```
**********************************************************
    6735 Tue Aug 18 18:36:34 2015
new/usr/src/uts/i86pc/os/memnode.c
6138 don't abuse atomic_cas_*
**********************************************************
```

```
/*
 * 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.
 */

#include <sys/systm.h>
#include <sys/sysmacros.h>
#include <sys/bootconf.h>
#include <sys/atomic.h>
#include <sys/lgrp.h>
#include <sys/memlist.h>
#include <sys/memnode.h>
#include <sys/platform_module.h>
#include <vm/vm_dep.h>

int     max_mem_nodes = 1;

struct mem_node_conf mem_node_config[MAX_MEM_NODES];
int mem_node_pfn_shift;
/*
 * num_memnodes should be updated atomically and always >=
 * the number of bits in memnodes_mask or the algorithm may fail.
 */
uint16_t num_memnodes;
mnodeset_t memnodes_mask; /* assumes 8*(sizeof(mnodeset_t)) >= MAX_MEM_NODES */

/*
 * If set, mem_node_physalign should be a power of two, and
 * should reflect the minimum address alignment of each node.
 */
uint64_t mem_node_physalign;

/*
 * Platform hooks we will need.
 */

#pragma weak plat_build_mem_nodes
#pragma weak plat_slice_add
#pragma weak plat_slice_del

/*
```



```
 * Adjust the memnode config after a DR operation.
 *
 * It is rather tricky to do these updates since we can't
 * protect the memnode structures with locks, so we must
 * be mindful of the order in which updates and reads to
 * these values can occur.
 */

void
mem_node_add_slice(pfn_t start, pfn_t end)
{
        int mnode;
        mnodeset_t newmask, oldmask;

        /*
         * DR will pass us the first pfn that is allocatable.
         * We need to round down to get the real start of
         * the slice.
         */
        if (mem_node_physalign) {
                start &= ~(btop(mem_node_physalign) - 1);
                end = roundup(end, btop(mem_node_physalign)) - 1;
        }

        mnode = PFN_2_MEM_NODE(start);
        ASSERT(mnode >= 0 && mnode < max_mem_nodes);

        if (atomic_cas_32((uint32_t *)&mem_node_config[mnode].exists, 0, 1)) {
                /*
                 * Add slice to existing node.
                 */
                if (start < mem_node_config[mnode].physbase)
                        mem_node_config[mnode].physbase = start;
                if (end > mem_node_config[mnode].physmax)
                        mem_node_config[mnode].physmax = end;
        } else {
                mem_node_config[mnode].physbase = start;
                mem_node_config[mnode].physmax = end;
                atomic_inc_16(&num_memnodes);
                atomic_or_64(&memnodes_mask, 1ull << mnode);
                do {
                        oldmask = memnodes_mask;
                        newmask = memnodes_mask | (1ull << mnode);
                } while (atomic_cas_64(&memnodes_mask, oldmask, newmask) !=
                    oldmask);
        }

        /*
         * Inform the common lgrp framework about the new memory
         */
        lgrp_config(LGRP_CONFIG_MEM_ADD, mnode, MEM_NODE_2_LGRPHAND(mnode));
}

/*
 * Remove a PFN range from a memnode.  On some platforms,
 * the memnode will be created with physbase at the first
 * allocatable PFN, but later deleted with the MC slice
 * base address converted to a PFN, in which case we need
 * to assume physbase and up.
 */
void
mem_node_del_slice(pfn_t start, pfn_t end)
{
        int mnode;
        pgcnt_t delta_pgcnt, node_size;
        mnodeset_t omask, nmask;
```

```
122         if (mem_node_physalign) {
123                 start &= ~(btop(mem_node_physalign) - 1);
124                 end = roundup(end, btop(mem_node_physalign)) - 1;
125         }
126         mnode = PFN_2_MEM_NODE(start);

128         ASSERT(mnode >= 0 && mnode < max_mem_nodes);
129         ASSERT(mem_node_config[mnode].exists == 1);

131         delta_pgcnt = end - start;
132         node_size = mem_node_config[mnode].physmax -
133             mem_node_config[mnode].physbase;

135         if (node_size > delta_pgcnt) {
136                 /*
137                  * Subtract the slice from the memnode.
138                  */
139                 if (start <= mem_node_config[mnode].physbase)
140                         mem_node_config[mnode].physbase = end + 1;
141                 ASSERT(end <= mem_node_config[mnode].physmax);
142                 if (end == mem_node_config[mnode].physmax)
143                         mem_node_config[mnode].physmax = start - 1;
144         } else {
145                 /*
146                  * Let the common lgrp framework know this mnode is
147                  * leaving
148                  */
149                 lgrp_config(LGRP_CONFIG_MEM_DEL,
150                     mnode, MEM_NODE_2_LGRPHAND(mnode));

152                 /*
153                  * Delete the whole node.
154                  */
155                 ASSERT(MNODE_PGCNT(mnode) == 0);
156                 atomic_and_64(&memnodes_mask, ~(1ull << mnode));
162                 do {
163                         omask = memnodes_mask;
164                         nmask = omask & ~(1ull << mnode);
165                 } while (atomic_cas_64(&memnodes_mask, omask, nmask) != omask);
157                 atomic_dec_16(&num_memnodes);
158                 mem_node_config[mnode].exists = 0;
159         }
160 }
```

*unchanged_portion_omitted_*

```
210 /*
211  * Allocate an unassigned memnode.
212  */
213 int
214 mem_node_alloc()
215 {
216         int mnode;
226         mnodeset_t newmask, oldmask;

218         /*
219          * Find an unused memnode.  Update it atomically to prevent
220          * a first time memnode creation race.
221          */
222         for (mnode = 0; mnode < max_mem_nodes; mnode++)
223                 if (atomic_cas_32((uint32_t *)&mem_node_config[mnode].exists,
224                     0, 1) == 0)
225                         break;

227         if (mnode >= max_mem_nodes)
228                 panic("Out of free memnodes\n");
```



```
230         mem_node_config[mnode].physbase = (pfn_t)-1l;
231         mem_node_config[mnode].physmax = 0;
232         atomic_inc_16(&num_memnodes);
233         atomic_or_64(&memnodes_mask, 1ull << mnode);
243         do {
244                 oldmask = memnodes_mask;
245                 newmask = memnodes_mask | (1ull << mnode);
246         } while (atomic_cas_64(&memnodes_mask, oldmask, newmask) != oldmask);

235         return (mnode);
236 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   18786 Tue Aug 18 18:36:34 2015
new/usr/src/uts/i86pc/os/x_call.c
6138 don't abuse atomic_cas_*
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
  23  * Use is subject to license terms.
  24  */
  25 /*
  26  * Copyright (c) 2010, Intel Corporation.
  27  * All rights reserved.
  28  */

  30 #include <sys/types.h>
  31 #include <sys/param.h>
  32 #include <sys/t_lock.h>
  33 #include <sys/thread.h>
  34 #include <sys/cpuvar.h>
  35 #include <sys/x_call.h>
  36 #include <sys/xc_levels.h>
  37 #include <sys/cpu.h>
  38 #include <sys/psw.h>
  39 #include <sys/sunddi.h>
  40 #include <sys/debug.h>
  41 #include <sys/systm.h>
  42 #include <sys/archsystm.h>
  43 #include <sys/machsystm.h>
  44 #include <sys/mutex_impl.h>
  45 #include <sys/stack.h>
  46 #include <sys/promif.h>
  47 #include <sys/x86_archext.h>

  49 /*
  50  * Implementation for cross-processor calls via interprocessor interrupts
  51  *
  52  * This implementation uses a message passing architecture to allow multiple
  53  * concurrent cross calls to be in flight at any given time. We use the cmpxchg
  54  * instruction, aka atomic_cas_ptr(), to implement simple efficient work
  55  * queues for message passing between CPUs with almost no need for regular
  56  * locking.  See xc_extract() and xc_insert() below.
  57  *
  58  * The general idea is that initiating a cross call means putting a message
  59  * on a target(s) CPU's work queue. Any synchronization is handled by passing
  60  * the message back and forth between initiator and target(s).
  61  *
```



```
  62  * Every CPU has xc_work_cnt, which indicates it has messages to process.
  63  * This value is incremented as message traffic is initiated and decremented
  64  * with every message that finishes all processing.
  65  *
  66  * The code needs no mfence or other membar_*() calls. The uses of
  67  * atomic_cas_ptr(), atomic_inc_32_nv() and atomic_dec_32() for the message
  67  * atomic_cas_ptr(), atomic_cas_32() and atomic_dec_32() for the message
  68  * passing are implemented with LOCK prefix instructions which are
  69  * equivalent to mfence.
  70  *
  71  * One interesting aspect of this implmentation is that it allows 2 or more
  72  * CPUs to initiate cross calls to intersecting sets of CPUs at the same time.
  73  * The cross call processing by the CPUs will happen in any order with only
  74  * a guarantee, for xc_call() and xc_sync(), that an initiator won't return
  75  * from cross calls before all slaves have invoked the function.
  76  *
  77  * The reason for this asynchronous approach is to allow for fast global
  78  * TLB shootdowns. If all CPUs, say N, tried to do a global TLB invalidation
  79  * on a different Virtual Address at the same time. The old code required
  80  * N squared IPIs. With this method, depending on timing, it could happen
  81  * with just N IPIs.
  82  */

  84 /*
  85  * The default is to not enable collecting counts of IPI information, since
  86  * the updating of shared cachelines could cause excess bus traffic.
  87  */
  88 uint_t xc_collect_enable = 0;
  89 uint64_t xc_total_cnt = 0;      /* total #IPIs sent for cross calls */
  90 uint64_t xc_multi_cnt = 0;      /* # times we piggy backed on another IPI */

  92 /*
  93  * Values for message states. Here are the normal transitions. A transition
  94  * of "->" happens in the slave cpu and "=>" happens in the master cpu as
  95  * the messages are passed back and forth.
  96  *
  97  * FREE => ASYNC ->                       DONE => FREE
  98  * FREE => CALL ->                        DONE => FREE
  99  * FREE => SYNC -> WAITING => RELEASED -> DONE => FREE
 100  *
 101  * The interesing one above is ASYNC. You might ask, why not go directly
 102  * to FREE, instead of DONE. If it did that, it might be possible to exhaust
 103  * the master's xc_free list if a master can generate ASYNC messages faster
 104  * then the slave can process them. That could be handled with more complicated
 105  * handling. However since nothing important uses ASYNC, I've not bothered.
 106  */
 107 #define XC_MSG_FREE     (0)     /* msg in xc_free queue */
 108 #define XC_MSG_ASYNC    (1)     /* msg in slave xc_msgbox */
 109 #define XC_MSG_CALL     (2)     /* msg in slave xc_msgbox */
 110 #define XC_MSG_SYNC     (3)     /* msg in slave xc_msgbox */
 111 #define XC_MSG_WAITING  (4)     /* msg in master xc_msgbox or xc_waiters */
 112 #define XC_MSG_RELEASED (5)     /* msg in slave xc_msgbox */
 113 #define XC_MSG_DONE     (6)     /* msg in master xc_msgbox */

 115 /*
 116  * We allow for one high priority message at a time to happen in the system.
 117  * This is used for panic, kmdb, etc., so no locking is done.
 118  */
 119 static volatile cpuset_t xc_priority_set_store;
 120 static volatile ulong_t *xc_priority_set = CPUSET2BV(xc_priority_set_store);
 121 static xc_data_t xc_priority_data;

 123 /*
 124  * Wrappers to avoid C compiler warnings due to volatile. The atomic bit
 125  * operations don't accept volatile bit vectors - which is a bit silly.
 126  */
```

```
 127 #define XC_BT_SET(vector, b)    BT_ATOMIC_SET((ulong_t *)(vector), (b))
 128 #define XC_BT_CLEAR(vector, b)  BT_ATOMIC_CLEAR((ulong_t *)(vector), (b))

 130 /*
 131  * Decrement a CPU's work count
 132  */
 133 static void
 134 xc_decrement(struct machcpu *mcpu)
 135 {
 136         atomic_dec_32(&mcpu->xc_work_cnt);
 137 }

 139 /*
 140  * Increment a CPU's work count and return the old value
 141  */
 142 static int
 143 xc_increment(struct machcpu *mcpu)
 144 {
 145         return (atomic_inc_32_nv(&mcpu->xc_work_cnt) - 1);
 145         int old;
 146         do {
 147                 old = mcpu->xc_work_cnt;
 148         } while (atomic_cas_32(&mcpu->xc_work_cnt, old, old + 1) != old);
 149         return (old);
 146 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
    7852 Tue Aug 18 18:36:35 2015
new/usr/src/uts/sun4/os/memnode.c
6138 don't abuse atomic_cas_*
**********************************************************
```

```
/*
 * 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.
 */

#include <sys/systm.h>
#include <sys/platform_module.h>
#include <sys/sysmacros.h>
#include <sys/atomic.h>
#include <sys/memlist.h>
#include <sys/memnode.h>
#include <vm/vm_dep.h>

int max_mem_nodes = 1;          /* max memory nodes on this system */

struct mem_node_conf mem_node_config[MAX_MEM_NODES];
int mem_node_pfn_shift;
/*
 * num_memnodes should be updated atomically and always >=
 * the number of bits in memnodes_mask or the algorithm may fail.
 */
uint16_t num_memnodes;
mnodeset_t memnodes_mask; /* assumes 8*(sizeof(mnodeset_t)) >= MAX_MEM_NODES */

/*
 * If set, mem_node_physalign should be a power of two, and
 * should reflect the minimum address alignment of each node.
 */
uint64_t mem_node_physalign;

/*
 * Platform hooks we will need.
 */

#pragma weak plat_build_mem_nodes
#pragma weak plat_slice_add
#pragma weak plat_slice_del

/*
 * Adjust the memnode config after a DR operation.
 *
```



```
 * It is rather tricky to do these updates since we can't
 * protect the memnode structures with locks, so we must
 * be mindful of the order in which updates and reads to
 * these values can occur.
 */
void
mem_node_add_slice(pfn_t start, pfn_t end)
{
        int mnode;
        mnodeset_t newmask, oldmask;

        /*
         * DR will pass us the first pfn that is allocatable.
         * We need to round down to get the real start of
         * the slice.
         */
        if (mem_node_physalign) {
                start &= ~(btop(mem_node_physalign) - 1);
                end = roundup(end, btop(mem_node_physalign)) - 1;
        }

        mnode = PFN_2_MEM_NODE(start);
        ASSERT(mnode < max_mem_nodes);

        if (atomic_cas_32((uint32_t *)&mem_node_config[mnode].exists, 0, 1)) {
                /*
                 * Add slice to existing node.
                 */
                if (start < mem_node_config[mnode].physbase)
                        mem_node_config[mnode].physbase = start;
                if (end > mem_node_config[mnode].physmax)
                        mem_node_config[mnode].physmax = end;
        } else {
                mem_node_config[mnode].physbase = start;
                mem_node_config[mnode].physmax = end;
                atomic_inc_16(&num_memnodes);
                atomic_or_64(&memnodes_mask, 1ull << mnode);
                do {
                        oldmask = memnodes_mask;
                        newmask = memnodes_mask | (1ull << mnode);
                } while (atomic_cas_64(&memnodes_mask, oldmask, newmask) !=
                         oldmask);
        }
        /*
         * Let the common lgrp framework know about the new memory
         */
        lgrp_config(LGRP_CONFIG_MEM_ADD, mnode, MEM_NODE_2_LGRPHAND(mnode));
}

/*
 * Remove a PFN range from a memnode.  On some platforms,
 * the memnode will be created with physbase at the first
 * allocatable PFN, but later deleted with the MC slice
 * base address converted to a PFN, in which case we need
 * to assume physbase and up.
 */
void
mem_node_del_slice(pfn_t start, pfn_t end)
{
        int mnode;
        pgcnt_t delta_pgcnt, node_size;
        mnodeset_t omask, nmask;

        if (mem_node_physalign) {
                start &= ~(btop(mem_node_physalign) - 1);
                end = roundup(end, btop(mem_node_physalign)) - 1;
```

```
121         }
122         mnode = PFN_2_MEM_NODE(start);

124         ASSERT(mnode < max_mem_nodes);
125         ASSERT(mem_node_config[mnode].exists == 1);

127         delta_pgcnt = end - start;
128         node_size = mem_node_config[mnode].physmax -
129             mem_node_config[mnode].physbase;

131         if (node_size > delta_pgcnt) {
132                 /*
133                  * Subtract the slice from the memnode.
134                  */
135                 if (start <= mem_node_config[mnode].physbase)
136                         mem_node_config[mnode].physbase = end + 1;
137                 ASSERT(end <= mem_node_config[mnode].physmax);
138                 if (end == mem_node_config[mnode].physmax)
139                         mem_node_config[mnode].physmax = start - 1;
140         } else {

142                 /*
143                  * Let the common lgrp framework know the mnode is
144                  * leaving
145                  */
146                 lgrp_config(LGRP_CONFIG_MEM_DEL, mnode,
147                     MEM_NODE_2_LGRPHAND(mnode));

149                 /*
150                  * Delete the whole node.
151                  */
152                 ASSERT(MNODE_PGCNT(mnode) == 0);
153                 atomic_and_64(&memnodes_mask, ~(1ull << mnode));
159                 do {
160                         omask = memnodes_mask;
161                         nmask = omask & ~(1ull << mnode);
162                 } while (atomic_cas_64(&memnodes_mask, omask, nmask) != omask);
154                 atomic_dec_16(&num_memnodes);
155                 mem_node_config[mnode].exists = 0;
156         }
157 }
```

_____***unchanged_portion_omitted_***

```
201 /*
202  * Allocate an unassigned memnode.
203  */
204 int
205 mem_node_alloc()
206 {
207         int mnode;
217         mnodeset_t newmask, oldmask;

209         /*
210          * Find an unused memnode.  Update it atomically to prevent
211          * a first time memnode creation race.
212          */
213         for (mnode = 0; mnode < max_mem_nodes; mnode++)
214                 if (atomic_cas_32((uint32_t *)&mem_node_config[mnode].exists,
215                     0, 1) == 0)
216                         break;

218         if (mnode >= max_mem_nodes)
219                         panic("Out of free memnodes\n");

221         mem_node_config[mnode].physbase = (uint64_t)-1;
222         mem_node_config[mnode].physmax = 0;
```



```
223         atomic_inc_16(&num_memnodes);
224         atomic_or_64(&memnodes_mask, 1ull << mnode);
234         do {
235                 oldmask = memnodes_mask;
236                 newmask = memnodes_mask | (1ull << mnode);
237         } while (atomic_cas_64(&memnodes_mask, oldmask, newmask) != oldmask);

226         return (mnode);
227 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   26374 Tue Aug 18 18:36:35 2015
new/usr/src/uts/sun4/vm/vm_dep.c
6138 don't abuse atomic_cas_*
**********************************************************
______unchanged_portion_omitted_

 878 /*
 879  * To select our starting bin, we stride through the bins with a stride
 880  * of 337.  Why 337?  It's prime, it's largeish, and it performs well both
 881  * in simulation and practice for different workloads on varying cache sizes.
 882  */
 883 uint32_t color_start_current = 0;
 884 uint32_t color_start_stride = 337;
 885 int color_start_random = 0;

 887 /* ARGSUSED */
 888 uint_t
 889 get_color_start(struct as *as)
 890 {
 891         uint32_t old, new;

 891         if (consistent_coloring == 2 || color_start_random) {
 892                 return ((uint_t)(((gettick()) << (vac_shift - MMU_PAGESHIFT)) &
 893                     (hw_page_array[0].hp_colors - 1)));
 894         }

 896         return ((uint_t)atomic_add_32_nv(&color_start_current,
 897             color_start_stride << (vac_shift - MMU_PAGESHIFT)));
 898         do {
 899                 old = color_start_current;
 900                 new = old + (color_start_stride << (vac_shift - MMU_PAGESHIFT));
 901         } while (atomic_cas_32(&color_start_current, old, new) != old);

 903         return ((uint_t)(new));
 898 }
______unchanged_portion_omitted_
```