reboot.
 */
int force_shutdown_method = AD_UNKNOWN;

/*
 * The panicbuf array is used to record messages and state:
 */
char panicbuf[PANICBUFSIZE];

/*
 * Flags to control Dynamic Reconfiguration features.
 */
uint64_t plat_dr_options;

/*
 * Maximum physical address for memory DR operations.
 */
uint64_t plat_dr_physmax;

/*
 * maxphys - used during physio
 * klustsize - used for klustering by swapfs and specfs
 */
int maxphys = 56 * 1024;    /* XXX See vm_subr.c - max b_count in physio */
int klustsize = 56 * 1024;

caddr_t p0_va;          /* Virtual address for accessing physical page 0 */

/*
 * defined here, though unused on x86,
 * to make kstat_fr.c happy.
 */
int vac;

void debug_enter(char *);

extern void pm_cfb_check_and_powerup(void);
extern void pm_cfb_rele(void);

extern fastboot_info_t newkernel;

/*
 * Machine dependent code to reboot.
 * "mdep" is interpreted as a character pointer; if non-null, it is a pointer
```



```
 * to a string to be used as the argument string when rebooting.
 *
 * "invoke_cb" is a boolean. It is set to true when mdboot() can safely
 * invoke CB_CL_MDBOOT callbacks before shutting the system down, i.e. when
 * we are in a normal shutdown sequence (interrupts are not blocked, the
 * system is not panic'ing or being suspended).
 */
/*ARGSUSED*/
void
mdboot(int cmd, int fcn, char *mdep, boolean_t invoke_cb)
{
        processorid_t bootcpuid = 0;
        static int is_first_quiesce = 1;
        static int is_first_reset = 1;
        int reset_status = 0;
        static char fallback_str[] = "Falling back to regular reboot.\n";

        if (fcn == AD_FASTREBOOT && !newkernel.fi_valid)
                fcn = AD_BOOT;

        if (!panicstr) {
                kpreempt_disable();
                if (fcn == AD_FASTREBOOT) {
                        mutex_enter(&cpu_lock);
                        if (CPU_ACTIVE(cpu_get(bootcpuid))) {
                                affinity_set(bootcpuid);
                        }
                        mutex_exit(&cpu_lock);
                } else {
                        affinity_set(CPU_CURRENT);
                }
        }

        if (force_shutdown_method != AD_UNKNOWN)
                fcn = force_shutdown_method;

        /*
         * XXX - rconsvp is set to NULL to ensure that output messages
         * are sent to the underlying "hardware" device using the
         * monitor's printf routine since we are in the process of
         * either rebooting or halting the machine.
         */
        rconsvp = NULL;

        /*
         * Print the reboot message now, before pausing other cpus.
         * There is a race condition in the printing support that
         * can deadlock multiprocessor machines.
         */
        if (!(fcn == AD_HALT || fcn == AD_POWEROFF))
                prom_printf("rebooting...\n");

        if (IN_XPV_PANIC())
                reset();

        /*
         * We can't bring up the console from above lock level, so do it now
         */
        pm_cfb_check_and_powerup();

        /* make sure there are no more changes to the device tree */
        devtree_freeze();

        if (invoke_cb)
                (void) callb_execute_class(CB_CL_MDBOOT, NULL);
```

```
260         /*
261          * Clear any unresolved UEs from memory.
262          */
263         page_retire_mdboot();

265 #if defined(__xpv)
266         /*
267          * XXPV Should probably think some more about how we deal
268          *      with panicing before it's really safe to panic.
269          *      On hypervisors, we reboot very quickly..  Perhaps panic
270          *      should only attempt to recover by rebooting if,
271          *      say, we were able to mount the root filesystem,
272          *      or if we successfully launched init(1m).
273          */
274         if (panicstr && proc_init == NULL)
275                 (void) HYPERVISOR_shutdown(SHUTDOWN_poweroff);
276 #endif
277         /*
278          * stop other cpus and raise our priority.  since there is only
279          * one active cpu after this, and our priority will be too high
280          * for us to be preempted, we're essentially single threaded
281          * from here on out.
282          */
283         (void) spl6();
284         if (!panicstr) {
285                 mutex_enter(&cpu_lock);
286                 pause_cpus(NULL, NULL);
286                 pause_cpus(NULL);
287                 mutex_exit(&cpu_lock);
288         }

290         /*
291          * If the system is panicking, the preloaded kernel is valid, and
292          * fastreboot_onpanic has been set, and the system has been up for
293          * longer than fastreboot_onpanic_uptime (default to 10 minutes),
294          * choose Fast Reboot.
295          */
296         if (fcn == AD_BOOT && panicstr && newkernel.fi_valid &&
297             fastreboot_onpanic &&
298             (panic_lbolt - lbolt_at_boot) > fastreboot_onpanic_uptime) {
299                 fcn = AD_FASTREBOOT;
300         }

302         /*
303          * Try to quiesce devices.
304          */
305         if (is_first_quiesce) {
306                 /*
307                  * Clear is_first_quiesce before calling quiesce_devices()
308                  * so that if quiesce_devices() causes panics, it will not
309                  * be invoked again.
310                  */
311                 is_first_quiesce = 0;

313                 quiesce_active = 1;
314                 quiesce_devices(ddi_root_node(), &reset_status);
315                 if (reset_status == -1) {
316                         if (fcn == AD_FASTREBOOT && !force_fastreboot) {
317                                 prom_printf("Driver(s) not capable of fast "
318                                     "reboot.\n");
319                                 prom_printf(fallback_str);
320                                 fastreboot_capable = 0;
321                                 fcn = AD_BOOT;
322                         } else if (fcn != AD_FASTREBOOT)
323                                 fastreboot_capable = 0;
324                 }
```



```
325                 quiesce_active = 0;
326         }

328         /*
329          * Try to reset devices. reset_leaves() should only be called
330          * a) when there are no other threads that could be accessing devices,
331          *    and
332          * b) on a system that's not capable of fast reboot (fastreboot_capable
333          *    being 0), or on a system where quiesce_devices() failed to
334          *    complete (quiesce_active being 1).
335          */
336         if (is_first_reset && (!fastreboot_capable || quiesce_active)) {
337                 /*
338                  * Clear is_first_reset before calling reset_devices()
339                  * so that if reset_devices() causes panics, it will not
340                  * be invoked again.
341                  */
342                 is_first_reset = 0;
343                 reset_leaves();
344         }

346         /* Verify newkernel checksum */
347         if (fastreboot_capable && fcn == AD_FASTREBOOT &&
348             fastboot_cksum_verify(&newkernel) != 0) {
349                 fastreboot_capable = 0;
350                 prom_printf("Fast reboot: checksum failed for the new "
351                     "kernel.\n");
352                 prom_printf(fallback_str);
353         }

355         (void) spl8();

357         if (fastreboot_capable && fcn == AD_FASTREBOOT) {
358                 /*
359                  * psm_shutdown is called within fast_reboot()
360                  */
361                 fast_reboot();
362         } else {
363                 (*psm_shutdownf)(cmd, fcn);

365                 if (fcn == AD_HALT || fcn == AD_POWEROFF)
366                         halt((char *)NULL);
367                 else
368                         prom_reboot("");
369         }
370         /*NOTREACHED*/
371 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   17010 Tue Nov  4 16:25:28 2014
new/usr/src/uts/i86pc/os/mp_pc.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright (c) 2007, 2010, Oracle and/or its affiliates. All rights reserved.
  23  */
  24 /*
  25  * Copyright (c) 2010, Intel Corporation.
  26  * All rights reserved.
  27  */
  28 /*
  29  * Copyright 2011 Joyent, Inc. All rights reserved.
  30  */

  32 /*
  33  * Welcome to the world of the "real mode platter".
  34  * See also startup.c, mpcore.s and apic.c for related routines.
  35  */

  37 #include <sys/types.h>
  38 #include <sys/systm.h>
  39 #include <sys/cpuvar.h>
  40 #include <sys/cpu_module.h>
  41 #include <sys/kmem.h>
  42 #include <sys/archsystm.h>
  43 #include <sys/machsystm.h>
  44 #include <sys/controlregs.h>
  45 #include <sys/x86_archext.h>
  46 #include <sys/smp_impldefs.h>
  47 #include <sys/sysmacros.h>
  48 #include <sys/mach_mmu.h>
  49 #include <sys/promif.h>
  50 #include <sys/cpu.h>
  51 #include <sys/cpu_event.h>
  52 #include <sys/sunndi.h>
  53 #include <sys/fs/dv_node.h>
  54 #include <vm/hat_i86.h>
  55 #include <vm/as.h>

  57 extern cpuset_t cpu_ready_set;

  59 extern int  mp_start_cpu_common(cpu_t *cp, boolean_t boot);
  60 extern void real_mode_start_cpu(void);
  61 extern void real_mode_start_cpu_end(void);
```



```
  62 extern void real_mode_stop_cpu_stage1(void);
  63 extern void real_mode_stop_cpu_stage1_end(void);
  64 extern void real_mode_stop_cpu_stage2(void);
  65 extern void real_mode_stop_cpu_stage2_end(void);
  66 extern void *(*cpu_pause_func)(void *);

  67 void rmp_gdt_init(rm_platter_t *);

  69 /*
  70  * Fill up the real mode platter to make it easy for real mode code to
  71  * kick it off. This area should really be one passed by boot to kernel
  72  * and guaranteed to be below 1MB and aligned to 16 bytes. Should also
  73  * have identical physical and virtual address in paged mode.
  74  */
  75 static ushort_t *warm_reset_vector = NULL;

  77 int
  78 mach_cpucontext_init(void)
  79 {
  80         ushort_t *vec;
  81         ulong_t addr;
  82         struct rm_platter *rm = (struct rm_platter *)rm_platter_va;

  84         if (!(vec = (ushort_t *)psm_map_phys(WARM_RESET_VECTOR,
  85             sizeof (vec), PROT_READ | PROT_WRITE)))
  86                 return (-1);

  88         /*
  89          * setup secondary cpu bios boot up vector
  90          * Write page offset to 0x467 and page frame number to 0x469.
  91          */
  92         addr = (ulong_t)((caddr_t)rm->rm_code - (caddr_t)rm) + rm_platter_pa;
  93         vec[0] = (ushort_t)(addr & PAGEOFFSET);
  94         vec[1] = (ushort_t)((addr & (0xfffff & PAGEMASK)) >> 4);
  95         warm_reset_vector = vec;

  97         /* Map real mode platter into kas so kernel can access it. */
  98         hat_devload(kas.a_hat,
  99             (caddr_t)(uintptr_t)rm_platter_pa, MMU_PAGESIZE,
 100             btop(rm_platter_pa), PROT_READ | PROT_WRITE | PROT_EXEC,
 101             HAT_LOAD_NOCONSIST);

 103         /* Copy CPU startup code to rm_platter if it's still during boot. */
 104         if (!plat_dr_enabled()) {
 105                 ASSERT((size_t)real_mode_start_cpu_end -
 106                     (size_t)real_mode_start_cpu <= RM_PLATTER_CODE_SIZE);
 107                 bcopy((caddr_t)real_mode_start_cpu, (caddr_t)rm->rm_code,
 108                     (size_t)real_mode_start_cpu_end -
 109                     (size_t)real_mode_start_cpu);
 110         }

 112         return (0);
 113 }
```

_____*unchanged_portion_omitted_*

****
18856 Tue Nov 4 16:25:28 2014
new/usr/src/uts/i86pc/os/x_call.c
5285 pass in cpu_pause_func via pause_cpus
****
_____***unchanged_portion_omitted_***

```
 268 #define XC_FLUSH_MAX_WAITS              1000

 270 /* Flush inflight message buffers. */
 271 int
 272 xc_flush_cpu(struct cpu *cpup)
 273 {
 274         int i;

 276         ASSERT((cpup->cpu_flags & CPU_READY) == 0);

 278         /*
 279          * Pause all working CPUs, which ensures that there's no CPU in
 280          * function xc_common().
 281          * This is used to work around a race condition window in xc_common()
 282          * between checking CPU_READY flag and increasing working item count.
 283          */
 284         pause_cpus(cpup, NULL);
 284         pause_cpus(cpup);
 285         start_cpus();

 287         for (i = 0; i < XC_FLUSH_MAX_WAITS; i++) {
 288                 if (cpup->cpu_m.xc_work_cnt == 0) {
 289                         break;
 290                 }
 291                 DELAY(1);
 292         }
 293         for (; i < XC_FLUSH_MAX_WAITS; i++) {
 294                 if (!BT_TEST(xc_priority_set, cpup->cpu_id)) {
 295                         break;
 296                 }
 297                 DELAY(1);
 298         }

 300         return (i >= XC_FLUSH_MAX_WAITS ? ETIME : 0);
 301 }
```

_____***unchanged_portion_omitted_***

****
25091 Tue Nov 4 16:25:28 2014
new/usr/src/uts/i86xpv/os/mp_xen.c
5285 pass in cpu_pause_func via pause_cpus
****
_____***unchanged_portion_omitted_***

```
 579 void
 580 mp_enter_barrier(void)
 581 {
 582         hrtime_t last_poke_time = 0;
 583         int poke_allowed = 0;
 584         int done = 0;
 585         int i;

 587         ASSERT(MUTEX_HELD(&cpu_lock));

 589         pause_cpus(NULL, NULL);
 589         pause_cpus(NULL);

 591         while (!done) {
 592                 done = 1;
 593                 poke_allowed = 0;

 595                 if (xpv_gethrtime() - last_poke_time > POKE_TIMEOUT) {
 596                         last_poke_time = xpv_gethrtime();
 597                         poke_allowed = 1;
 598                 }

 600                 for (i = 0; i < NCPU; i++) {
 601                         cpu_t *cp = cpu_get(i);

 603                         if (cp == NULL || cp == CPU)
 604                                 continue;

 606                         switch (cpu_phase[i]) {
 607                         case CPU_PHASE_NONE:
 608                                 cpu_phase[i] = CPU_PHASE_WAIT_SAFE;
 609                                 poke_cpu(i);
 610                                 done = 0;
 611                                 break;

 613                         case CPU_PHASE_WAIT_SAFE:
 614                                 if (poke_allowed)
 615                                         poke_cpu(i);
 616                                 done = 0;
 617                                 break;

 619                         case CPU_PHASE_SAFE:
 620                         case CPU_PHASE_POWERED_OFF:
 621                                 break;
 622                         }
 623                 }

 625                 SMT_PAUSE();
 626         }
 627 }
```

_____***unchanged_portion_omitted_***

****
6667 Tue Nov 4 16:25:28 2014
new/usr/src/uts/sun4/os/mp_states.c
5285 pass in cpu_pause_func via pause_cpus
****
_____***unchanged_portion_omitted_***

```
 187 /*
 188  * Stop all other cpu's before halting or rebooting. We pause the cpu's
 189  * instead of sending a cross call.
 190  */
 191 void
 192 stop_other_cpus(void)
 193 {
 194         mutex_enter(&cpu_lock);
 195         if (cpu_are_paused) {
 196                 mutex_exit(&cpu_lock);
 197                 return;
 198         }

 200         if (ncpus > 1)
 201                 intr_redist_all_cpus_shutdown();

 203         pause_cpus(NULL, NULL);
 203         pause_cpus(NULL);
 204         cpu_are_paused = 1;

 206         mutex_exit(&cpu_lock);
 207 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   16813 Tue Nov  4 16:25:28 2014
new/usr/src/uts/sun4/os/prom_subr.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_

 404 /*
 405  * This routine is a special form of pause_cpus().  It ensures that
 406  * prom functions are callable while the cpus are paused.
 407  */
 408 void
 409 promsafe_pause_cpus(void)
 410 {
 411         pause_cpus(NULL, NULL);
 411         pause_cpus(NULL);

 413         /* If some other cpu is entering or is in the prom, spin */
 414         while (prom_cpu || mutex_owner(&prom_mutex)) {

 416                 start_cpus();
 417                 mutex_enter(&prom_mutex);

 419                 /* Wait for other cpu to exit prom */
 420                 while (prom_cpu)
 421                         cv_wait(&prom_cv, &prom_mutex);

 423                 mutex_exit(&prom_mutex);
 424                 pause_cpus(NULL, NULL);
 424                 pause_cpus(NULL);
 425         }

 427         /* At this point all cpus are paused and none are in the prom */
 428 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   24533 Tue Nov  4 16:25:28 2014
new/usr/src/uts/sun4u/io/mem_cache.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_

 553 static int
 554 mem_cache_ioctl_ops(int cmd, int mode, cache_info_t *cache_info)
 555 {
 556         int     ret_val = 0;
 557         uint64_t afar, tag_addr;
 558         ch_cpu_logout_t clop;
 559         uint64_t Lxcache_tag_data[PN_CACHE_NWAYS];
 560         int     i, retire_retry_count;
 561         cpu_t   *cpu;
 562         uint64_t tag_data;
 563         uint8_t state;

 565         if (cache_info->way >= PN_CACHE_NWAYS)
 566                 return (EINVAL);
 567         switch (cache_info->cache) {
 568                 case L2_CACHE_TAG:
 569                 case L2_CACHE_DATA:
 570                         if (cache_info->index >=
 571                             (PN_L2_SET_SIZE/PN_L2_LINESIZE))
 572                                 return (EINVAL);
 573                         break;
 574                 case L3_CACHE_TAG:
 575                 case L3_CACHE_DATA:
 576                         if (cache_info->index >=
 577                             (PN_L3_SET_SIZE/PN_L3_LINESIZE))
 578                                 return (EINVAL);
 579                         break;
 580                 default:
 581                         return (ENOTSUP);
 582         }
 583         /*
 584          * Check if we have a valid cpu ID and that
 585          * CPU is ONLINE.
 586          */
 587         mutex_enter(&cpu_lock);
 588         cpu = cpu_get(cache_info->cpu_id);
 589         if ((cpu == NULL) || (!cpu_is_online(cpu))) {
 590                 mutex_exit(&cpu_lock);
 591                 return (EINVAL);
 592         }
 593         mutex_exit(&cpu_lock);
 594         pattern = 0;    /* default value of TAG PA when cacheline is retired. */
 595         switch (cmd) {
 596                 case MEM_CACHE_RETIRE:
 597                         tag_addr = get_tag_addr(cache_info);
 598                         pattern |= PN_ECSTATE_NA;
 599                         retire_retry_count = 0;
 600                         affinity_set(cache_info->cpu_id);
 601                         switch (cache_info->cache) {
 602                                 case L2_CACHE_DATA:
 603                                 case L2_CACHE_TAG:
 604                                         if ((cache_info->bit & MSB_BIT_MASK) ==
 605                                             MSB_BIT_MASK)
 606                                                 pattern |= PN_L2TAG_PA_MASK;
 607 retry_l2_retire:
 608                                         if (tag_addr_collides(tag_addr,
 609                                             cache_info->cache,
 610                                             retire_l2_start, retire_l2_end))
 611                                                 ret_val =
 612                                                     retire_l2_alternate(
 613                                                     tag_addr, pattern);
 614                                         else
 615                                                 ret_val = retire_l2(tag_addr,
 616                                                     pattern);
 617                                         if (ret_val == 1) {
 618                                                 /*
 619                                                  * cacheline was in retired
 620                                                  * STATE already.
 621                                                  * so return success.
 622                                                  */
 623                                                 ret_val = 0;
 624                                         }
 625                                         if (ret_val < 0) {
 626                                                 cmn_err(CE_WARN,
 627                 "retire_l2() failed. index = 0x%x way %d. Retrying...\n",
 628                                                     cache_info->index,
 629                                                     cache_info->way);
 630                                                 if (retire_retry_count >= 2) {
 631                                                         retire_failures++;
 632                                                         affinity_clear();
 633                                                         return (EIO);
 634                                                 }
 635                                                 retire_retry_count++;
 636                                                 goto retry_l2_retire;
 637                                         }
 638                                         if (ret_val == 2)
 639                                                 l2_flush_retries_done++;
 640                                 /*
 641                                  * We bind ourself to a CPU and send cross trap to
 642                                  * ourself. On return from xt_one we can rely on the
 643                                  * data in tag_data being filled in. Normally one would
 644                                  * do a xt_sync to make sure that the CPU has completed
 645                                  * the cross trap call xt_one.
 646                                  */
 647                                         xt_one(cache_info->cpu_id,
 648                                             (xcfunc_t *)(get_l2_tag_tl1),
 649                                             tag_addr, (uint64_t)(&tag_data));
 650                                         state = tag_data & CH_ECSTATE_MASK;
 651                                         if (state != PN_ECSTATE_NA) {
 652                                                 retire_failures++;
 653                                                 print_l2_tag(tag_addr,
 654                                                     tag_data);
 655                                                 cmn_err(CE_WARN,
 656                 "L2 RETIRE:failed for index 0x%x way %d. Retrying...\n",
 657                                                     cache_info->index,
 658                                                     cache_info->way);
 659                                                 if (retire_retry_count >= 2) {
 660                                                         retire_failures++;
 661                                                         affinity_clear();
 662                                                         return (EIO);
 663                                                 }
 664                                                 retire_retry_count++;
 665                                                 goto retry_l2_retire;
 666                                         }
 667                                         break;
 668                                 case L3_CACHE_TAG:
 669                                 case L3_CACHE_DATA:
 670                                         if ((cache_info->bit & MSB_BIT_MASK) ==
 671                                             MSB_BIT_MASK)
 672                                                 pattern |= PN_L3TAG_PA_MASK;
 673                                         if (tag_addr_collides(tag_addr,
 674                                             cache_info->cache,
 675                                             retire_l3_start, retire_l3_end))
 676                                                 ret_val =
 677                                                     retire_l3_alternate(
```

```
678                                             tag_addr, pattern);
679                                     else
680                                             ret_val = retire_l3(tag_addr,
681                                                 pattern);
682                                     if (ret_val == 1) {
683                                             /*
684                                              * cacheline was in retired
685                                              * STATE already.
686                                              * so return success.
687                                              */
688                                             ret_val = 0;
689                                     }
690                                     if (ret_val < 0) {
691                                             cmn_err(CE_WARN,
692                             "retire_l3() failed. ret_val = %d index = 0x%x\n",
693                                                 ret_val,
694                                                 cache_info->index);
695                                             retire_failures++;
696                                             affinity_clear();
697                                             return (EIO);
698                                     }
699                             /*
700                              * We bind ourself to a CPU and send cross trap to
701                              * ourself. On return from xt_one we can rely on the
702                              * data in tag_data being filled in. Normally one would
703                              * do a xt_sync to make sure that the CPU has completed
704                              * the cross trap call xt_one.
705                              */
706                                     xt_one(cache_info->cpu_id,
707                                         (xcfunc_t *)(get_l3_tag_tl1),
708                                         tag_addr, (uint64_t)(&tag_data));
709                                     state = tag_data & CH_ECSTATE_MASK;
710                                     if (state != PN_ECSTATE_NA) {
711                                             cmn_err(CE_WARN,
712                                     "L3 RETIRE failed for index 0x%x\n",
713                                                 cache_info->index);
714                                             retire_failures++;
715                                             affinity_clear();
716                                             return (EIO);
717                                     }

719                                     break;
720                             }
721                             affinity_clear();
722                             break;
723                     case MEM_CACHE_UNRETIRE:
724                             tag_addr = get_tag_addr(cache_info);
725                             pattern = PN_ECSTATE_INV;
726                             affinity_set(cache_info->cpu_id);
727                             switch (cache_info->cache) {
728                                     case L2_CACHE_DATA:
729                                     case L2_CACHE_TAG:
730                             /*
731                              * We bind ourself to a CPU and send cross trap to
732                              * ourself. On return from xt_one we can rely on the
733                              * data in tag_data being filled in. Normally one would
734                              * do a xt_sync to make sure that the CPU has completed
735                              * the cross trap call xt_one.
736                              */
737                                     xt_one(cache_info->cpu_id,
738                                         (xcfunc_t *)(get_l2_tag_tl1),
739                                         tag_addr, (uint64_t)(&tag_data));
740                                     state = tag_data & CH_ECSTATE_MASK;
741                                     if (state != PN_ECSTATE_NA) {
742                                             affinity_clear();
743                                             return (EINVAL);
```



```
744                                     }
745                                     if (tag_addr_collides(tag_addr,
746                                         cache_info->cache,
747                                         unretire_l2_start, unretire_l2_end))
748                                             ret_val =
749                                                 unretire_l2_alternate(
750                                                 tag_addr, pattern);
751                                     else
752                                             ret_val =
753                                                 unretire_l2(tag_addr,
754                                                 pattern);
755                                     if (ret_val != 0) {
756                                             cmn_err(CE_WARN,
757                             "unretire_l2() failed. ret_val = %d index = 0x%x\n",
758                                                 ret_val,
759                                                 cache_info->index);
760                                             retire_failures++;
761                                             affinity_clear();
762                                             return (EIO);
763                                     }
764                                     break;
765                             case L3_CACHE_TAG:
766                             case L3_CACHE_DATA:
767                             /*
768                              * We bind ourself to a CPU and send cross trap to
769                              * ourself. On return from xt_one we can rely on the
770                              * data in tag_data being filled in. Normally one would
771                              * do a xt_sync to make sure that the CPU has completed
772                              * the cross trap call xt_one.
773                              */
774                                     xt_one(cache_info->cpu_id,
775                                         (xcfunc_t *)(get_l3_tag_tl1),
776                                         tag_addr, (uint64_t)(&tag_data));
777                                     state = tag_data & CH_ECSTATE_MASK;
778                                     if (state != PN_ECSTATE_NA) {
779                                             affinity_clear();
780                                             return (EINVAL);
781                                     }
782                                     if (tag_addr_collides(tag_addr,
783                                         cache_info->cache,
784                                         unretire_l3_start, unretire_l3_end))
785                                             ret_val =
786                                                 unretire_l3_alternate(
787                                                 tag_addr, pattern);
788                                     else
789                                             ret_val =
790                                                 unretire_l3(tag_addr,
791                                                 pattern);
792                                     if (ret_val != 0) {
793                                             cmn_err(CE_WARN,
794                             "unretire_l3() failed. ret_val = %d index = 0x%x\n",
795                                                 ret_val,
796                                                 cache_info->index);
797                                             affinity_clear();
798                                             return (EIO);
799                                     }
800                                     break;
801                             }
802                             affinity_clear();
803                             break;
804                     case MEM_CACHE_ISRETIRED:
805                     case MEM_CACHE_STATE:
806                             return (ENOTSUP);
807                     case MEM_CACHE_READ_TAGS:
808 #ifdef DEBUG
809                     case MEM_CACHE_READ_ERROR_INJECTED_TAGS:
```

```
#endif
                        /*
                         * Read tag and data for all the ways at a given afar
                         */
                        afar = (uint64_t)(cache_info->index
                            << PN_CACHE_LINE_SHIFT);
                        mutex_enter(&cpu_lock);
                        affinity_set(cache_info->cpu_id);
                        (void) pause_cpus(NULL, NULL);
                        (void) pause_cpus(NULL);
                        mutex_exit(&cpu_lock);
                        /*
                         * We bind ourself to a CPU and send cross trap to
                         * ourself. On return from xt_one we can rely on the
                         * data in clop being filled in. Normally one would
                         * do a xt_sync to make sure that the CPU has completed
                         * the cross trap call xt_one.
                         */
                        xt_one(cache_info->cpu_id,
                            (xcfunc_t *)(get_ecache_dtags_tl1),
                            afar, (uint64_t)(&clop));
                        mutex_enter(&cpu_lock);
                        (void) start_cpus();
                        mutex_exit(&cpu_lock);
                        affinity_clear();
                        switch (cache_info->cache) {
                                case L2_CACHE_TAG:
                                        for (i = 0; i < PN_CACHE_NWAYS; i++) {
                                                Lxcache_tag_data[i] =
                                                    clop.clo_data.chd_l2_data
                                                    [i].ec_tag;
                                        }
#ifdef DEBUG
                                        last_error_injected_bit =
                                            last_l2tag_error_injected_bit;
                                        last_error_injected_way =
                                            last_l2tag_error_injected_way;
#endif
                                        break;
                                case L3_CACHE_TAG:
                                        for (i = 0; i < PN_CACHE_NWAYS; i++) {
                                                Lxcache_tag_data[i] =
                                                    clop.clo_data.chd_ec_data
                                                    [i].ec_tag;
                                        }
#ifdef DEBUG
                                        last_error_injected_bit =
                                            last_l3tag_error_injected_bit;
                                        last_error_injected_way =
                                            last_l3tag_error_injected_way;
#endif
                                        break;
                                default:
                                        return (ENOTSUP);
                        }       /* end if switch(cache) */
#ifdef DEBUG
                        if ((cmd == MEM_CACHE_READ_ERROR_INJECTED_TAGS) &&
                            (inject_anonymous_tag_error == 0) &&
                            (last_error_injected_way >= 0) &&
                            (last_error_injected_way <= 3)) {
                                pattern = ((uint64_t)1 <<
                                    last_error_injected_bit);
                                /*
                                 * If error bit is ECC we need to make sure
                                 * ECC on all all WAYS are corrupted.
                                 */
```



```
                                if ((last_error_injected_bit >= 6) &&
                                    (last_error_injected_bit <= 14)) {
                                        for (i = 0; i < PN_CACHE_NWAYS; i++)
                                                Lxcache_tag_data[i] ^=
                                                    pattern;
                                } else
                                        Lxcache_tag_data
                                            [last_error_injected_way] ^=
                                            pattern;
                        }
#endif
                        if (ddi_copyout((caddr_t)Lxcache_tag_data,
                            (caddr_t)cache_info->datap,
                            sizeof (Lxcache_tag_data), mode)
                            != DDI_SUCCESS) {
                                return (EFAULT);
                        }
                        break;  /* end of READ_TAGS */
                default:
                        return (ENOTSUP);
        }       /* end if switch(cmd) */
        return (ret_val);
}
```

_____unchanged_portion_omitted_

```
**********************************************************
   25005 Tue Nov  4 16:25:29 2014
new/usr/src/uts/sun4u/ngdr/io/dr_quiesce.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_

 822 int
 823 dr_suspend(dr_sr_handle_t *srh)
 824 {
 825         dr_handle_t     *handle;
 826         int             force;
 827         int             dev_errs_idx;
 828         uint64_t        dev_errs[DR_MAX_ERR_INT];
 829         int             rc = DDI_SUCCESS;

 831         handle = srh->sr_dr_handlep;

 833         force = dr_cmd_flags(handle) & SBD_FLAG_FORCE;

 835         /*
 836          * update the signature block
 837          */
 838         CPU_SIGNATURE(OS_SIG, SIGST_QUIESCE_INPROGRESS, SIGSUBST_NULL,
 839             CPU->cpu_id);

 841         prom_printf("\nDR: suspending user threads...\n");
 842         srh->sr_suspend_state = DR_SRSTATE_USER;
 843         if (((rc = dr_stop_user_threads(srh)) != DDI_SUCCESS) &&
 844             dr_check_user_stop_result) {
 845                 dr_resume(srh);
 846                 return (rc);
 847         }

 849         if (!force) {
 850                 struct dr_ref drc = {0};

 852                 prom_printf("\nDR: checking devices...\n");
 853                 dev_errs_idx = 0;

 855                 drc.arr = dev_errs;
 856                 drc.idx = &dev_errs_idx;
 857                 drc.len = DR_MAX_ERR_INT;

 859                 /*
 860                  * Since the root node can never go away, it
 861                  * doesn't have to be held.
 862                  */
 863                 ddi_walk_devs(ddi_root_node(), dr_check_unsafe_major, &drc);
 864                 if (dev_errs_idx) {
 865                         handle->h_err = drerr_int(ESBD_UNSAFE, dev_errs,
 866                             dev_errs_idx, 1);
 867                         dr_resume(srh);
 868                         return (DDI_FAILURE);
 869                 }
 870                 PR_QR("done\n");
 871         } else {
 872                 prom_printf("\nDR: dr_suspend invoked with force flag\n");
 873         }

 875 #ifndef SKIP_SYNC
 876         /*
 877          * This sync swap out all user pages
 878          */
 879         vfs_sync(SYNC_ALL);
 880 #endif
```



```
 882         /*
 883          * special treatment for lock manager
 884          */
 885         lm_cprsuspend();

 887 #ifndef SKIP_SYNC
 888         /*
 889          * sync the file system in case we never make it back
 890          */
 891         sync();
 892 #endif

 894         /*
 895          * now suspend drivers
 896          */
 897         prom_printf("DR: suspending drivers...\n");
 898         srh->sr_suspend_state = DR_SRSTATE_DRIVER;
 899         srh->sr_err_idx = 0;
 900         /* No parent to hold busy */
 901         if ((rc = dr_suspend_devices(ddi_root_node(), srh)) != DDI_SUCCESS) {
 902                 if (srh->sr_err_idx && srh->sr_dr_handlep) {
 903                         (srh->sr_dr_handlep)->h_err = drerr_int(ESBD_SUSPEND,
 904                             srh->sr_err_ints, srh->sr_err_idx, 1);
 905                 }
 906                 dr_resume(srh);
 907                 return (rc);
 908         }

 910         drmach_suspend_last();

 912         /*
 913          * finally, grab all cpus
 914          */
 915         srh->sr_suspend_state = DR_SRSTATE_FULL;

 917         /*
 918          * if watchdog was activated, disable it
 919          */
 920         if (watchdog_activated) {
 921                 mutex_enter(&tod_lock);
 922                 tod_ops.tod_clear_watchdog_timer();
 923                 mutex_exit(&tod_lock);
 924                 srh->sr_flags |= SR_FLAG_WATCHDOG;
 925         } else {
 926                 srh->sr_flags &= ~(SR_FLAG_WATCHDOG);
 927         }

 929         /*
 930          * Update the signature block.
 931          * This must be done before cpus are paused, since on Starcat the
 932          * cpu signature update aquires an adaptive mutex in the iosram driver.
 933          * Blocking with cpus paused can lead to deadlock.
 934          */
 935         CPU_SIGNATURE(OS_SIG, SIGST_QUIESCED, SIGSUBST_NULL, CPU->cpu_id);

 937         mutex_enter(&cpu_lock);
 938         pause_cpus(NULL, NULL);
 938         pause_cpus(NULL);
 939         dr_stop_intr();

 941         return (rc);
 942 }
______unchanged_portion_omitted_
```

```
**********************************************************
   50490 Tue Nov  4 16:25:29 2014
new/usr/src/uts/sun4u/os/cpr_impl.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_


/*
 * launch slave cpus into kernel text, pause them,
 * and restore the original prom pages
 */
void
i_cpr_mp_setup(void)
{
        extern void restart_other_cpu(int);
        cpu_t *cp;

        uint64_t kctx = kcontextreg;

        /*
         * Do not allow setting page size codes in MMU primary context
         * register while using cif wrapper. This is needed to work
         * around OBP incorrect handling of this MMU register.
         */
        kcontextreg = 0;

        /*
         * reset cpu_ready_set so x_calls work properly
         */
        CPUSET_ZERO(cpu_ready_set);
        CPUSET_ADD(cpu_ready_set, getprocessorid());

        /*
         * setup cif to use the cookie from the new/tmp prom
         * and setup tmp handling for calling prom services.
         */
        i_cpr_cif_setup(CIF_SPLICE);

        /*
         * at this point, only the nucleus and a few cpr pages are
         * mapped in.  once we switch to the kernel trap table,
         * we can access the rest of kernel space.
         */
        prom_set_traptable(&trap_table);

        if (ncpus > 1) {
                sfmmu_init_tsbs();

                mutex_enter(&cpu_lock);
                /*
                 * All of the slave cpus are not ready at this time,
                 * yet the cpu structures have various cpu_flags set;
                 * clear cpu_flags and mutex_ready.
                 * Since we are coming up from a CPU suspend, the slave cpus
                 * are frozen.
                 */
                for (cp = CPU->cpu_next; cp != CPU; cp = cp->cpu_next) {
                        cp->cpu_flags = CPU_FROZEN;
                        cp->cpu_m.mutex_ready = 0;
                }

                for (cp = CPU->cpu_next; cp != CPU; cp = cp->cpu_next)
                        restart_other_cpu(cp->cpu_id);

                pause_cpus(NULL, NULL);
```



```
                pause_cpus(NULL);
                mutex_exit(&cpu_lock);

                i_cpr_xcall(i_cpr_clear_entries);
        } else
                i_cpr_clear_entries(0, 0);

        /*
         * now unlink the cif wrapper;  WARNING: do not call any
         * prom_xxx() routines until after prom pages are restored.
         */
        i_cpr_cif_setup(CIF_UNLINK);

        (void) i_cpr_prom_pages(CPR_PROM_RESTORE);

        /* allow setting page size codes in MMU primary context register */
        kcontextreg = kctx;
}
______unchanged_portion_omitted_
```

```
**********************************************************
   19584 Tue Nov  4 16:25:29 2014
new/usr/src/uts/sun4u/serengeti/io/sbdp_quiesce.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_

 764 int
 765 sbdp_suspend(sbdp_sr_handle_t *srh)
 766 {
 767         int force;
 768         int rc = DDI_SUCCESS;

 770         force = (srh && (srh->sr_flags & SBDP_IOCTL_FLAG_FORCE));

 772         /*
 773          * if no force flag, check for unsafe drivers
 774          */
 775         if (force) {
 776                 SBDP_DBG_QR("\nsbdp_suspend invoked with force flag");
 777         }

 779         /*
 780          * update the signature block
 781          */
 782         CPU_SIGNATURE(OS_SIG, SIGST_QUIESCE_INPROGRESS, SIGSUBST_NULL,
 783             CPU->cpu_id);

 785         /*
 786          * first, stop all user threads
 787          */
 788         SBDP_DBG_QR("SBDP: suspending user threads...\n");
 789         SR_SET_STATE(srh, SBDP_SRSTATE_USER);
 790         if (((rc = sbdp_stop_user_threads(srh)) != DDI_SUCCESS) &&
 791             sbdp_check_user_stop_result) {
 792                 sbdp_resume(srh);
 793                 return (rc);
 794         }

 796 #ifndef SKIP_SYNC
 797         /*
 798          * This sync swap out all user pages
 799          */
 800         vfs_sync(SYNC_ALL);
 801 #endif

 803         /*
 804          * special treatment for lock manager
 805          */
 806         lm_cprsuspend();

 808 #ifndef SKIP_SYNC
 809         /*
 810          * sync the file system in case we never make it back
 811          */
 812         sync();

 814 #endif
 815         /*
 816          * now suspend drivers
 817          */
 818         SBDP_DBG_QR("SBDP: suspending drivers...\n");
 819         SR_SET_STATE(srh, SBDP_SRSTATE_DRIVER);

 821         /*
 822          * Root node doesn't have to be held in any way.
```



```
 823          */
 824         if ((rc = sbdp_suspend_devices(ddi_root_node(), srh)) != DDI_SUCCESS) {
 825                 sbdp_resume(srh);
 826                 return (rc);
 827         }

 829         /*
 830          * finally, grab all cpus
 831          */
 832         SR_SET_STATE(srh, SBDP_SRSTATE_FULL);

 834         /*
 835          * if watchdog was activated, disable it
 836          */
 837         if (watchdog_activated) {
 838                 mutex_enter(&tod_lock);
 839                 saved_watchdog_seconds = tod_ops.tod_clear_watchdog_timer();
 840                 mutex_exit(&tod_lock);
 841                 SR_SET_FLAG(srh, SR_FLAG_WATCHDOG);
 842         } else {
 843                 SR_CLEAR_FLAG(srh, SR_FLAG_WATCHDOG);
 844         }

 846         mutex_enter(&cpu_lock);
 847         pause_cpus(NULL, NULL);
 847         pause_cpus(NULL);
 848         sbdp_stop_intr();

 850         /*
 851          * update the signature block
 852          */
 853         CPU_SIGNATURE(OS_SIG, SIGST_QUIESCED, SIGSUBST_NULL, CPU->cpu_id);

 855         return (rc);
 856 }
______unchanged_portion_omitted_
```

```
**********************************************************
   54782 Tue Nov  4 16:25:29 2014
new/usr/src/uts/sun4v/os/mpo.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_

 210 /*
 211  * The MPO locks are to protect the MPO metadata while that
 212  * information is updated as a result of a memory DR operation.
 213  * The read lock must be acquired to read the metadata and the
 214  * write locks must be acquired to update it.
 215  */
 216 #define mpo_rd_lock     kpreempt_disable
 217 #define mpo_rd_unlock   kpreempt_enable

 219 static void
 220 mpo_wr_lock()
 221 {
 222         mutex_enter(&cpu_lock);
 223         pause_cpus(NULL, NULL);
 223         pause_cpus(NULL);
 224         mutex_exit(&cpu_lock);
 225 }
______unchanged_portion_omitted_
```

```
**********************************************************
   21873 Tue Nov  4 16:25:29 2014
new/usr/src/uts/sun4v/os/suspend.c
5285 pass in cpu_pause_func via pause_cpus
**********************************************************
______unchanged_portion_omitted_

 348 /*
 349  * Obtain an updated MD from the hypervisor and update cpunodes, CPU HW
 350  * sharing data structures, and processor groups.
 351  */
 352 static void
 353 update_cpu_mappings(void)
 354 {
 355         md_t            *mdp;
 356         processorid_t   id;
 357         cpu_t           *cp;
 358         cpu_pg_t        *pgps[NCPU];

 360         if ((mdp = md_get_handle()) == NULL) {
 361                 DBG("suspend: md_get_handle failed");
 362                 return;
 363         }

 365         DBG("suspend: updating CPU mappings");

 367         mutex_enter(&cpu_lock);

 369         setup_chip_mappings(mdp);
 370         setup_exec_unit_mappings(mdp);
 371         for (id = 0; id < NCPU; id++) {
 372                 if ((cp = cpu_get(id)) == NULL)
 373                         continue;
 374                 cpu_map_exec_units(cp);
 375         }

 377         /*
 378          * Re-calculate processor groups.
 379          *
 380          * First tear down all PG information before adding any new PG
 381          * information derived from the MD we just downloaded. We must
 382          * call pg_cpu_inactive and pg_cpu_active with CPUs paused and
 383          * we want to minimize the number of times pause_cpus is called.
 384          * Inactivating all CPUs would leave PGs without any active CPUs,
 385          * so while CPUs are paused, call pg_cpu_inactive and swap in the
 386          * bootstrap PG structure saving the original PG structure to be
 387          * fini'd afterwards. This prevents the dispatcher from encountering
 388          * PGs in which all CPUs are inactive. Offline CPUs are already
 389          * inactive in their PGs and shouldn't be reactivated, so we must
 390          * not call pg_cpu_inactive or pg_cpu_active for those CPUs.
 391          */
 392         pause_cpus(NULL, NULL);
 392         pause_cpus(NULL);
 393         for (id = 0; id < NCPU; id++) {
 394                 if ((cp = cpu_get(id)) == NULL)
 395                         continue;
 396                 if ((cp->cpu_flags & CPU_OFFLINE) == 0)
 397                         pg_cpu_inactive(cp);
 398                 pgps[id] = cp->cpu_pg;
 399                 pg_cpu_bootstrap(cp);
 400         }
 401         start_cpus();

 403         /*
 404          * pg_cpu_fini* and pg_cpu_init* must be called while CPUs are
 405          * not paused. Use two separate loops here so that we do not
```



```
 406          * initialize PG data for CPUs until all the old PG data structures
 407          * are torn down.
 408          */
 409         for (id = 0; id < NCPU; id++) {
 410                 if ((cp = cpu_get(id)) == NULL)
 411                         continue;
 412                 pg_cpu_fini(cp, pgps[id]);
 413                 mpo_cpu_remove(id);
 414         }

 416         /*
 417          * Initialize PG data for each CPU, but leave the bootstrapped
 418          * PG structure in place to avoid running with any PGs containing
 419          * nothing but inactive CPUs.
 420          */
 421         for (id = 0; id < NCPU; id++) {
 422                 if ((cp = cpu_get(id)) == NULL)
 423                         continue;
 424                 mpo_cpu_add(mdp, id);
 425                 pgps[id] = pg_cpu_init(cp, B_TRUE);
 426         }

 428         /*
 429          * Now that PG data has been initialized for all CPUs in the
 430          * system, replace the bootstrapped PG structure with the
 431          * initialized PG structure and call pg_cpu_active for each CPU.
 432          */
 433         pause_cpus(NULL, NULL);
 433         pause_cpus(NULL);
 434         for (id = 0; id < NCPU; id++) {
 435                 if ((cp = cpu_get(id)) == NULL)
 436                         continue;
 437                 cp->cpu_pg = pgps[id];
 438                 if ((cp->cpu_flags & CPU_OFFLINE) == 0)
 439                         pg_cpu_active(cp);
 440         }
 441         start_cpus();

 443         mutex_exit(&cpu_lock);

 445         (void) md_fini_handle(mdp);
 446 }
______unchanged_portion_omitted_

 585 /*
 586  * Suspends the OS by pausing CPUs and calling into the HV to initiate
 587  * the suspend. When the HV routine hv_guest_suspend returns, the system
 588  * will be resumed. Must be called after a successful call to suspend_pre.
 589  * suspend_post must be called after suspend_start, whether or not
 590  * suspend_start returns an error.
 591  */
 592 /*ARGSUSED*/
 593 int
 594 suspend_start(char *error_reason, size_t max_reason_len)
 595 {
 596         uint64_t        source_tick;
 597         uint64_t        source_stick;
 598         uint64_t        rv;
 599         timestruc_t     source_tod;
 600         int             spl;

 602         ASSERT(suspend_supported());
 603         DBG("suspend: %s", __func__);

 605         sfmmu_ctxdoms_lock();
```

```
607         mutex_enter(&cpu_lock);

609         /* Suspend the watchdog */
610         watchdog_suspend();

612         /* Record the TOD */
613         mutex_enter(&tod_lock);
614         source_tod = tod_get();
615         mutex_exit(&tod_lock);

617         /* Pause all other CPUs */
618         pause_cpus(NULL, NULL);
618         pause_cpus(NULL);
619         DBG_PROM("suspend: CPUs paused\n");

621         /* Suspend cyclics */
622         cyclic_suspend();
623         DBG_PROM("suspend: cyclics suspended\n");

625         /* Disable interrupts */
626         spl = spl8();
627         DBG_PROM("suspend: spl8()\n");

629         source_tick = gettick_counter();
630         source_stick = gettick();
631         DBG_PROM("suspend: source_tick: 0x%lx\n", source_tick);
632         DBG_PROM("suspend: source_stick: 0x%lx\n", source_stick);

634         /*
635          * Call into the HV to initiate the suspend. hv_guest_suspend()
636          * returns after the guest has been resumed or if the suspend
637          * operation failed or was cancelled. After a successful suspend,
638          * the %tick and %stick registers may have changed by an amount
639          * that is not proportional to the amount of time that has passed.
640          * They may have jumped forwards or backwards. Some variation is
641          * allowed and accounted for using suspend_tick_stick_max_delta,
642          * but otherwise this jump must be uniform across all CPUs and we
643          * operate under the assumption that it is (maintaining two global
644          * offset variables--one for %tick and one for %stick.)
645          */
646         DBG_PROM("suspend: suspending... \n");
647         rv = hv_guest_suspend();
648         if (rv != 0) {
649                 splx(spl);
650                 cyclic_resume();
651                 start_cpus();
652                 watchdog_resume();
653                 mutex_exit(&cpu_lock);
654                 sfmmu_ctxdoms_unlock();
655                 DBG("suspend: failed, rv: %ld\n", rv);
656                 return (rv);
657         }

659         suspend_count++;

661         /* Update the global tick and stick offsets and the preserved TOD */
662         set_tick_offsets(source_tick, source_stick, &source_tod);

664         /* Ensure new offsets are globally visible before resuming CPUs */
665         membar_sync();

667         /* Enable interrupts */
668         splx(spl);

670         /* Set the {%tick,%stick}.NPT bits on all CPUs */
671         if (enable_user_tick_stick_emulation) {
```



```
672                 xc_all((xcfunc_t *)enable_tick_stick_npt, NULL, NULL);
673                 xt_sync(cpu_ready_set);
674                 ASSERT(gettick_npt() != 0);
675                 ASSERT(getstick_npt() != 0);
676         }

678         /* If emulation is enabled, but not currently active, enable it */
679         if (enable_user_tick_stick_emulation && !tick_stick_emulation_active) {
680                 tick_stick_emulation_active = B_TRUE;
681         }

683         sfmmu_ctxdoms_remove();

685         /* Resume cyclics, unpause CPUs */
686         cyclic_resume();
687         start_cpus();

689         /* Set the TOD */
690         mutex_enter(&tod_lock);
691         tod_set(source_tod);
692         mutex_exit(&tod_lock);

694         /* Re-enable the watchdog */
695         watchdog_resume();

697         mutex_exit(&cpu_lock);

699         /* Download the latest MD */
700         if ((rv = mach_descrip_update()) != 0)
701                 cmn_err(CE_PANIC, "suspend: mach_descrip_update failed: %ld",
702                     rv);

704         sfmmu_ctxdoms_update();
705         sfmmu_ctxdoms_unlock();

707         /* Get new MD, update CPU mappings/relationships */
708         if (suspend_update_cpu_mappings)
709                 update_cpu_mappings();

711         DBG("suspend: target tick: 0x%lx", gettick_counter());
712         DBG("suspend: target stick: 0x%llx", gettick());
713         DBG("suspend: user %%tick/%%stick emulation is %d",
714             tick_stick_emulation_active);
715         DBG("suspend: finished");

717         return (0);
718 }
```

_____***unchanged_portion_omitted_***