```
**********************************************************
   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_
```

```
int
cpr_suspend_cpus(void)
{
        int     ret = 0;
        extern void *i_cpr_save_context(void *arg);

        mutex_enter(&cpu_lock);

        /*
         * the machine could not have booted without a bootcpu
         */
        ASSERT(i_cpr_bootcpu() != NULL);

        /*
         * bring all the offline cpus online
         */
        if ((ret = cpr_all_online())) {
                mutex_exit(&cpu_lock);
                return (ret);
        }

        /*
         * Set the affinity to be the boot processor
         * This is cleared in either cpr_resume_cpus() or cpr_unpause_cpus()
         */
        affinity_set(i_cpr_bootcpuid());

        ASSERT(CPU->cpu_id == 0);

        PMD(PMD_SX, ("curthread running on bootcpu\n"))

        /*
         * pause all other running CPUs and save the CPU state at the sametime
         */
        pause_cpus(NULL, i_cpr_save_context);
        cpu_pause_func = i_cpr_save_context;
        pause_cpus(NULL);

        mutex_exit(&cpu_lock);

        return (0);
}
```

______***unchanged_portion_omitted_***

```
void
cpr_resume_cpus(void)
{
        /*
         * this is a cut down version of start_other_cpus()
         * just do the initialization to wake the other cpus
         */

#if defined(__x86)
        /*
         * Initialize our syscall handlers
         */
        init_cpu_syscall(CPU);

#endif

        i_cpr_pre_resume_cpus();

        /*
         * Restart the paused cpus
         */
```

```
        mutex_enter(&cpu_lock);
        start_cpus();
        mutex_exit(&cpu_lock);

        i_cpr_post_resume_cpus();

        mutex_enter(&cpu_lock);
        /*
         * Restore this cpu to use the regular cpu_pause(), so that
         * online and offline will work correctly
         */
        cpu_pause_func = NULL;

        /*
         * clear the affinity set in cpr_suspend_cpus()
         */
        affinity_clear();

        /*
         * offline all the cpus that were brought online during suspend
         */
        cpr_restore_offline();

        mutex_exit(&cpu_lock);
}

void
cpr_unpause_cpus(void)
{
        /*
         * Now restore the system back to what it was before we suspended
         */

        PMD(PMD_SX, ("cpr_unpause_cpus: restoring system\n"))

        mutex_enter(&cpu_lock);

        /*
         * Restore this cpu to use the regular cpu_pause(), so that
         * online and offline will work correctly
         */
        cpu_pause_func = NULL;

        /*
         * Restart the paused cpus
         */
        start_cpus();

        /*
         * clear the affinity set in cpr_suspend_cpus()
         */
        affinity_clear();

        /*
         * offline all the cpus that were brought online during suspend
         */
        cpr_restore_offline();

        mutex_exit(&cpu_lock);
}

/*
 * Bring the system back up from a checkpoint, at this point
 * the VM has been minimally restored by boot, the following
 * are executed sequentially:
 *
 *    - machdep setup and enable interrupts (mp startup if it’s mp)
 *    - resume all devices
 *    - restart daemons
 *    - put all threads back on run queue
 */
static int
cpr_resume(int sleeptype)
{
        cpr_time_t pwron_tv, *ctp;
        char *str;
        int rc = 0;

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

        /*
         * Note:
         *
         * The rollback labels rb_xyz do not represent the cpr resume
         * state when event ’xyz’ has happened. Instead they represent
         * the state during cpr suspend when event ’xyz’ was being
         * entered (and where cpr suspend failed). The actual call that
         * failed may also need to be partially rolled back, since they
         * aren’t atomic in most cases.  In other words, rb_xyz means
         * "roll back all cpr suspend events that happened before ’xyz’,
         * and the one that caused the failure, if necessary."
         */
        switch (CPR->c_substate) {
#if defined(__sparc)
        case C_ST_DUMP:
                /*
                 * This is most likely a full-fledged cpr_resume after
                 * a complete and successful cpr suspend. Just roll back
                 * everything.
                 */
                ASSERT(sleeptype == CPR_TODISK);
                break;

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

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

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

```

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

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

        case C_ST_DISABLE_UFS_LOGGING:
                ASSERT(sleeptype == CPR_TODISK);
                goto rb_disable_ufs_logging;
#endif

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

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

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

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


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

rb_all:
        /*
         * perform platform-dependent initialization
         */
        if (cpr_suspend_succeeded)
                i_cpr_machdep_setup();

        /*
         * system did not really go down if we jump here
         */
rb_dump:
        /*
         * IMPORTANT:  SENSITIVE RESUME SEQUENCE
         *
         * DO NOT ADD ANY INITIALIZATION STEP BEFORE THIS POINT!!
         */
rb_nodump:
        /*
         * If we did suspend to RAM, we didn’t generate a dump
         */
        PMD(PMD_SX, ("cpr_resume: CPR DMA callback\n"))
        (void) callb_execute_class(CB_CL_CPR_DMA, CB_CODE_CPR_RESUME);
        if (cpr_suspend_succeeded) {
                PMD(PMD_SX, ("cpr_resume: CPR RPC callback\n"))
                (void) callb_execute_class(CB_CL_CPR_RPC, CB_CODE_CPR_RESUME);
        }

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

#endif

#if defined(__sparc)
        if (cpr_suspend_succeeded && (boothowto & RB_DEBUG))
                kdi_dvec_cpr_restart();
#endif


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

        if (cpr_suspend_succeeded) {
                cpr_resume_cpus();
        } else {
                cpr_unpause_cpus();
        }
#endif

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

        i_cpr_enable_intr();

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

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

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

        /*
         * statistics gathering
         */
        if (cpr_suspend_succeeded) {
                /*
                 * Prevent false alarm in tod_validate() due to tod
                 * value change between suspend and resume
                 */
                cpr_tod_status_set(TOD_CPR_RESUME_DONE);

                cpr_convert_promtime(&pwron_tv);

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

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

                CPR_STAT_EVENT_START_TMZ("Resume Total", &pwron_tv);

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

                str = "  read statefile";
                CPR_STAT_EVENT_START_TMZ(str, ctp);
                ctp = &cpr_term.tm_cprboot_end;
                CPR_STAT_EVENT_END_TMZ(str, ctp);
        }

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

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

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

#if defined(__x86)
        /*
         * If cpr_resume_uniproc is set, then pause all the other cpus
         * apart from the current cpu, so that broken drivers that think
         * that they are on a uniprocessor machine will resume
         */
        if (cpr_resume_uniproc) {
                mutex_enter(&cpu_lock);
                pause_cpus(NULL, NULL);
                pause_cpus(NULL);
                mutex_exit(&cpu_lock);
        }
#endif

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

        cpr_sae(0);

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

        if (rc) {
                if (CPR->c_substate == C_ST_DUMP ||
                    (sleeptype == CPR_TORAM &&
                    CPR->c_substate == C_ST_NODUMP)) {
                        if (cpr_test_point == FORCE_SUSPEND_TO_RAM) {
                                PMD(PMD_SX, ("cpr_resume: resume device "
                                    "warn\n"))
                                cpr_err(CE_WARN, str);
                        } else {
                                PMD(PMD_SX, ("cpr_resume: resume device "
                                    "panic\n"))
                                cpr_err(CE_PANIC, str);
                        }
                } else {
                        PMD(PMD_SX, ("cpr_resume: resume device warn\n"))
                        cpr_err(CE_WARN, str);
                }
        }

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

#if defined(__x86)
        /*
         * If cpr_resume_uniproc is set, then unpause all the processors
         * that were paused before resuming the drivers
         */
        if (cpr_resume_uniproc) {
                mutex_enter(&cpu_lock);
                start_cpus();
                mutex_exit(&cpu_lock);
        }
#endif

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

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

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

        if (sleeptype == CPR_TODISK) {
rb_statef_alloc:
                cpr_statef_close();

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

```
                 */
                (void) cpr_ufs_logging(1);
        }
#endif

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

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

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

rb_stop_user_threads:
        CPR_DEBUG(CPR_DEBUG1, "starting user threads...");
        PMD(PMD_SX, ("cpr_resume: starting user threads\n"))
        cpr_start_user_threads();
        CPR_DEBUG(CPR_DEBUG1, "done\n");
        /*
         * Ask Xorg to resume the frame buffer, and wait for it to happen
         */
        mutex_enter(&srn_clone_lock);
        if (srn_signal) {
                PMD(PMD_SX, ("cpr_suspend: (*srn_signal)(..., "
                    "SRN_NORMAL_RESUME)\n"))
                srn_inuse = 1;          /* because (*srn_signal) cv_waits */
                (*srn_signal)(SRN_TYPE_APM, SRN_NORMAL_RESUME);
                srn_inuse = 0;
        } else {
                PMD(PMD_SX, ("cpr_suspend: srn_signal NULL\n"))
        }
        mutex_exit(&srn_clone_lock);

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

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

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

        if (cpr_suspend_succeeded) {
                cpr_stat_record_events();
        }
```

```
#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_
```

```
/*
 * Promote PG above it's current parent.
 * This is only legal if PG has an equal or greater number of CPUs than its
 * parent.
 *
 * This routine operates on the CPU specific processor group data (for the CPUs
 * in the PG being promoted), and may be invoked from a context where one CPU's
 * PG data is under construction. In this case the argument "pgdata", if not
 * NULL, is a reference to the CPU's under-construction PG data.
 */
static void
cmt_hier_promote(pg_cmt_t *pg, cpu_pg_t *pgdata)
{
        pg_cmt_t        *parent;
        group_t         *children;
        cpu_t           *cpu;
        group_iter_t    iter;
        pg_cpu_itr_t    cpu_iter;
        int             r;
        int             err;
        int             nchildren;

        ASSERT(MUTEX_HELD(&cpu_lock));

        parent = pg->cmt_parent;
        if (parent == NULL) {
                /*
                 * Nothing to do
                 */
                return;
        }

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

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

        /*
         * If necessary, update the parent's sibling set, replacing parent
         * with PG.
         */
        if (parent->cmt_siblings) {
                if (group_remove(parent->cmt_siblings, parent, GRP_NORESIZE)
                    != -1) {
                        r = group_add(parent->cmt_siblings, pg, GRP_NORESIZE);
                        ASSERT(r != -1);
                }
        }

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

```
                if (group_remove(&cmt_root->cl_pgs, parent, GRP_NORESIZE)
                    != -1) {
                        r = group_add(&cmt_root->cl_pgs, pg, GRP_NORESIZE);
                        ASSERT(r != -1);
                }
        }

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

        children = pg->cmt_children;
        pg->cmt_children = parent->cmt_children;
        parent->cmt_children = children;

        nchildren = pg->cmt_nchildren;
        pg->cmt_nchildren = parent->cmt_nchildren;
        parent->cmt_nchildren = nchildren;

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

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

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

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

                /*
                 * Update the CMT load balancing lineage
                 */
                if ((idx = group_find(&pgd->cmt_pgs, (void *)pg)) == -1) {
                        /*
                         * Unless this is the CPU who's lineage is being
                         * constructed, the PG being promoted should be
                         * in the lineage.
                         */
                        ASSERT(pg_cpu_is_bootstrapped(cpu));
                        continue;
                }

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

                /*
                 * Have the child and the parent swap places in the CPU's
                 * lineage
                 */
                group_remove_at(&pgd->cmt_pgs, idx);
                group_remove_at(&pgd->cmt_pgs, idx - 1);
                err = group_add_at(&pgd->cmt_pgs, parent, idx);
                ASSERT(err == 0);
                err = group_add_at(&pgd->cmt_pgs, pg, idx - 1);
                ASSERT(err == 0);

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

        /*
         * Update the parent references for PG and it's parent
         */
        pg->cmt_parent = parent->cmt_parent;
        parent->cmt_parent = pg;

        start_cpus();
}
______unchanged_portion_omitted_

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

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

```

        /*
         * Inform pghw layer that this PG is pruned.
         */
        pghw_cmt_fini((pghw_t *)pg_bad);

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

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

        /*
         * Find and eliminate the PG from the lineage.
         */
        for (i = 0; i < size; i++) {
                if (lineage[i] == pg_bad) {
                        for (j = i; j < size - 1; j++)
                                lineage[j] = lineage[j + 1];
                        *sz = size - 1;
                        break;
                }
        }

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

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

        /*
         * For each of the PGs being pruned, ensure sufficient capacity in
         * the siblings set for the PG's children
         */
        group_iter_init(&hw_iter);
        while ((pg = group_iterate(hwset, &hw_iter)) != NULL) {
                /*
                 * PG is being pruned, but if it is bringing up more than
                 * one child, ask for more capacity in the siblings group.
                 */
                cap_needed = 0;
                if (pg->cmt_children &&
                    GROUP_SIZE(pg->cmt_children) > 1) {
                        cap_needed = GROUP_SIZE(pg->cmt_children) - 1;

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

                        /*
                         * If this is a top level group, also ensure the
                         * capacity in the root lgrp level CMT grouping.
                         */
                        if (pg->cmt_parent == NULL &&
                            pg->cmt_siblings != &cmt_root->cl_pgs) {
                                group_expand(&cmt_root->cl_pgs,
                                    GROUP_SIZE(&cmt_root->cl_pgs) + cap_needed);
                                cmt_root->cl_npgs += cap_needed;
                        }
                }
        }

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

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

                /*
                 * Remove PG from it's group of siblings, if it's there.
                 */
                if (pg->cmt_siblings) {
                        (void) group_remove(pg->cmt_siblings, pg, GRP_NORESIZE);
                }
                if (pg->cmt_parent == NULL &&
                    pg->cmt_siblings != &cmt_root->cl_pgs) {
                        (void) group_remove(&cmt_root->cl_pgs, pg,
                            GRP_NORESIZE);
                }

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

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

                        group_iter_init(&child_iter);
                        while ((child = group_iterate(children, &child_iter))
                            != NULL) {
                                if (pg->cmt_siblings != NULL) {
                                        r = group_add(pg->cmt_siblings, child,
                                            GRP_NORESIZE);
                                        ASSERT(r == 0);

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

```
                                            &cmt_root->cl_pgs) {
                                                r = group_add(&cmt_root->cl_pgs,
                                                    child, GRP_NORESIZE);
                                                ASSERT(r == 0);
                                        }
                                }
                        }
                        group_empty(pg->cmt_children);
                }

                /*
                 * Reset the callbacks to the defaults
                 */
                pg_callback_set_defaults((pg_t *)pg);

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

                        /*
                         * The CPU's lineage is under construction still
                         * references the bootstrap CPU PG data structure.
                         */
                        if (pg_cpu_is_bootstrapped(cpu))
                                cpd = pgdata;
                        else
                                cpd = cpu->cpu_pg;

                        /*
                         * Iterate over the CPU's PGs updating the children
                         * of the PG being promoted, since they have a new
                         * parent and siblings set.
                         */
                        group_iter_init(&liter);
                        while ((cpu_pg = group_iterate(&cpd->pgs,
                            &liter)) != NULL) {
                                if (cpu_pg->cmt_parent == pg) {
                                        cpu_pg->cmt_parent = pg->cmt_parent;
                                        cpu_pg->cmt_siblings = pg->cmt_siblings;
                                }
                        }

                        /*
                         * Update the CPU's lineages
                         *
                         * Remove the PG from the CPU's group used for CMT
                         * scheduling.
                         */
                        (void) group_remove(&cpd->cmt_pgs, pg, GRP_NORESIZE);
                }
        }
        start_cpus();
        return (0);
}

/*
 * Disable CMT scheduling
 */
static void
pg_cmt_disable(void)
{
        cpu_t           *cpu;

        ASSERT(MUTEX_HELD(&cpu_lock));

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

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

        cmt_sched_disabled = 1;
        start_cpus();
        cmn_err(CE_NOTE, "!CMT thread placement optimizations unavailable");
}
______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_
```

```
static int
cpupart_move_cpu(cpu_t *cp, cpupart_t *newpp, int forced)
{
        cpupart_t *oldpp;
        cpu_t   *ncp, *newlist;
        kthread_t *t;
        int     move_threads = 1;
        lgrp_id_t lgrpid;
        proc_t  *p;
        int lgrp_diff_lpl;
        lpl_t   *cpu_lpl;
        int     ret;
        boolean_t unbind_all_threads = (forced != 0);

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

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

        if (newpp == oldpp) {
                /*
                 * Don’t need to do anything.
                 */
                return (0);
        }

        cpu_state_change_notify(cp->cpu_id, CPU_CPUPART_OUT);

        if (!disp_bound_partition(cp, 0)) {
                /*
                 * Don’t need to move threads if there are no threads in
                 * the partition.  Note that threads can’t enter the
                 * partition while we’re holding cpu_lock.
                 */
                move_threads = 0;
        } else if (oldpp->cp_ncpus == 1) {
                /*
                 * The last CPU is removed from a partition which has threads
                 * running in it. Some of these threads may be bound to this
                 * CPU.
                 *
                 * Attempt to unbind threads from the CPU and from the processor
                 * set. Note that no threads should be bound to this CPU since
                 * cpupart_move_threads will refuse to move bound threads to
                 * other CPUs.
                 */
                (void) cpu_unbind(oldpp->cp_cpulist->cpu_id, B_FALSE);
                (void) cpupart_unbind_threads(oldpp, B_FALSE);

                if (!disp_bound_partition(cp, 0)) {
                        /*
                         * No bound threads in this partition any more
                         */
                        move_threads = 0;
                } else {
                        /*
```

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

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

        /*
         * Stop further threads weak binding to this cpu.
         */
        cpu_inmotion = cp;
        membar_enter();

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

again:
        if (move_threads) {
                int loop_count;
                /*
                 * Check for threads strong or weak bound to this CPU.
                 */
                for (loop_count = 0; disp_bound_threads(cp, 0); loop_count++) {
                        if (loop_count >= 5) {
                                cpu_state_change_notify(cp->cpu_id,
                                    CPU_CPUPART_IN);
                                pg_cpupart_out(cp, newpp);
                                pg_cpupart_in(cp, oldpp);
                                cpu_inmotion = NULL;
                                return (EBUSY); /* some threads still bound */
                        }
                        delay(1);
                }
        }

        /*
         * Before we actually start changing data structures, notify
         * the cyclic subsystem that we want to move this CPU out of its
         * partition.
         */
        if (!cyclic_move_out(cp)) {
                /*
                 * This CPU must be the last CPU in a processor set with
                 * a bound cyclic.
                 */
                cpu_state_change_notify(cp->cpu_id, CPU_CPUPART_IN);
                pg_cpupart_out(cp, newpp);
                pg_cpupart_in(cp, oldpp);
                cpu_inmotion = NULL;
                return (EBUSY);
        }
        pause_cpus(cp, NULL);
        pause_cpus(cp);

        if (move_threads) {
                /*
                 * The thread on cpu before the pause thread may have read
                 * cpu_inmotion before we raised the barrier above.  Check
                 * again.
                 */
                if (disp_bound_threads(cp, 1)) {
                        start_cpus();
                        goto again;
                }

        }

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

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

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

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

                ncp = cp->cpu_prev_part->cpu_next_part = cp->cpu_next_part;
                cp->cpu_next_part->cpu_prev_part = cp->cpu_prev_part;
                if (oldpp->cp_cpulist == cp) {
                        oldpp->cp_cpulist = ncp;
                }
        } else {
                ncp = oldpp->cp_cpulist = NULL;
                cp_numparts_nonempty--;
                ASSERT(cp_numparts_nonempty != 0);
        }
        oldpp->cp_gen++;

        /* move into new partition */
        newlist = newpp->cp_cpulist;
        if (newlist == NULL) {
                newpp->cp_cpulist = cp->cpu_next_part = cp->cpu_prev_part = cp;
                cp_numparts_nonempty++;
                ASSERT(cp_numparts_nonempty != 0);
        } else {
                cp->cpu_next_part = newlist;
                cp->cpu_prev_part = newlist->cpu_prev_part;
                newlist->cpu_prev_part->cpu_next_part = cp;
                newlist->cpu_prev_part = cp;
        }
        cp->cpu_part = newpp;
        newpp->cp_ncpus++;
        newpp->cp_gen++;

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

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

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

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

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

                        t = p->p_tlist;

                        if (t == NULL)
                                continue;

                        lgrp_diff_lpl = 0;

                        do {

                                ASSERT(t->t_lpl != NULL);

                                /*
                                 * Update the count of how many threads are
                                 * in this CPU’s lgroup but have a different lpl
                                 */

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

                                if (!LGRP_CPUS_IN_PART(t->t_lpl->lpl_lgrpid,
                                    t->t_cpupart)) {
                                        lgrp_move_thread(t,
                                            lgrp_choose(t, t->t_cpupart), 0);
                                }

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

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

                                /* Update CPU last ran on if it was this CPU */
                                if (t->t_cpu == cp && t->t_cpupart == oldpp &&
                                    t->t_bound_cpu != cp) {
                                        t->t_cpu = disp_lowpri_cpu(ncp,
                                            t->t_lpl, t->t_pri, NULL);
                                }
                                t = t->t_forw;
                        } while (t != p->p_tlist);

                        /*
                         * Didn’t find any threads in the same lgroup as this
                         * CPU with a different lpl, so remove the lgroup from
                         * the process lgroup bitmask.
                         */

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

                /*
                 * Walk thread list looking for threads that need to be
                 * rehomed, since there are some threads that are not in
                 * their process’s p_tlist.
                 */

                t = curthread;

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

                        /*
                         * If the lgroup that t is assigned to no
                         * longer has any CPUs in t’s partition,
                         * we’ll have to choose a new lgroup for t.
                         * Also, choose best lgroup for home when
                         * thread has specified lgroup affinities,
                         * since there may be an lgroup with more
                         * affinity available after moving CPUs
                         * around.
                         */
                        if (!LGRP_CPUS_IN_PART(t->t_lpl->lpl_lgrpid,
                            t->t_cpupart) || t->t_lgrp_affinity) {
                                lgrp_move_thread(t,
                                    lgrp_choose(t, t->t_cpupart), 1);
                        }

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

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

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

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

                /*
                 * Clear off the CPU’s run queue, and the kp queue if the
                 * partition is now empty.
                 */
                disp_cpu_inactive(cp);

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

```
                 */
                cp->cpu_runrun = 1;
                cp->cpu_kprunrun = 1;
        }

        cpu_inmotion = NULL;
        start_cpus();

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

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

        return (0);
}
```

______***unchanged_portion_omitted_***

```
/*
 * Create a new partition.  On MP systems, this also allocates a
 * kpreempt disp queue for that partition.
 */
int
cpupart_create(psetid_t *psid)
{
        cpupart_t       *pp;

        ASSERT(pool_lock_held());

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

        mutex_enter(&cpu_lock);
        if (cp_numparts == cp_max_numparts) {
                mutex_exit(&cpu_lock);
                kmem_free(pp->cp_lgrploads, sizeof (lpl_t) * pp->cp_nlgrploads);
                pp->cp_lgrploads = NULL;
                kmem_free(pp, sizeof (cpupart_t));
                return (ENOMEM);
        }
        cp_numparts++;
        /* find the next free partition ID */
        while (cpupart_find(CPTOPS(cp_id_next)) != NULL)
                cp_id_next++;
        pp->cp_id = cp_id_next++;
        pp->cp_ncpus = 0;
        pp->cp_cpulist = NULL;
        pp->cp_attr = 0;
        klgrpset_clear(pp->cp_lgrpset);
        pp->cp_kp_queue.disp_maxrunpri = -1;
        pp->cp_kp_queue.disp_max_unbound_pri = -1;
        pp->cp_kp_queue.disp_cpu = NULL;
        pp->cp_gen = 0;
        DISP_LOCK_INIT(&pp->cp_kp_queue.disp_lock);
        *psid = CPTOPS(pp->cp_id);
        disp_kp_alloc(&pp->cp_kp_queue, v.v_nglobpris);
        cpupart_kstat_create(pp);
        cpupart_lpl_initialize(pp);
        bitset_init(&pp->cp_cmt_pgs);

        /*
         * Initialize and size the partition’s bitset of halted CPUs.
         */
        bitset_init_fanout(&pp->cp_haltset, cp_haltset_fanout);
        bitset_resize(&pp->cp_haltset, max_ncpus);

        /*
         * Pause all CPUs while changing the partition list, to make sure
         * the clock thread (which traverses the list without holding
         * cpu_lock) isn’t running.
         */
        pause_cpus(NULL, NULL);
        pause_cpus(NULL);
        pp->cp_next = cp_list_head;
        pp->cp_prev = cp_list_head->cp_prev;
        cp_list_head->cp_prev->cp_next = pp;
        cp_list_head->cp_prev = pp;
        start_cpus();
        mutex_exit(&cpu_lock);

        return (0);
}
```

______***unchanged_portion_omitted_***

```
/*
 * Destroy a partition.
 */
int
cpupart_destroy(psetid_t psid)
{
        cpu_t   *cp, *first_cp;
        cpupart_t *pp, *newpp;
        int     err = 0;

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

        pp = cpupart_find(psid);
        if (pp == NULL || pp == &cp_default) {
                mutex_exit(&cpu_lock);
                return (EINVAL);
        }

        /*
         * Unbind all the threads currently bound to the partition.
         */
        err = cpupart_unbind_threads(pp, B_TRUE);
        if (err) {
                mutex_exit(&cpu_lock);
                return (err);
        }

        newpp = &cp_default;
        while ((cp = pp->cp_cpulist) != NULL) {
                if (err = cpupart_move_cpu(cp, newpp, 0)) {
                        mutex_exit(&cpu_lock);
                        return (err);
                }
        }

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

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

        /*
         * Reset the pointers in any offline processors so they won’t
         * try to rejoin the destroyed partition when they’re turned
         * online.
         */
        first_cp = cp = CPU;
        do {
                if (cp->cpu_part == pp) {
                        ASSERT(cp->cpu_flags & CPU_OFFLINE);
                        cp->cpu_part = newpp;
                }
                cp = cp->cpu_next;
        } while (cp != first_cp);

        /*
         * Pause all CPUs while changing the partition list, to make sure
         * the clock thread (which traverses the list without holding
         * cpu_lock) isn’t running.
         */
        pause_cpus(NULL, NULL);
        pause_cpus(NULL);
        pp->cp_prev->cp_next = pp->cp_next;
        pp->cp_next->cp_prev = pp->cp_prev;
        if (cp_list_head == pp)
                cp_list_head = pp->cp_next;
        start_cpus();

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

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

        cp_numparts--;

        disp_kp_free(&pp->cp_kp_queue);

        cpupart_lpl_teardown(pp);

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

        return (err);
}
```

______***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_
```

```
/*
 * For each CPU, allocate new dispatch queues
 * with the stated number of priorities.
 */
static void
cpu_dispqalloc(int numpris)
{
        cpu_t   *cpup;
        struct disp_queue_info  *disp_mem;
        int i, num;

        ASSERT(MUTEX_HELD(&cpu_lock));

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

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

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

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

        kmem_free(disp_mem, NCPU * sizeof (struct disp_queue_info));
}
______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.
 */
```

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

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

int ncpus = 1;
int ncpus_online = 1;

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

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

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

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

static kmutex_t pause_free_mutex;
static kcondvar_t pause_free_cv;

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


static struct cpu_sys_stats_ks_data {
        kstat_named_t cpu_ticks_idle;
        kstat_named_t cpu_ticks_user;
        kstat_named_t cpu_ticks_kernel;
        kstat_named_t cpu_ticks_wait;
        kstat_named_t cpu_nsec_idle;
        kstat_named_t cpu_nsec_user;
        kstat_named_t cpu_nsec_kernel;
        kstat_named_t cpu_nsec_dtrace;
        kstat_named_t cpu_nsec_intr;
        kstat_named_t cpu_load_intr;
        kstat_named_t wait_ticks_io;
        kstat_named_t dtrace_probes;
        kstat_named_t bread;
```

```
        kstat_named_t bwrite;
        kstat_named_t lread;
        kstat_named_t lwrite;
        kstat_named_t phread;
        kstat_named_t phwrite;
        kstat_named_t pswitch;
        kstat_named_t trap;
        kstat_named_t intr;
        kstat_named_t syscall;
        kstat_named_t sysread;
        kstat_named_t syswrite;
        kstat_named_t sysfork;
        kstat_named_t sysvfork;
        kstat_named_t sysexec;
        kstat_named_t readch;
        kstat_named_t writech;
        kstat_named_t rcvint;
        kstat_named_t xmtint;
        kstat_named_t mdmint;
        kstat_named_t rawch;
        kstat_named_t canch;
        kstat_named_t outch;
        kstat_named_t msg;
        kstat_named_t sema;
        kstat_named_t namei;
        kstat_named_t ufsiget;
        kstat_named_t ufsdirblk;
        kstat_named_t ufsipage;
        kstat_named_t ufsinopage;
        kstat_named_t procovf;
        kstat_named_t intrthread;
        kstat_named_t intrblk;
        kstat_named_t intrunpin;
        kstat_named_t idlethread;
        kstat_named_t inv_swtch;
        kstat_named_t nthreads;
        kstat_named_t cpumigrate;
        kstat_named_t xcalls;
        kstat_named_t mutex_adenters;
        kstat_named_t rw_rdfails;
        kstat_named_t rw_wrfails;
        kstat_named_t modload;
        kstat_named_t modunload;
        kstat_named_t bawrite;
        kstat_named_t iowait;
} cpu_sys_stats_ks_data_template = {
______unchanged_portion_omitted_

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

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

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

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

                mach_cpu_pause(safe);

                splx(s);
                /*
                 * Waiting is at an end. Switch out of cpu_pause
                 * loop and resume useful work.
                 */
                swtch();
        }

        mutex_enter(&pause_free_mutex);
        *safe = PAUSE_DEAD;
        cv_broadcast(&pause_free_cv);
        mutex_exit(&pause_free_mutex);
}
______unchanged_portion_omitted_


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

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

        ASSERT(MUTEX_HELD(&cpu_lock));
        ASSERT(cpi->cp_paused == NULL);
        cpi->cp_count = 0;
        cpi->cp_go = 0;
        for (i = 0; i < NCPU; i++)
                safe_list[i] = PAUSE_IDLE;
        kpreempt_disable();

        cpi->cp_func = func;

#endif /* ! codereview */
        /*
         * If running on the cpu that is going offline, get off it.
         * This is so that it won’t be necessary to rechoose a CPU
         * when done.
         */
        if (CPU == off_cp)
                cpu_id = off_cp->cpu_next_part->cpu_id;
        else
                cpu_id = CPU->cpu_id;
        affinity_set(cpu_id);

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

        /*
         * Now wait for all CPUs to be running the pause thread.
         */
        while (cpi->cp_count > 0) {
                /*
                 * Spin reading the count without grabbing the disp
                 * lock to make sure we don’t prevent the pause
                 * threads from getting the lock.
                 */
                while (sema_held(&cpi->cp_sem))
                        ;
                if (sema_tryp(&cpi->cp_sem))
                        --cpi->cp_count;
        }
        cpi->cp_go = 1;                 /* all have reached cpu_pause */

        /*
         * Now wait for all CPUs to spl. (Transition from PAUSE_READY
         * to PAUSE_WAIT.)
         */
        for (i = 0; i < NCPU; i++) {
                while (safe_list[i] != PAUSE_WAIT)
                        ;
        }
        cpi->cp_spl = splhigh();        /* block dispatcher on this CPU */
        cpi->cp_paused = curthread;
}

/*
 * Check whether the current thread has CPUs paused
 */
int
cpus_paused(void)
{
        if (cpu_pause_info.cp_paused != NULL) {
                ASSERT(cpu_pause_info.cp_paused == curthread);
                return (1);
        }
        return (0);
}

static cpu_t *
cpu_get_all(processorid_t cpun)
{
        ASSERT(MUTEX_HELD(&cpu_lock));

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

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

        ASSERT(MUTEX_HELD(&cpu_lock));
        c = cpu_get_all(cpun);
        if (c != NULL && !INGLOBALZONE(curproc) && pool_pset_enabled() &&
            zone_pset_get(curproc->p_zone) != cpupart_query_cpu(c))
                return (NULL);
        return (c);
}

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

/*
 * Determine whether the CPU is online and handling interrupts.
 */
int
cpu_is_online(cpu_t *cpu)
{
        ASSERT(MUTEX_HELD(&cpu_lock));
        return (cpu_flagged_online(cpu->cpu_flags));
}
```

```

/*
 * Determine whether the CPU is offline (this includes spare and faulted).
 */
int
cpu_is_offline(cpu_t *cpu)
{
        ASSERT(MUTEX_HELD(&cpu_lock));
        return (cpu_flagged_offline(cpu->cpu_flags));
}

/*
 * Determine whether the CPU is powered off.
 */
int
cpu_is_poweredoff(cpu_t *cpu)
{
        ASSERT(MUTEX_HELD(&cpu_lock));
        return (cpu_flagged_poweredoff(cpu->cpu_flags));
}

/*
 * Determine whether the CPU is handling interrupts.
 */
int
cpu_is_nointr(cpu_t *cpu)
{
        ASSERT(MUTEX_HELD(&cpu_lock));
        return (cpu_flagged_nointr(cpu->cpu_flags));
}

/*
 * Determine whether the CPU is active (scheduling threads).
 */
int
cpu_is_active(cpu_t *cpu)
{
        ASSERT(MUTEX_HELD(&cpu_lock));
        return (cpu_flagged_active(cpu->cpu_flags));
}

/*
 * Same as above, but these require cpu_flags instead of cpu_t pointers.
 */
int
cpu_flagged_online(cpu_flag_t cpu_flags)
{
        return (cpu_flagged_active(cpu_flags) &&
            (cpu_flags & CPU_ENABLE));
}

int
cpu_flagged_offline(cpu_flag_t cpu_flags)
{
        return (((cpu_flags & CPU_POWEROFF) == 0) &&
            ((cpu_flags & (CPU_READY | CPU_OFFLINE)) != CPU_READY));
}

int
cpu_flagged_poweredoff(cpu_flag_t cpu_flags)
{
        return ((cpu_flags & CPU_POWEROFF) == CPU_POWEROFF);
}

int
cpu_flagged_nointr(cpu_flag_t cpu_flags)
```

```
{
        return (cpu_flagged_active(cpu_flags) &&
            (cpu_flags & CPU_ENABLE) == 0);
}

int
cpu_flagged_active(cpu_flag_t cpu_flags)
{
        return (((cpu_flags & (CPU_POWEROFF | CPU_FAULTED | CPU_SPARE)) == 0) &&
            ((cpu_flags & (CPU_READY | CPU_OFFLINE)) == CPU_READY));
}

/*
 * Bring the indicated CPU online.
 */
int
cpu_online(cpu_t *cp)
{
        int     error = 0;

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

        ASSERT(MUTEX_HELD(&cpu_lock));

        /*
         * Put all the cpus into a known safe place.
         * No mutexes can be entered while CPUs are paused.
         */
        error = mp_cpu_start(cp);       /* arch-dep hook */
        if (error == 0) {
                pg_cpupart_in(cp, cp->cpu_part);
                pause_cpus(NULL, NULL);
                pause_cpus(NULL);
                cpu_add_active_internal(cp);
                if (cp->cpu_flags & CPU_FAULTED) {
                        cp->cpu_flags &= ~CPU_FAULTED;
                        mp_cpu_faulted_exit(cp);
                }
                cp->cpu_flags &= ~(CPU_QUIESCED | CPU_OFFLINE | CPU_FROZEN |
                    CPU_SPARE);
                CPU_NEW_GENERATION(cp);
                start_cpus();
                cpu_stats_kstat_create(cp);
                cpu_create_intrstat(cp);
                lgrp_kstat_create(cp);
                cpu_state_change_notify(cp->cpu_id, CPU_ON);
                cpu_intr_enable(cp);    /* arch-dep hook */
                cpu_state_change_notify(cp->cpu_id, CPU_INTR_ON);
                cpu_set_state(cp);
                cyclic_online(cp);
                /*
                 * This has to be called only after cyclic_online(). This
                 * function uses cyclics.
                 */
                callout_cpu_online(cp);
                poke_cpu(cp->cpu_id);
        }

        return (error);
}
```

```
/*
 * Take the indicated CPU offline.
 */
int
cpu_offline(cpu_t *cp, int flags)
{
        cpupart_t *pp;
        int     error = 0;
        cpu_t   *ncp;
        int     intr_enable;
        int     cyclic_off = 0;
        int     callout_off = 0;
        int     loop_count;
        int     no_quiesce = 0;
        int     (*bound_func)(struct cpu *, int);
        kthread_t *t;
        lpl_t   *cpu_lpl;
        proc_t  *p;
        int     lgrp_diff_lpl;
        boolean_t unbind_all_threads = (flags & CPU_FORCED) != 0;

        ASSERT(MUTEX_HELD(&cpu_lock));

        /*
         * If we’re going from faulted or spare to offline, just
         * clear these flags and update CPU state.
         */
        if (cp->cpu_flags & (CPU_FAULTED | CPU_SPARE)) {
                if (cp->cpu_flags & CPU_FAULTED) {
                        cp->cpu_flags &= ~CPU_FAULTED;
                        mp_cpu_faulted_exit(cp);
                }
                cp->cpu_flags &= ~CPU_SPARE;
                cpu_set_state(cp);
                return (0);
        }

        /*
         * Handle off-line request.
         */
        pp = cp->cpu_part;
        /*
         * Don’t offline last online CPU in partition
         */
        if (ncpus_online <= 1 || pp->cp_ncpus <= 1 || cpu_intr_count(cp) < 2)
                return (EBUSY);
        /*
         * Unbind all soft-bound threads bound to our CPU and hard bound threads
         * if we were asked to.
         */
        error = cpu_unbind(cp->cpu_id, unbind_all_threads);
        if (error != 0)
                return (error);
        /*
         * We shouldn’t be bound to this CPU ourselves.
         */
        if (curthread->t_bound_cpu == cp)
                return (EBUSY);

        /*
         * Tell interested parties that this CPU is going offline.
         */
        CPU_NEW_GENERATION(cp);
        cpu_state_change_notify(cp->cpu_id, CPU_OFF);
        /*
         * Tell the PG subsystem that the CPU is leaving the partition
         */
        pg_cpupart_out(cp, pp);

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

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

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

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

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

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

        if (error == 0 && callout_off == 0) {
                callout_cpu_offline(cp);
                callout_off = 1;
        }

        if (error == 0 && cyclic_off == 0) {
                if (!cyclic_offline(cp)) {
                        /*
                         * We must have bound cyclics...
                         */
                        error = EBUSY;
                        goto out;
                }
                cyclic_off = 1;
        }

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

        /*
         * If that all worked, take the CPU offline and decrement
         * ncpus_online.
         */
        if (error == 0) {
                /*
                 * Put all the cpus into a known safe place.
                 * No mutexes can be entered while CPUs are paused.
                 */
                pause_cpus(cp, NULL);
                pause_cpus(cp);
                /*
                 * Repeat the operation, if necessary, to make sure that
                 * all outstanding low-level interrupts run to completion
                 * before we set the CPU_QUIESCED flag.  It's also possible
                 * that a thread has weak bound to the cpu despite our raising
                 * cpu_inmotion above since it may have loaded that
                 * value before the barrier became visible (this would have
                 * to be the thread that was on the target cpu at the time
                 * we raised the barrier).
                 */
                if ((!no_quiesce && cp->cpu_intr_actv != 0) ||
                    (*bound_func)(cp, 1)) {
                        start_cpus();
                        (void) mp_cpu_start(cp);
                        goto again;
                }
                ncp = cp->cpu_next_part;
                cpu_lpl = cp->cpu_lpl;
                ASSERT(cpu_lpl != NULL);

                /*
                 * Remove the CPU from the list of active CPUs.
                 */
                cpu_remove_active(cp);
                /*
                 * Walk the active process list and look for threads
                 * whose home lgroup needs to be updated, or
                 * the last CPU they run on is the one being offlined now.
                 */

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

                        t = p->p_tlist;

                        if (t == NULL)
                                continue;

                        lgrp_diff_lpl = 0;

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

                                if (cpu_lpl->lpl_ncpu == 0) {
                                        if (t->t_lpl == cpu_lpl)
                                                lgrp_move_thread(t,
                                                    lgrp_choose(t,
                                                    t->t_cpupart), 0);
                                        else if (t->t_lpl->lpl_lgrpid ==
                                            cpu_lpl->lpl_lgrpid)
                                                lgrp_diff_lpl++;
                                }
                                ASSERT(t->t_lpl->lpl_ncpu > 0);

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

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

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

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

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

                t = curthread;
```

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

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

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

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

                        /*
                         * Update CPU last ran on if it was this CPU
                         */

                        if (t->t_cpu == cp && t->t_bound_cpu != cp) {
                                t->t_cpu = disp_lowpri_cpu(ncp,
                                    t->t_lpl, t->t_pri, NULL);
                        }
                        ASSERT(t->t_cpu != cp || t->t_bound_cpu == cp ||
                            t->t_weakbound_cpu == cp);
                        t = t->t_next;

                } while (t != curthread);
                ASSERT((cp->cpu_flags & (CPU_FAULTED | CPU_SPARE)) == 0);
                cp->cpu_flags |= CPU_OFFLINE;
                disp_cpu_inactive(cp);
                if (!no_quiesce)
                        cp->cpu_flags |= CPU_QUIESCED;
                ncpus_online--;
                cpu_set_state(cp);
                cpu_inmotion = NULL;
                start_cpus();
                cpu_stats_kstat_destroy(cp);
                cpu_delete_intrstat(cp);
                lgrp_kstat_destroy(cp);
        }

out:
        cpu_inmotion = NULL;

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

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

        /*
         * If we failed, but managed to offline callouts on this CPU,
         * bring it back online.
         */
        if (error && callout_off)
                callout_cpu_online(cp);
        /*
         * If we failed, tell the PG subsystem that the CPU is back
         */
        pg_cpupart_in(cp, pp);

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

        return (error);
}
```

______***unchanged_portion_omitted_***

```
/*
 * Insert a CPU into the list of available CPUs.
 */
void
cpu_add_unit(cpu_t *cp)
{
        int seqid;

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

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

        /*
         * Note: most users of the cpu_list will grab the
         * cpu_lock to insure that it isn't modified.  However,
         * certain users can't or won't do that.  To allow this
         * we pause the other cpus.  Users who walk the list
         * without cpu_lock, must disable kernel preemption
         * to insure that the list isn't modified underneath
         * them.  Also, any cached pointers to cpu structures
         * must be revalidated by checking to see if the
         * cpu_next pointer points to itself.  This check must
         * be done with the cpu_lock held or kernel preemption
         * disabled.  This check relies upon the fact that
         * old cpu structures are not free'ed or cleared after
         * then are removed from the cpu_list.
         *
         * Note that the clock code walks the cpu list dereferencing
         * the cpu_part pointer, so we need to initialize it before
         * adding the cpu to the list.
         */
        cp->cpu_part = &cp_default;
        (void) pause_cpus(NULL, NULL);
        (void) pause_cpus(NULL);
        cp->cpu_next = cpu_list;
        cp->cpu_prev = cpu_list->cpu_prev;
        cpu_list->cpu_prev->cpu_next = cp;
        cpu_list->cpu_prev = cp;
        start_cpus();

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

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

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

        /*
         * allocate a pause thread for this CPU.
         */
        cpu_pause_alloc(cp);

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

        init_cpu_mstate(cp, CMS_SYSTEM);

        pool_pset_mod = gethrtime();
}

/*
 * Do the opposite of cpu_add_unit().
 */
void
cpu_del_unit(int cpuid)
{
        struct cpu      *cp, *cpnext;

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

        ASSERT(cp->cpu_next_onln == cp);
        ASSERT(cp->cpu_prev_onln == cp);
        ASSERT(cp->cpu_next_part == cp);
        ASSERT(cp->cpu_prev_part == cp);

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

        /*
         * Destroy kstat stuff.
         */
        cpu_info_kstat_destroy(cp);
        term_cpu_mstate(cp);
        /*
         * Free up pause thread.
         */
        cpu_pause_free(cp);
        CPUSET_DEL(cpu_available, cp->cpu_id);
        cpu[cp->cpu_id] = NULL;
        cpu_seq[cp->cpu_seqid] = NULL;
        /*
         * The clock thread and mutex_vector_enter cannot hold the
         * cpu_lock while traversing the cpu list, therefore we pause
         * all other threads by pausing the other cpus. These, and any
         * other routines holding cpu pointers while possibly sleeping
         * must be sure to call kpreempt_disable before processing the
         * list and be sure to check that the cpu has not been deleted
         * after any sleeps (check cp->cpu_next != NULL). We guarantee
         * to keep the deleted cpu structure around.
         *
         * Note that this MUST be done AFTER cpu_available
         * has been updated so that we don't waste time
         * trying to pause the cpu we're trying to delete.
         */
        (void) pause_cpus(NULL, NULL);
        (void) pause_cpus(NULL);

        cpnext = cp->cpu_next;
        cp->cpu_prev->cpu_next = cp->cpu_next;
        cp->cpu_next->cpu_prev = cp->cpu_prev;
        if (cp == cpu_list)
                cpu_list = cpnext;

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

        start_cpus();

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

        pool_pset_mod = gethrtime();
}
______unchanged_portion_omitted_

/*
 * Add a CPU to the list of active CPUs.
 *      This is called from machine-dependent layers when a new CPU is started.
 */
void
cpu_add_active(cpu_t *cp)
{
        pg_cpupart_in(cp, cp->cpu_part);

        pause_cpus(NULL, NULL);
        pause_cpus(NULL);
        cpu_add_active_internal(cp);
        start_cpus();

        cpu_stats_kstat_create(cp);
        cpu_create_intrstat(cp);
        lgrp_kstat_create(cp);
        cpu_state_change_notify(cp->cpu_id, CPU_INIT);
}
______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_

static void
cpu_idle_insert_callback(cpu_idle_cb_impl_t *cip)
{
        int unlock = 0, unpause = 0;
        int i, cnt_new = 0, cnt_old = 0;
        char *buf_new = NULL, *buf_old = NULL;

        ASSERT(MUTEX_HELD(&cpu_idle_cb_lock));

        /*
         * Expand array if it’s full.
         * Memory must be allocated out of pause/start_cpus() scope because
         * kmem_zalloc() can’t be called with KM_SLEEP flag within that scope.
         */
        if (cpu_idle_cb_curr == cpu_idle_cb_max) {
                cnt_new = cpu_idle_cb_max + CPU_IDLE_ARRAY_CAPACITY_INC;
                buf_new = (char *)kmem_zalloc(cnt_new *
                    sizeof (cpu_idle_cb_item_t), KM_SLEEP);
        }

        /* Try to acquire cpu_lock if not held yet. */
        if (!MUTEX_HELD(&cpu_lock)) {
                mutex_enter(&cpu_lock);
                unlock = 1;
        }
        /*
         * Pause all other CPUs (and let them run pause thread).
         * It’s guaranteed that no other threads will access cpu_idle_cb_array
         * after pause_cpus().
         */
        if (!cpus_paused()) {
                pause_cpus(NULL, NULL);
                pause_cpus(NULL);
                unpause = 1;
        }

        /* Copy content to new buffer if needed. */
        if (buf_new != NULL) {
                buf_old = (char *)cpu_idle_cb_array;
                cnt_old = cpu_idle_cb_max;
                if (buf_old != NULL) {
                        ASSERT(cnt_old != 0);
                        bcopy(cpu_idle_cb_array, buf_new,
                            sizeof (cpu_idle_cb_item_t) * cnt_old);
                }
                cpu_idle_cb_array = (cpu_idle_cb_item_t *)buf_new;
                cpu_idle_cb_max = cnt_new;
        }

        /* Insert into array according to priority. */
        ASSERT(cpu_idle_cb_curr < cpu_idle_cb_max);
        for (i = cpu_idle_cb_curr; i > 0; i--) {
                if (cpu_idle_cb_array[i - 1].impl->priority >= cip->priority) {
                        break;
                }
                cpu_idle_cb_array[i] = cpu_idle_cb_array[i - 1];
        }
        cpu_idle_cb_array[i].arg = cip->argument;
        cpu_idle_cb_array[i].enter = cip->callback->idle_enter;
```

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

        /* Resume other CPUs from paused state if needed. */
        if (unpause) {
                start_cpus();
        }
        if (unlock) {
                mutex_exit(&cpu_lock);
        }

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

static void
cpu_idle_remove_callback(cpu_idle_cb_impl_t *cip)
{
        int i, found = 0;
        int unlock = 0, unpause = 0;
        cpu_idle_cb_state_t *sp;

        ASSERT(MUTEX_HELD(&cpu_idle_cb_lock));

        /* Try to acquire cpu_lock if not held yet. */
        if (!MUTEX_HELD(&cpu_lock)) {
                mutex_enter(&cpu_lock);
                unlock = 1;
        }
        /*
         * Pause all other CPUs.
         * It’s guaranteed that no other threads will access cpu_idle_cb_array
         * after pause_cpus().
         */
        if (!cpus_paused()) {
                pause_cpus(NULL, NULL);
                pause_cpus(NULL);
                unpause = 1;
        }

        /* Remove cip from array. */
        for (i = 0; i < cpu_idle_cb_curr; i++) {
                if (found == 0) {
                        if (cpu_idle_cb_array[i].impl == cip) {
                                found = 1;
                        }
                } else {
                        cpu_idle_cb_array[i - 1] = cpu_idle_cb_array[i];
                }
        }
        ASSERT(found != 0);
        cpu_idle_cb_curr--;

        /*
         * Reset property ready flag for all CPUs if no registered callback
         * left because cpu_idle_enter/exit will stop updating property if
         * there’s no callback registered.
         */
        if (cpu_idle_cb_curr == 0) {
                for (sp = cpu_idle_cb_state, i = 0; i < max_ncpus; i++, sp++) {
                        sp->v.ready = B_FALSE;
                }
        }

        /* Resume other CPUs from paused state if needed. */
        if (unpause) {
                start_cpus();
        }
        if (unlock) {
                mutex_exit(&cpu_lock);
        }
}
```

______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_

int
cpupm_set_policy(cpupm_policy_t new_policy)
{
        static int      gov_init = 0;
        int             result = 0;

        mutex_enter(&cpu_lock);
        if (new_policy == cpupm_policy) {
                mutex_exit(&cpu_lock);
                return (result);
        }

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

                result = cmt_pad_disable(PGHW_POW_ACTIVE);

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

                /*
                 * Bring all the active power domains to the maximum
                 * performance state.
                 */
                cpupm_state_change_global(CPUPM_DTYPE_ACTIVE,
                    CPUPM_STATE_MAX_PERF);

                break;
        case CPUPM_POLICY_ELASTIC:

                result = cmt_pad_enable(PGHW_POW_ACTIVE);
                if (result < 0) {
                        /*
                         * Failed to enable PAD across the active power
                         * domains, which may well be because none were
                         * enumerated.
                         */
                        break;
                }

                /*
                 * Initialize the governor parameters the first time through.
                 */
                if (gov_init == 0) {
                        cpupm_governor_initialize();
                        gov_init = 1;
                }
```

```

                pause_cpus(NULL, NULL);
                pause_cpus(NULL);
                cpupm_policy = CPUPM_POLICY_ELASTIC;
                start_cpus();

                break;
        default:
                cmn_err(CE_WARN, "Attempt to set unknown CPUPM policy %d\n",
                    new_policy);
                ASSERT(0);
                break;
        }
        mutex_exit(&cpu_lock);

        return (result);
}
______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_

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

        /*
         * Grab CPU lock (if we haven't already)
         */
        if (!MUTEX_HELD(&cpu_lock)) {
                mutex_enter(&cpu_lock);
                drop_lock = B_TRUE;
        }

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

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

        /*
         * Update lgroup topology with new memory resources, keeping track of
         * which lgroups change
         */
        count = 0;
        klgrpset_clear(changed);
        my_lgrp = lgrp_hand_to_lgrp(hand);
        if (my_lgrp == NULL) {
                /* new lgrp */
                my_lgrp = lgrp_create();
                lgrpid = my_lgrp->lgrp_id;
                my_lgrp->lgrp_plathand = hand;
                my_lgrp->lgrp_latency = lgrp_plat_latency(hand, hand);
                klgrpset_add(my_lgrp->lgrp_leaves, lgrpid);
                klgrpset_add(my_lgrp->lgrp_set[LGRP_RSRC_MEM], lgrpid);

                if (need_synch)
                        pause_cpus(NULL, NULL);
                        pause_cpus(NULL);
                count = lgrp_leaf_add(my_lgrp, lgrp_table, lgrp_alloc_max + 1,
                    &changed);
                if (need_synch)
                        start_cpus();
        } else if (my_lgrp->lgrp_latency == 0 && lgrp_plat_latency(hand, hand)
            > 0) {
                /*
                 * Leaf lgroup was created, but latency wasn't available
                 * then.  So, set latency for it and fill in rest of lgroup
                 * topology  now that we know how far it is from other leaf
                 * lgroups.
                 */
                klgrpset_clear(changed);
                lgrpid = my_lgrp->lgrp_id;
                if (!klgrpset_ismember(my_lgrp->lgrp_set[LGRP_RSRC_MEM],
                    lgrpid))
                        klgrpset_add(my_lgrp->lgrp_set[LGRP_RSRC_MEM], lgrpid);
                if (need_synch)
                        pause_cpus(NULL, NULL);
                        pause_cpus(NULL);
                count = lgrp_leaf_add(my_lgrp, lgrp_table, lgrp_alloc_max + 1,
                    &changed);
                if (need_synch)
                        start_cpus();
        } else if (!klgrpset_ismember(my_lgrp->lgrp_set[LGRP_RSRC_MEM],
            my_lgrp->lgrp_id)) {
                /*
                 * Add new lgroup memory resource to existing lgroup
                 */
                lgrpid = my_lgrp->lgrp_id;
                klgrpset_add(my_lgrp->lgrp_set[LGRP_RSRC_MEM], lgrpid);
                klgrpset_add(changed, lgrpid);
                count++;
                for (i = 0; i <= lgrp_alloc_max; i++) {
                        lgrp_t          *lgrp;

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

                        klgrpset_add(lgrp->lgrp_set[LGRP_RSRC_MEM], lgrpid);
                        klgrpset_add(changed, lgrp->lgrp_id);
                        count++;
                }
        }

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

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

	if (drop_lock)
		mutex_exit(&cpu_lock);
}

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

	/*
	 * Grab CPU lock (if we haven't already)
	 */
	if (!MUTEX_HELD(&cpu_lock)) {
		mutex_enter(&cpu_lock);
		drop_lock = B_TRUE;
	}

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

	my_lgrp = lgrp_hand_to_lgrp(hand);

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

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

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

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

		/*
		 * Remove memory node from lgroup.
		 */
		lgrp->lgrp_mnodes &= ~mnodes_mask;
		lgrp->lgrp_nmnodes--;
		ASSERT(lgrp->lgrp_nmnodes >= 0);
	}
	ASSERT(lgrp_root->lgrp_nmnodes > 0);

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

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

	/*
	 * Remove this lgroup from lgroup topology if it does not contain any
	 * resources now
	 */
	lgrpid = my_lgrp->lgrp_id;
	count = 0;
	klgrpset_clear(changed);
	if (lgrp_rsets_empty(my_lgrp->lgrp_set)) {
		/*
		 * Delete lgroup when no more resources
		 */
		if (need_synch)
```

```
                        pause_cpus(NULL, NULL);
                        pause_cpus(NULL);
                count = lgrp_leaf_delete(my_lgrp, lgrp_table,
                    lgrp_alloc_max + 1, &changed);
                ASSERT(count > 0);
                if (need_synch)
                        start_cpus();
        } else {
                /*
                 * Remove lgroup from memory resources of any lgroups that
                 * contain it as such
                 */
                for (i = 0; i <= lgrp_alloc_max; i++) {
                        lgrp_t          *lgrp;

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

                        klgrpset_del(lgrp->lgrp_set[LGRP_RSRC_MEM], lgrpid);
                }
        }
        if (drop_lock)
                mutex_exit(&cpu_lock);
}
______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_


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

        count = 0;
        if (changed)
                klgrpset_clear(*changed);

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

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

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

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

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

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

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

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

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

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

        start_cpus();
        mutex_exit(&cpu_lock);

        return (count);
}
______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_*

```
/*
 * Invalidate memseg pointers in cpu private vm data caches.
 */
static void
memseg_cpu_vm_flush()
{
        cpu_t *cp;
        vm_cpu_data_t *vc;

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

        cp = cpu_list;
        do {
                vc = cp->cpu_vm_data;
                vc->vc_pnum_memseg = NULL;
                vc->vc_pnext_memseg = NULL;

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

        start_cpus();
        mutex_exit(&cpu_lock);
}
```

______*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_***

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

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

#endif /* _KERNEL || _KMEMUSER */

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

struct zone;

void    cpu_list_init(cpu_t *);
void    cpu_add_unit(cpu_t *);
void    cpu_del_unit(int cpuid);
void    cpu_add_active(cpu_t *);
void    cpu_kstat_init(cpu_t *);
void    cpu_visibility_add(cpu_t *, struct zone *);
void    cpu_visibility_remove(cpu_t *, struct zone *);
void    cpu_visibility_configure(cpu_t *, struct zone *);
void    cpu_visibility_unconfigure(cpu_t *, struct zone *);
void    cpu_visibility_online(cpu_t *, struct zone *);
void    cpu_visibility_offline(cpu_t *, struct zone *);
void    cpu_create_intrstat(cpu_t *);
void    cpu_delete_intrstat(cpu_t *);
int     cpu_kstat_intrstat_update(kstat_t *, int);
void    cpu_intr_swtch_enter(kthread_t *);
void    cpu_intr_swtch_exit(kthread_t *);

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

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

void    mach_cpu_pause(volatile char *);

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

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

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

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

/*
 * Routines for checking CPU states.
 */
int     cpu_is_online(cpu_t *);         /* check if CPU is online */
int     cpu_is_nointr(cpu_t *);         /* check if CPU can service intrs */
int     cpu_is_active(cpu_t *);         /* check if CPU can run threads */
int     cpu_is_offline(cpu_t *);        /* check if CPU is offline */
int     cpu_is_poweredoff(cpu_t *);     /* check if CPU is powered off */

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

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


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

int     cpu_configure(int);
int     cpu_unconfigure(int);
void    cpu_destroy_bound_threads(cpu_t *cp);

extern int cpu_bind_thread(kthread_t *tp, processorid_t bind,
    processorid_t *obind, int *error);
extern int cpu_unbind(processorid_t cpu_id, boolean_t force);
extern void thread_affinity_set(kthread_t *t, int cpu_id);
extern void thread_affinity_clear(kthread_t *t);
extern void affinity_set(int cpu_id);
extern void affinity_clear(void);
extern void init_cpu_mstate(struct cpu *, int);
extern void term_cpu_mstate(struct cpu *);
extern void new_cpu_mstate(int, hrtime_t);
extern void get_cpu_mstate(struct cpu *, hrtime_t *);
extern void thread_nomigrate(void);
extern void thread_allowmigrate(void);
extern void weakbinding_stop(void);
extern void weakbinding_start(void);

/*
```

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

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

/*
 * CPU state change events
 *
 * Various subsystems need to know when CPUs change their state. They get this
 * information by registering  CPU state change callbacks using
 * register_cpu_setup_func(). Whenever any CPU changes its state, the callback
 * function is called. The callback function is passed three arguments:
 *
 *   Event, described by cpu_setup_t
 *   CPU ID
 *   Transparent pointer passed when registering the callback
 *
 * The callback function is called with cpu_lock held. The return value from the
 * callback function is usually ignored, except for CPU_CONFIG and CPU_UNCONFIG
 * events. For these two events, non-zero return value indicates a failure and
 * prevents successful completion of the operation.
 *
 * New events may be added in the future. Callback functions should ignore any
 * events that they do not understand.
 *
 * The following events provide notification callbacks:
 *
 *  CPU_INIT    A new CPU is started and added to the list of active CPUs
 *                This event is only used during boot
 *
 *  CPU_CONFIG  A newly inserted CPU is prepared for starting running code
 *                This event is called by DR code
 *
 *  CPU_UNCONFIG CPU has been powered off and needs cleanup
 *                This event is called by DR code
 *
 *  CPU_ON      CPU is enabled but does not run anything yet
 *
 *  CPU_INTR_ON CPU is enabled and has interrupts enabled
 *
 *  CPU_OFF     CPU is going offline but can still run threads
 *
 *  CPU_CPUPART_OUT     CPU is going to move out of its partition
 *
 *  CPU_CPUPART_IN      CPU is going to move to a new partition
 *
 *  CPU_SETUP   CPU is set up during boot and can run threads
 */
typedef enum {
        CPU_INIT,
        CPU_CONFIG,
        CPU_UNCONFIG,
        CPU_ON,
        CPU_OFF,
        CPU_CPUPART_IN,
        CPU_CPUPART_OUT,
```

```
        CPU_SETUP,
        CPU_INTR_ON
} 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_

/*
 * Top level routine to direct suspend/resume of a domain.
 */
void
xen_suspend_domain(void)
{
        extern void rtcsync(void);
        extern void ec_resume(void);
        extern kmutex_t ec_lock;
        struct xen_add_to_physmap xatp;
        ulong_t flags;
        int err;

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

        SUSPEND_DEBUG("xen_suspend_domain\n");

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

        /*
         * suspend interrupts and devices
         */
        if (xpvd_dip != NULL)
                (void) xen_suspend_devices(ddi_get_child(xpvd_dip));
        else
                cmn_err(CE_WARN, "No PV devices found to suspend");
        SUSPEND_DEBUG("xenbus_suspend\n");
        xenbus_suspend();

        mutex_enter(&cpu_lock);

        /*
         * Suspend on vcpu 0
         */
        thread_affinity_set(curthread, 0);
        kpreempt_disable();

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

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

        flags = intr_clear();
```

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

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

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

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

        intr_restore(flags);

        if (ncpus > 1)
                start_cpus();

        mutex_exit(&ec_lock);
        mutex_exit(&cpu_lock);

        /*
         * Now we can take locks again.
         */

        rtcsync();

        SUSPEND_DEBUG("xenbus_resume\n");
        xenbus_resume();
        SUSPEND_DEBUG("xen_resume_devices\n");
        if (xpvd_dip != NULL)
                (void) xen_resume_devices(ddi_get_child(xpvd_dip), 0);

        thread_affinity_clear(curthread);
        kpreempt_enable();

        SUSPEND_DEBUG("finished xen_suspend_domain\n");

        cmn_err(CE_NOTE, "domain restore/migrate completed");
}
______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_
```

```
int
dr_suspend(dr_sr_handle_t *srh)
{
        dr_handle_t     *handle;
        int             force;
        int             dev_errs_idx;
        uint64_t        dev_errs[DR_MAX_ERR_INT];
        int             rc = DDI_SUCCESS;

        handle = srh->sr_dr_handlep;

        force = dr_cmd_flags(handle) & SBD_FLAG_FORCE;

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

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

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

                drc.arr = dev_errs;
                drc.idx = &dev_errs_idx;
                drc.len = DR_MAX_ERR_INT;

                /*
                 * Since the root node can never go away, it
                 * doesn’t have to be held.
                 */
                ddi_walk_devs(ddi_root_node(), dr_check_unsafe_major, &drc);
                if (dev_errs_idx) {
                        handle->h_err = drerr_int(ESBD_UNSAFE, dev_errs,
                            dev_errs_idx, 1);
                        dr_resume(srh);
                        return (DDI_FAILURE);
                }
                PR_QR("done\n");
        } else {
                prom_printf("\nDR: dr_suspend invoked with force flag\n");
        }

#ifndef SKIP_SYNC
        /*
         * This sync swap out all user pages
         */
        vfs_sync(SYNC_ALL);
#endif

        /*
         * special treatment for lock manager
         */
        lm_cprsuspend();
```

```
#ifndef SKIP_SYNC
        /*
         * sync the file system in case we never make it back
         */
        sync();
#endif

        /*
         * now suspend drivers
         */
        prom_printf("DR: suspending drivers...\n");
        srh->sr_suspend_state = DR_SRSTATE_DRIVER;
        srh->sr_err_idx = 0;
        /* No parent to hold busy */
        if ((rc = dr_suspend_devices(ddi_root_node(), srh)) != DDI_SUCCESS) {
                if (srh->sr_err_idx && srh->sr_dr_handlep) {
                        (srh->sr_dr_handlep)->h_err = drerr_int(ESBD_SUSPEND,
                            srh->sr_err_ints, srh->sr_err_idx, 1);
                }
                dr_resume(srh);
                return (rc);
        }

        drmach_suspend_last();

        /*
         * finally, grab all cpus
         */
        srh->sr_suspend_state = DR_SRSTATE_FULL;

        mutex_enter(&cpu_lock);
        pause_cpus(NULL, NULL);
        pause_cpus(NULL);
        dr_stop_intr();

        return (rc);
}
______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) {
                PMD(PMD_SX, ("... failed\n!"))
                goto insomnia;
        }

        switch (s3ap->s3a_test_point) {
        case DEVICE_SUSPEND_TO_RAM:
        case FORCE_SUSPEND_TO_RAM:
        case LOOP_BACK_PASS:
                return (0);
        case LOOP_BACK_FAIL:
                return (1);
        default:
                ASSERT(s3ap->s3a_test_point == LOOP_BACK_NONE);
        }

        /*
         * Tell the hardware to sleep.
         */
        PT(PT_SXE);
        PMD(PMD_SX, ("Calling AcpiEnterSleepState(%d) ...\n", Sx))
        if (AcpiEnterSleepState(Sx) != AE_OK) {
                PT(PT_SXE_FAIL);
                PMD(PMD_SX, ("... failed!\n"))
        }

insomnia:
        PT(PT_INSOM);
        /* cleanup is done in the caller */
        return (1);
}
```

______***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_

/*
 * Stop all other cpu’s before halting or rebooting. We pause the cpu’s
 * instead of sending a cross call.
 * Stolen from sun4/os/mp_states.c
 */

static int cpu_are_paused;      /* sic */

void
i_cpr_stop_other_cpus(void)
{
        mutex_enter(&cpu_lock);
        if (cpu_are_paused) {
                mutex_exit(&cpu_lock);
                return;
        }
        pause_cpus(NULL, NULL);
        pause_cpus(NULL);
        cpu_are_paused = 1;

        mutex_exit(&cpu_lock);
}
______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);
        /*
         * Clear any unresolved UEs from memory.
         */
        page_retire_mdboot();

#if defined(__xpv)
        /*
         * XXPV Should probably think some more about how we deal
         *      with panicing before it’s really safe to panic.
         *      On hypervisors, we reboot very quickly..  Perhaps panic
         *      should only attempt to recover by rebooting if,
         *      say, we were able to mount the root filesystem,
         *      or if we successfully launched init(1m).
         */
        if (panicstr && proc_init == NULL)
                (void) HYPERVISOR_shutdown(SHUTDOWN_poweroff);
#endif
        /*
         * stop other cpus and raise our priority.  since there is only
         * one active cpu after this, and our priority will be too high
         * for us to be preempted, we’re essentially single threaded
         * from here on out.
         */
        (void) spl6();
        if (!panicstr) {
                mutex_enter(&cpu_lock);
                pause_cpus(NULL, NULL);
                pause_cpus(NULL);
                mutex_exit(&cpu_lock);
        }

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

        /*
         * Try to quiesce devices.
         */
        if (is_first_quiesce) {
                /*
                 * Clear is_first_quiesce before calling quiesce_devices()
                 * so that if quiesce_devices() causes panics, it will not
                 * be invoked again.
                 */
                is_first_quiesce = 0;

                quiesce_active = 1;
                quiesce_devices(ddi_root_node(), &reset_status);
                if (reset_status == -1) {
                        if (fcn == AD_FASTREBOOT && !force_fastreboot) {
                                prom_printf("Driver(s) not capable of fast "
                                    "reboot.\n");
                                prom_printf(fallback_str);
                                fastreboot_capable = 0;
                                fcn = AD_BOOT;
                        } else if (fcn != AD_FASTREBOOT)
                                fastreboot_capable = 0;
                }
```

```
                quiesce_active = 0;
        }

        /*
         * Try to reset devices. reset_leaves() should only be called
         * a) when there are no other threads that could be accessing devices,
         *    and
         * b) on a system that’s not capable of fast reboot (fastreboot_capable
         *    being 0), or on a system where quiesce_devices() failed to
         *    complete (quiesce_active being 1).
         */
        if (is_first_reset && (!fastreboot_capable || quiesce_active)) {
                /*
                 * Clear is_first_reset before calling reset_devices()
                 * so that if reset_devices() causes panics, it will not
                 * be invoked again.
                 */
                is_first_reset = 0;
                reset_leaves();
        }

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

        (void) spl8();

        if (fastreboot_capable && fcn == AD_FASTREBOOT) {
                /*
                 * psm_shutdown is called within fast_reboot()
                 */
                fast_reboot();
        } else {
                (*psm_shutdownf)(cmd, fcn);

                if (fcn == AD_HALT || fcn == AD_POWEROFF)
                        halt((char *)NULL);
                else
                        prom_reboot("");
        }
        /*NOTREACHED*/
}
```

______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
****

```
/*
 * 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) 2007, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright (c) 2010, Intel Corporation.
 * All rights reserved.
 */
/*
 * Copyright 2011 Joyent, Inc. All rights reserved.
 */

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

#include <sys/types.h>
#include <sys/systm.h>
#include <sys/cpuvar.h>
#include <sys/cpu_module.h>
#include <sys/kmem.h>
#include <sys/archsystm.h>
#include <sys/machsystm.h>
#include <sys/controlregs.h>
#include <sys/x86_archext.h>
#include <sys/smp_impldefs.h>
#include <sys/sysmacros.h>
#include <sys/mach_mmu.h>
#include <sys/promif.h>
#include <sys/cpu.h>
#include <sys/cpu_event.h>
#include <sys/sunndi.h>
#include <sys/fs/dv_node.h>
#include <vm/hat_i86.h>
#include <vm/as.h>

extern cpuset_t cpu_ready_set;

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

```
extern void real_mode_stop_cpu_stage1(void);
extern void real_mode_stop_cpu_stage1_end(void);
extern void real_mode_stop_cpu_stage2(void);
extern void real_mode_stop_cpu_stage2_end(void);
extern void *(*cpu_pause_func)(void *);

void rmp_gdt_init(rm_platter_t *);

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

int
mach_cpucontext_init(void)
{
        ushort_t *vec;
        ulong_t addr;
        struct rm_platter *rm = (struct rm_platter *)rm_platter_va;

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

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

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

        /* Copy CPU startup code to rm_platter if it’s still during boot. */
        if (!plat_dr_enabled()) {
                ASSERT((size_t)real_mode_start_cpu_end -
                    (size_t)real_mode_start_cpu <= RM_PLATTER_CODE_SIZE);
                bcopy((caddr_t)real_mode_start_cpu, (caddr_t)rm->rm_code,
                    (size_t)real_mode_start_cpu_end -
                    (size_t)real_mode_start_cpu);
        }

        return (0);
}
```

______*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_

#define XC_FLUSH_MAX_WAITS              1000

/* Flush inflight message buffers. */
int
xc_flush_cpu(struct cpu *cpup)
{
        int i;

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

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

        for (i = 0; i < XC_FLUSH_MAX_WAITS; i++) {
                if (cpup->cpu_m.xc_work_cnt == 0) {
                        break;
                }
                DELAY(1);
        }
        for (; i < XC_FLUSH_MAX_WAITS; i++) {
                if (!BT_TEST(xc_priority_set, cpup->cpu_id)) {
                        break;
                }
                DELAY(1);
        }

        return (i >= XC_FLUSH_MAX_WAITS ? ETIME : 0);
}
______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_

void
mp_enter_barrier(void)
{
        hrtime_t last_poke_time = 0;
        int poke_allowed = 0;
        int done = 0;
        int i;

        ASSERT(MUTEX_HELD(&cpu_lock));

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

        while (!done) {
                done = 1;
                poke_allowed = 0;

                if (xpv_gethrtime() - last_poke_time > POKE_TIMEOUT) {
                        last_poke_time = xpv_gethrtime();
                        poke_allowed = 1;
                }

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

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

                        switch (cpu_phase[i]) {
                        case CPU_PHASE_NONE:
                                cpu_phase[i] = CPU_PHASE_WAIT_SAFE;
                                poke_cpu(i);
                                done = 0;
                                break;

                        case CPU_PHASE_WAIT_SAFE:
                                if (poke_allowed)
                                        poke_cpu(i);
                                done = 0;
                                break;

                        case CPU_PHASE_SAFE:
                        case CPU_PHASE_POWERED_OFF:
                                break;
                        }
                }

                SMT_PAUSE();
        }
}
______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_

/*
 * Stop all other cpu’s before halting or rebooting. We pause the cpu’s
 * instead of sending a cross call.
 */
void
stop_other_cpus(void)
{
        mutex_enter(&cpu_lock);
        if (cpu_are_paused) {
                mutex_exit(&cpu_lock);
                return;
        }

        if (ncpus > 1)
                intr_redist_all_cpus_shutdown();

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

        mutex_exit(&cpu_lock);
}
______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_

/*
 * This routine is a special form of pause_cpus().  It ensures that
 * prom functions are callable while the cpus are paused.
 */
void
promsafe_pause_cpus(void)
{
        pause_cpus(NULL, NULL);
        pause_cpus(NULL);

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

                start_cpus();
                mutex_enter(&prom_mutex);

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

                mutex_exit(&prom_mutex);
                pause_cpus(NULL, NULL);
                pause_cpus(NULL);
        }

        /* At this point all cpus are paused and none are in the prom */
}
______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_
```

```
static int
mem_cache_ioctl_ops(int cmd, int mode, cache_info_t *cache_info)
{
        int     ret_val = 0;
        uint64_t afar, tag_addr;
        ch_cpu_logout_t clop;
        uint64_t Lxcache_tag_data[PN_CACHE_NWAYS];
        int     i, retire_retry_count;
        cpu_t   *cpu;
        uint64_t tag_data;
        uint8_t state;

        if (cache_info->way >= PN_CACHE_NWAYS)
                return (EINVAL);
        switch (cache_info->cache) {
                case L2_CACHE_TAG:
                case L2_CACHE_DATA:
                        if (cache_info->index >=
                            (PN_L2_SET_SIZE/PN_L2_LINESIZE))
                                return (EINVAL);
                        break;
                case L3_CACHE_TAG:
                case L3_CACHE_DATA:
                        if (cache_info->index >=
                            (PN_L3_SET_SIZE/PN_L3_LINESIZE))
                                return (EINVAL);
                        break;
                default:
                        return (ENOTSUP);
        }
        /*
         * Check if we have a valid cpu ID and that
         * CPU is ONLINE.
         */
        mutex_enter(&cpu_lock);
        cpu = cpu_get(cache_info->cpu_id);
        if ((cpu == NULL) || (!cpu_is_online(cpu))) {
                mutex_exit(&cpu_lock);
                return (EINVAL);
        }
        mutex_exit(&cpu_lock);
        pattern = 0;    /* default value of TAG PA when cacheline is retired. */
        switch (cmd) {
                case MEM_CACHE_RETIRE:
                        tag_addr = get_tag_addr(cache_info);
                        pattern |= PN_ECSTATE_NA;
                        retire_retry_count = 0;
                        affinity_set(cache_info->cpu_id);
                        switch (cache_info->cache) {
                                case L2_CACHE_DATA:
                                case L2_CACHE_TAG:
                                        if ((cache_info->bit & MSB_BIT_MASK) ==
                                            MSB_BIT_MASK)
                                                pattern |= PN_L2TAG_PA_MASK;
retry_l2_retire:
                                        if (tag_addr_collides(tag_addr,
                                            cache_info->cache,
                                            retire_l2_start, retire_l2_end))
                                                ret_val =
```

```
                                                    retire_l2_alternate(
                                                    tag_addr, pattern);
                                        else
                                                ret_val = retire_l2(tag_addr,
                                                    pattern);
                                        if (ret_val == 1) {
                                                /*
                                                 * cacheline was in retired
                                                 * STATE already.
                                                 * so return success.
                                                 */
                                                ret_val = 0;
                                        }
                                        if (ret_val < 0) {
                                                cmn_err(CE_WARN,
                "retire_l2() failed. index = 0x%x way %d. Retrying...\n",
                                                    cache_info->index,
                                                    cache_info->way);
                                                if (retire_retry_count >= 2) {
                                                        retire_failures++;
                                                        affinity_clear();
                                                        return (EIO);
                                                }
                                                retire_retry_count++;
                                                goto retry_l2_retire;
                                        }
                                        if (ret_val == 2)
                                                l2_flush_retries_done++;
                        /*
                         * We bind ourself to a CPU and send cross trap to
                         * ourself. On return from xt_one we can rely on the
                         * data in tag_data 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_l2_tag_tl1),
                                            tag_addr, (uint64_t)(&tag_data));
                                        state = tag_data & CH_ECSTATE_MASK;
                                        if (state != PN_ECSTATE_NA) {
                                                retire_failures++;
                                                print_l2_tag(tag_addr,
                                                    tag_data);
                                                cmn_err(CE_WARN,
                "L2 RETIRE:failed for index 0x%x way %d. Retrying...\n",
                                                    cache_info->index,
                                                    cache_info->way);
                                                if (retire_retry_count >= 2) {
                                                        retire_failures++;
                                                        affinity_clear();
                                                        return (EIO);
                                                }
                                                retire_retry_count++;
                                                goto retry_l2_retire;
                                        }
                                        break;
                                case L3_CACHE_TAG:
                                case L3_CACHE_DATA:
                                        if ((cache_info->bit & MSB_BIT_MASK) ==
                                            MSB_BIT_MASK)
                                                pattern |= PN_L3TAG_PA_MASK;
                                        if (tag_addr_collides(tag_addr,
                                            cache_info->cache,
                                            retire_l3_start, retire_l3_end))
                                                ret_val =
                                                    retire_l3_alternate(
                                                    tag_addr, pattern);
                                        else
                                                ret_val = retire_l3(tag_addr,
                                                    pattern);
                                        if (ret_val == 1) {
                                                /*
                                                 * cacheline was in retired
                                                 * STATE already.
                                                 * so return success.
                                                 */
                                                ret_val = 0;
                                        }
                                        if (ret_val < 0) {
                                                cmn_err(CE_WARN,
                        "retire_l3() failed. ret_val = %d index = 0x%x\n",
                                                    ret_val,
                                                    cache_info->index);
                                                retire_failures++;
                                                affinity_clear();
                                                return (EIO);
                                        }
                        /*
                         * We bind ourself to a CPU and send cross trap to
                         * ourself. On return from xt_one we can rely on the
                         * data in tag_data 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_l3_tag_tl1),
                                            tag_addr, (uint64_t)(&tag_data));
                                        state = tag_data & CH_ECSTATE_MASK;
                                        if (state != PN_ECSTATE_NA) {
                                                cmn_err(CE_WARN,
                                        "L3 RETIRE failed for index 0x%x\n",
                                                    cache_info->index);
                                                retire_failures++;
                                                affinity_clear();
                                                return (EIO);
                                        }

                                        break;
                        }
                        affinity_clear();
                        break;
                case MEM_CACHE_UNRETIRE:
                        tag_addr = get_tag_addr(cache_info);
                        pattern = PN_ECSTATE_INV;
                        affinity_set(cache_info->cpu_id);
                        switch (cache_info->cache) {
                                case L2_CACHE_DATA:
                                case L2_CACHE_TAG:
                        /*
                         * We bind ourself to a CPU and send cross trap to
                         * ourself. On return from xt_one we can rely on the
                         * data in tag_data 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_l2_tag_tl1),
                                            tag_addr, (uint64_t)(&tag_data));
                                        state = tag_data & CH_ECSTATE_MASK;
                                        if (state != PN_ECSTATE_NA) {
                                                affinity_clear();
                                                return (EINVAL);
```

```
                                        }
                                        if (tag_addr_collides(tag_addr,
                                            cache_info->cache,
                                            unretire_l2_start, unretire_l2_end))
                                                ret_val =
                                                    unretire_l2_alternate(
                                                    tag_addr, pattern);
                                        else
                                                ret_val =
                                                    unretire_l2(tag_addr,
                                                    pattern);
                                        if (ret_val != 0) {
                                                cmn_err(CE_WARN,
                        "unretire_l2() failed. ret_val = %d index = 0x%x\n",
                                                    ret_val,
                                                    cache_info->index);
                                                retire_failures++;
                                                affinity_clear();
                                                return (EIO);
                                        }
                                        break;
                                case L3_CACHE_TAG:
                                case L3_CACHE_DATA:
                        /*
                         * We bind ourself to a CPU and send cross trap to
                         * ourself. On return from xt_one we can rely on the
                         * data in tag_data 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_l3_tag_tl1),
                                            tag_addr, (uint64_t)(&tag_data));
                                        state = tag_data & CH_ECSTATE_MASK;
                                        if (state != PN_ECSTATE_NA) {
                                                affinity_clear();
                                                return (EINVAL);
                                        }
                                        if (tag_addr_collides(tag_addr,
                                            cache_info->cache,
                                            unretire_l3_start, unretire_l3_end))
                                                ret_val =
                                                    unretire_l3_alternate(
                                                    tag_addr, pattern);
                                        else
                                                ret_val =
                                                    unretire_l3(tag_addr,
                                                    pattern);
                                        if (ret_val != 0) {
                                                cmn_err(CE_WARN,
                        "unretire_l3() failed. ret_val = %d index = 0x%x\n",
                                                    ret_val,
                                                    cache_info->index);
                                                affinity_clear();
                                                return (EIO);
                                        }
                                        break;
                        }
                        affinity_clear();
                        break;
                case MEM_CACHE_ISRETIRED:
                case MEM_CACHE_STATE:
                        return (ENOTSUP);
                case MEM_CACHE_READ_TAGS:
#ifdef DEBUG
                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_

int
dr_suspend(dr_sr_handle_t *srh)
{
        dr_handle_t     *handle;
        int             force;
        int             dev_errs_idx;
        uint64_t        dev_errs[DR_MAX_ERR_INT];
        int             rc = DDI_SUCCESS;

        handle = srh->sr_dr_handlep;

        force = dr_cmd_flags(handle) & SBD_FLAG_FORCE;

        /*
         * update the signature block
         */
        CPU_SIGNATURE(OS_SIG, SIGST_QUIESCE_INPROGRESS, SIGSUBST_NULL,
            CPU->cpu_id);

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

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

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

                drc.arr = dev_errs;
                drc.idx = &dev_errs_idx;
                drc.len = DR_MAX_ERR_INT;

                /*
                 * Since the root node can never go away, it
                 * doesn't have to be held.
                 */
                ddi_walk_devs(ddi_root_node(), dr_check_unsafe_major, &drc);
                if (dev_errs_idx) {
                        handle->h_err = drerr_int(ESBD_UNSAFE, dev_errs,
                            dev_errs_idx, 1);
                        dr_resume(srh);
                        return (DDI_FAILURE);
                }
                PR_QR("done\n");
        } else {
                prom_printf("\nDR: dr_suspend invoked with force flag\n");
        }

#ifndef SKIP_SYNC
        /*
         * This sync swap out all user pages
         */
        vfs_sync(SYNC_ALL);
#endif
```

```
        /*
         * special treatment for lock manager
         */
        lm_cprsuspend();

#ifndef SKIP_SYNC
        /*
         * sync the file system in case we never make it back
         */
        sync();
#endif

        /*
         * now suspend drivers
         */
        prom_printf("DR: suspending drivers...\n");
        srh->sr_suspend_state = DR_SRSTATE_DRIVER;
        srh->sr_err_idx = 0;
        /* No parent to hold busy */
        if ((rc = dr_suspend_devices(ddi_root_node(), srh)) != DDI_SUCCESS) {
                if (srh->sr_err_idx && srh->sr_dr_handlep) {
                        (srh->sr_dr_handlep)->h_err = drerr_int(ESBD_SUSPEND,
                            srh->sr_err_ints, srh->sr_err_idx, 1);
                }
                dr_resume(srh);
                return (rc);
        }

        drmach_suspend_last();

        /*
         * finally, grab all cpus
         */
        srh->sr_suspend_state = DR_SRSTATE_FULL;

        /*
         * if watchdog was activated, disable it
         */
        if (watchdog_activated) {
                mutex_enter(&tod_lock);
                tod_ops.tod_clear_watchdog_timer();
                mutex_exit(&tod_lock);
                srh->sr_flags |= SR_FLAG_WATCHDOG;
        } else {
                srh->sr_flags &= ~(SR_FLAG_WATCHDOG);
        }

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

        mutex_enter(&cpu_lock);
        pause_cpus(NULL, NULL);
        pause_cpus(NULL);
        dr_stop_intr();

        return (rc);
}
______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_

int
sbdp_suspend(sbdp_sr_handle_t *srh)
{
        int force;
        int rc = DDI_SUCCESS;

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

        /*
         * if no force flag, check for unsafe drivers
         */
        if (force) {
                SBDP_DBG_QR("\nsbdp_suspend invoked with force flag");
        }

        /*
         * update the signature block
         */
        CPU_SIGNATURE(OS_SIG, SIGST_QUIESCE_INPROGRESS, SIGSUBST_NULL,
            CPU->cpu_id);

        /*
         * first, stop all user threads
         */
        SBDP_DBG_QR("SBDP: suspending user threads...\n");
        SR_SET_STATE(srh, SBDP_SRSTATE_USER);
        if (((rc = sbdp_stop_user_threads(srh)) != DDI_SUCCESS) &&
            sbdp_check_user_stop_result) {
                sbdp_resume(srh);
                return (rc);
        }

#ifndef SKIP_SYNC
        /*
         * This sync swap out all user pages
         */
        vfs_sync(SYNC_ALL);
#endif

        /*
         * special treatment for lock manager
         */
        lm_cprsuspend();

#ifndef SKIP_SYNC
        /*
         * sync the file system in case we never make it back
         */
        sync();

#endif
        /*
         * now suspend drivers
         */
        SBDP_DBG_QR("SBDP: suspending drivers...\n");
        SR_SET_STATE(srh, SBDP_SRSTATE_DRIVER);

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

```
         */
        if ((rc = sbdp_suspend_devices(ddi_root_node(), srh)) != DDI_SUCCESS) {
                sbdp_resume(srh);
                return (rc);
        }

        /*
         * finally, grab all cpus
         */
        SR_SET_STATE(srh, SBDP_SRSTATE_FULL);

        /*
         * if watchdog was activated, disable it
         */
        if (watchdog_activated) {
                mutex_enter(&tod_lock);
                saved_watchdog_seconds = tod_ops.tod_clear_watchdog_timer();
                mutex_exit(&tod_lock);
                SR_SET_FLAG(srh, SR_FLAG_WATCHDOG);
        } else {
                SR_CLEAR_FLAG(srh, SR_FLAG_WATCHDOG);
        }

        mutex_enter(&cpu_lock);
        pause_cpus(NULL, NULL);
        pause_cpus(NULL);
        sbdp_stop_intr();

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

        return (rc);
}
______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_

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

static void
mpo_wr_lock()
{
        mutex_enter(&cpu_lock);
        pause_cpus(NULL, NULL);
        pause_cpus(NULL);
        mutex_exit(&cpu_lock);
}
______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_

```
/*
 * Obtain an updated MD from the hypervisor and update cpunodes, CPU HW
 * sharing data structures, and processor groups.
 */
static void
update_cpu_mappings(void)
{
        md_t            *mdp;
        processorid_t   id;
        cpu_t           *cp;
        cpu_pg_t        *pgps[NCPU];

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

        DBG("suspend: updating CPU mappings");

        mutex_enter(&cpu_lock);

        setup_chip_mappings(mdp);
        setup_exec_unit_mappings(mdp);
        for (id = 0; id < NCPU; id++) {
                if ((cp = cpu_get(id)) == NULL)
                        continue;
                cpu_map_exec_units(cp);
        }

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

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

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

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

        /*
         * Now that PG data has been initialized for all CPUs in the
         * system, replace the bootstrapped PG structure with the
         * initialized PG structure and call pg_cpu_active for each CPU.
         */
        pause_cpus(NULL, NULL);
        pause_cpus(NULL);
        for (id = 0; id < NCPU; id++) {
                if ((cp = cpu_get(id)) == NULL)
                        continue;
                cp->cpu_pg = pgps[id];
                if ((cp->cpu_flags & CPU_OFFLINE) == 0)
                        pg_cpu_active(cp);
        }
        start_cpus();

        mutex_exit(&cpu_lock);

        (void) md_fini_handle(mdp);
}
```

______unchanged_portion_omitted_

```
/*
 * Suspends the OS by pausing CPUs and calling into the HV to initiate
 * the suspend. When the HV routine hv_guest_suspend returns, the system
 * will be resumed. Must be called after a successful call to suspend_pre.
 * suspend_post must be called after suspend_start, whether or not
 * suspend_start returns an error.
 */
/*ARGSUSED*/
int
suspend_start(char *error_reason, size_t max_reason_len)
{
        uint64_t        source_tick;
        uint64_t        source_stick;
        uint64_t        rv;
        timestruc_t     source_tod;
        int             spl;

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

        sfmmu_ctxdoms_lock();
        mutex_enter(&cpu_lock);

        /* Suspend the watchdog */
        watchdog_suspend();

        /* Record the TOD */
        mutex_enter(&tod_lock);
        source_tod = tod_get();
        mutex_exit(&tod_lock);

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

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

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

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

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

        suspend_count++;

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

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

        /* Enable interrupts */
        splx(spl);

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

```
                xc_all((xcfunc_t *)enable_tick_stick_npt, NULL, NULL);
                xt_sync(cpu_ready_set);
                ASSERT(gettick_npt() != 0);
                ASSERT(getstick_npt() != 0);
        }

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

        sfmmu_ctxdoms_remove();

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

        /* Set the TOD */
        mutex_enter(&tod_lock);
        tod_set(source_tod);
        mutex_exit(&tod_lock);

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

        mutex_exit(&cpu_lock);

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

        sfmmu_ctxdoms_update();
        sfmmu_ctxdoms_unlock();

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

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

        return (0);
}
```

______***unchanged_portion_omitted_***