```
**********************************************************
   95160 Tue May  7 07:35:42 2019
new/usr/src/uts/common/os/cpu.c
10923 thread_affinity_set(CPU_CURRENT) can skip cpu_lock
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>
Reviewed by: John Levon <john.levon@joyent.com>
**********************************************************
/*
 * 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.
 * Copyright 2018 Joyent, Inc.
 */

/*
 * 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 *);
} cpu_pause_info;
______unchanged_portion_omitted_

/*
 * Set affinity for a specified CPU.
 *
 * Specifying a cpu_id of CPU_CURRENT, allowed _only_ when setting affinity for
 * curthread, will set affinity to the CPU on which the thread is currently
 * running.  For other cpu_id values, the caller must ensure that the
 * referenced CPU remains valid, which can be done by holding cpu_lock across
 * this call.
 *
 * CPU affinity is guaranteed after return of thread_affinity_set().  If a
 * caller setting affinity to CPU_CURRENT requires that its thread not migrate
 * CPUs prior to a successful return, it should take extra precautions (such as
 * their own call to kpreempt_disable) to ensure that safety.
 *
 * A CPU affinity reference count is maintained by thread_affinity_set and
 * thread_affinity_clear (incrementing and decrementing it, respectively),
 * maintaining CPU affinity while the count is non-zero, and allowing regions
```

```
 * of code which require affinity to be nested.
 * A reference count is incremented and the affinity is held until the
 * reference count is decremented to zero by thread_affinity_clear().
 * This is so regions of code requiring affinity can be nested.
 * Caller needs to ensure that cpu_id remains valid, which can be
 * done by holding cpu_lock across this call, unless the caller
 * specifies CPU_CURRENT in which case the cpu_lock will be acquired
 * by thread_affinity_set and CPU->cpu_id will be the target CPU.
 */
void
thread_affinity_set(kthread_id_t t, int cpu_id)
{
        cpu_t *cp;
        int             c;

        ASSERT(!(t == curthread && t->t_weakbound_cpu != NULL));

        if (cpu_id == CPU_CURRENT) {
                VERIFY3P(t, ==, curthread);
                kpreempt_disable();
                cp = CPU;
        } else {
        if ((c = cpu_id) == CPU_CURRENT) {
                mutex_enter(&cpu_lock);
                cpu_id = CPU->cpu_id;
        }
                /*
                 * We should be asserting that cpu_lock is held here, but
                 * the NCA code doesn't acquire it.  The following assert
                 * should be uncommented when the NCA code is fixed.
                 *
                 * ASSERT(MUTEX_HELD(&cpu_lock));
                 */
                VERIFY((cpu_id >= 0) && (cpu_id < NCPU));
        ASSERT((cpu_id >= 0) && (cpu_id < NCPU));
                cp = cpu[cpu_id];

                /* user must provide a good cpu_id */
                VERIFY(cp != NULL);
        }

        ASSERT(cp != NULL);             /* user must provide a good cpu_id */
        /*
         * If there is already a hard affinity requested, and this affinity
         * conflicts with that, panic.
         */
        thread_lock(t);
        if (t->t_affinitycnt > 0 && t->t_bound_cpu != cp) {
                panic("affinity_set: setting %p but already bound to %p",
                    (void *)cp, (void *)t->t_bound_cpu);
        }
        t->t_affinitycnt++;
        t->t_bound_cpu = cp;

        /*
         * Make sure we're running on the right CPU.
         */
        if (cp != t->t_cpu || t != curthread) {
                ASSERT(cpu_id != CPU_CURRENT);
                force_thread_migrate(t);        /* drops thread lock */
        } else {
                thread_unlock(t);
        }

        if (cpu_id == CPU_CURRENT) {
                kpreempt_enable();
        }
        if (c == CPU_CURRENT)
                mutex_exit(&cpu_lock);
}
```

______*unchanged_portion_omitted_*