

```

*****
6735 Tue Aug 18 18:36:34 2015
new/usr/src/uts/i86pc/os/memnode.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 2010 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 #include <sys/systm.h>
27 #include <sys/sysmacros.h>
28 #include <sys/bootconf.h>
29 #include <sys/atomic.h>
30 #include <sys/lgrp.h>
31 #include <sys/memlist.h>
32 #include <sys/memnode.h>
33 #include <sys/platform_module.h>
34 #include <vm/vm_dep.h>

36 int      max_mem_nodes = 1;

38 struct mem_node_conf mem_node_config[MAX_MEM_NODES];
39 int mem_node_pfn_shift;
40 /*
41  * num_memnodes should be updated atomically and always >=
42  * the number of bits in memnodes_mask or the algorithm may fail.
43  */
44 uint16_t num_memnodes;
45 mnodeset_t memnodes_mask; /* assumes 8*(sizeof(mnodeset_t)) >= MAX_MEM_NODES */

47 /*
48  * If set, mem_node_physalign should be a power of two, and
49  * should reflect the minimum address alignment of each node.
50  */
51 uint64_t mem_node_physalign;

53 /*
54  * Platform hooks we will need.
55  */

57 #pragma weak plat_build_mem_nodes
58 #pragma weak plat_slice_add
59 #pragma weak plat_slice_del

61 /*

```

```

62  * Adjust the memnode config after a DR operation.
63  *
64  * It is rather tricky to do these updates since we can't
65  * protect the memnode structures with locks, so we must
66  * be mindful of the order in which updates and reads to
67  * these values can occur.
68  */

70 void
71 mem_node_add_slice(pfn_t start, pfn_t end)
72 {
73     int mnode;
74     mnodeset_t newmask, oldmask;

75     /*
76      * DR will pass us the first pfn that is allocatable.
77      * We need to round down to get the real start of
78      * the slice.
79      */
80     if (mem_node_physalign) {
81         start &= ~(btop(mem_node_physalign) - 1);
82         end = roundup(end, btop(mem_node_physalign)) - 1;
83     }

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

88     if (atomic_cas_32((uint32_t *)&mem_node_config[mnode].exists, 0, 1)) {
89         /*
90          * Add slice to existing node.
91          */
92         if (start < mem_node_config[mnode].physbase)
93             mem_node_config[mnode].physbase = start;
94         if (end > mem_node_config[mnode].physmax)
95             mem_node_config[mnode].physmax = end;
96     } else {
97         mem_node_config[mnode].physbase = start;
98         mem_node_config[mnode].physmax = end;
99         atomic_inc_16(&num_memnodes);
100        atomic_or_64(&memnodes_mask, 1ull << mnode);
101        do {
102            oldmask = memnodes_mask;
103            newmask = memnodes_mask | (1ull << mnode);
104        } while (atomic_cas_64(&memnodes_mask, oldmask, newmask) !=
105                oldmask);
106    }

107 }

109 /*
110  * Remove a PFN range from a memnode. On some platforms,
111  * the memnode will be created with physbase at the first
112  * allocatable PFN, but later deleted with the MC slice
113  * base address converted to a PFN, in which case we need
114  * to assume physbase and up.
115  */
116 void
117 mem_node_del_slice(pfn_t start, pfn_t end)
118 {
119     int mnode;
120     pgcnt_t delta_pgcnt, node_size;
121     mnodeset_t omask, rmask;

```

```

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));
157         do {
158             omask = memnodes_mask;
159             nmask = omask & ~(1ull << mnode);
160         } while (atomic_cas_64(&memnodes_mask, omask, nmask) != omask);
161         atomic_dec_16(&num_memnodes);
162         mem_node_config[mnode].exists = 0;
163     }
164 }
165 }
166 }
167 }
168 }
169 }
170 }
171 }
172 }
173 }
174 }
175 }
176 }
177 }
178 }
179 }
180 }
181 }
182 }
183 }
184 }
185 }
186 }
187 }
188 }
189 }
190 }
191 }
192 }
193 }
194 }
195 }
196 }
197 }
198 }
199 }
200 }
201 }
202 }
203 }
204 }
205 }
206 }
207 }
208 }
209 }
210 }
211 }
212 }
213 }
214 }
215 }
216 }
217 }
218 }
219 }
220 }
221 }
222 }
223 }
224 }
225 }
226 }
227 }
228 }

```

```

230     mem_node_config[mnode].physbase = (pfn_t)-1;
231     mem_node_config[mnode].physmax = 0;
232     atomic_inc_16(&num_memnodes);
233     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);

238     return (mnode);
239 }
240 }
241 }
242 }
243 }
244 }
245 }
246 }
247 }
248 }
249 }
250 }
251 }
252 }
253 }
254 }
255 }
256 }
257 }
258 }
259 }
260 }
261 }
262 }
263 }
264 }
265 }
266 }
267 }
268 }
269 }
270 }
271 }
272 }
273 }
274 }
275 }
276 }
277 }
278 }
279 }
280 }
281 }
282 }
283 }
284 }
285 }
286 }
287 }
288 }
289 }
290 }
291 }
292 }
293 }
294 }
295 }
296 }
297 }
298 }
299 }
300 }
301 }
302 }
303 }
304 }
305 }
306 }
307 }
308 }
309 }
310 }
311 }
312 }
313 }
314 }
315 }
316 }
317 }
318 }
319 }
320 }
321 }
322 }
323 }
324 }
325 }
326 }
327 }
328 }
329 }
330 }
331 }
332 }
333 }
334 }
335 }
336 }
337 }
338 }
339 }
340 }
341 }
342 }
343 }
344 }
345 }
346 }
347 }
348 }
349 }
350 }
351 }
352 }
353 }
354 }
355 }
356 }
357 }
358 }
359 }
360 }
361 }
362 }
363 }
364 }
365 }
366 }
367 }
368 }
369 }
370 }
371 }
372 }
373 }
374 }
375 }
376 }
377 }
378 }
379 }
380 }
381 }
382 }
383 }
384 }
385 }
386 }
387 }
388 }
389 }
390 }
391 }
392 }
393 }
394 }
395 }
396 }
397 }
398 }
399 }
400 }
401 }
402 }
403 }
404 }
405 }
406 }
407 }
408 }
409 }
410 }
411 }
412 }
413 }
414 }
415 }
416 }
417 }
418 }
419 }
420 }
421 }
422 }
423 }
424 }
425 }
426 }
427 }
428 }
429 }
430 }
431 }
432 }
433 }
434 }
435 }
436 }
437 }
438 }
439 }
440 }
441 }
442 }
443 }
444 }
445 }
446 }
447 }
448 }
449 }
450 }
451 }
452 }
453 }
454 }
455 }
456 }
457 }
458 }
459 }
460 }
461 }
462 }
463 }
464 }
465 }
466 }
467 }
468 }
469 }
470 }
471 }
472 }
473 }
474 }
475 }
476 }
477 }
478 }
479 }
480 }
481 }
482 }
483 }
484 }
485 }
486 }
487 }
488 }
489 }
490 }
491 }
492 }
493 }
494 }
495 }
496 }
497 }
498 }
499 }
500 }
501 }
502 }
503 }
504 }
505 }
506 }
507 }
508 }
509 }
510 }
511 }
512 }
513 }
514 }
515 }
516 }
517 }
518 }
519 }
520 }
521 }
522 }
523 }
524 }
525 }
526 }
527 }
528 }
529 }
530 }
531 }
532 }
533 }
534 }
535 }
536 }
537 }
538 }
539 }
540 }
541 }
542 }
543 }
544 }
545 }
546 }
547 }
548 }
549 }
550 }
551 }
552 }
553 }
554 }
555 }
556 }
557 }
558 }
559 }
560 }
561 }
562 }
563 }
564 }
565 }
566 }
567 }
568 }
569 }
570 }
571 }
572 }
573 }
574 }
575 }
576 }
577 }
578 }
579 }
580 }
581 }
582 }
583 }
584 }
585 }
586 }
587 }
588 }
589 }
590 }
591 }
592 }
593 }
594 }
595 }
596 }
597 }
598 }
599 }
600 }
601 }
602 }
603 }
604 }
605 }
606 }
607 }
608 }
609 }
610 }
611 }
612 }
613 }
614 }
615 }
616 }
617 }
618 }
619 }
620 }
621 }
622 }
623 }
624 }
625 }
626 }
627 }
628 }
629 }
630 }
631 }
632 }
633 }
634 }
635 }
636 }
637 }
638 }
639 }
640 }
641 }
642 }
643 }
644 }
645 }
646 }
647 }
648 }
649 }
650 }
651 }
652 }
653 }
654 }
655 }
656 }
657 }
658 }
659 }
660 }
661 }
662 }
663 }
664 }
665 }
666 }
667 }
668 }
669 }
670 }
671 }
672 }
673 }
674 }
675 }
676 }
677 }
678 }
679 }
680 }
681 }
682 }
683 }
684 }
685 }
686 }
687 }
688 }
689 }
690 }
691 }
692 }
693 }
694 }
695 }
696 }
697 }
698 }
699 }
700 }
701 }
702 }
703 }
704 }
705 }
706 }
707 }
708 }
709 }
710 }
711 }
712 }
713 }
714 }
715 }
716 }
717 }
718 }
719 }
720 }
721 }
722 }
723 }
724 }
725 }
726 }
727 }
728 }
729 }
730 }
731 }
732 }
733 }
734 }
735 }
736 }
737 }
738 }
739 }
740 }
741 }
742 }
743 }
744 }
745 }
746 }
747 }
748 }
749 }
750 }
751 }
752 }
753 }
754 }
755 }
756 }
757 }
758 }
759 }
760 }
761 }
762 }
763 }
764 }
765 }
766 }
767 }
768 }
769 }
770 }
771 }
772 }
773 }
774 }
775 }
776 }
777 }
778 }
779 }
780 }
781 }
782 }
783 }
784 }
785 }
786 }
787 }
788 }
789 }
790 }
791 }
792 }
793 }
794 }
795 }
796 }
797 }
798 }
799 }
800 }
801 }
802 }
803 }
804 }
805 }
806 }
807 }
808 }
809 }
810 }
811 }
812 }
813 }
814 }
815 }
816 }
817 }
818 }
819 }
820 }
821 }
822 }
823 }
824 }
825 }
826 }
827 }
828 }
829 }
830 }
831 }
832 }
833 }
834 }
835 }
836 }
837 }
838 }
839 }
840 }
841 }
842 }
843 }
844 }
845 }
846 }
847 }
848 }
849 }
850 }
851 }
852 }
853 }
854 }
855 }
856 }
857 }
858 }
859 }
860 }
861 }
862 }
863 }
864 }
865 }
866 }
867 }
868 }
869 }
870 }
871 }
872 }
873 }
874 }
875 }
876 }
877 }
878 }
879 }
880 }
881 }
882 }
883 }
884 }
885 }
886 }
887 }
888 }
889 }
890 }
891 }
892 }
893 }
894 }
895 }
896 }
897 }
898 }
899 }
900 }
901 }
902 }
903 }
904 }
905 }
906 }
907 }
908 }
909 }
910 }
911 }
912 }
913 }
914 }
915 }
916 }
917 }
918 }
919 }
920 }
921 }
922 }
923 }
924 }
925 }
926 }
927 }
928 }
929 }
930 }
931 }
932 }
933 }
934 }
935 }
936 }
937 }
938 }
939 }
940 }
941 }
942 }
943 }
944 }
945 }
946 }
947 }
948 }
949 }
950 }
951 }
952 }
953 }
954 }
955 }
956 }
957 }
958 }
959 }
960 }
961 }
962 }
963 }
964 }
965 }
966 }
967 }
968 }
969 }
970 }
971 }
972 }
973 }
974 }
975 }
976 }
977 }
978 }
979 }
980 }
981 }
982 }
983 }
984 }
985 }
986 }
987 }
988 }
989 }
990 }
991 }
992 }
993 }
994 }
995 }
996 }
997 }
998 }
999 }
1000 }

```

new/usr/src/uts/i86pc/os/x_call.c

1

```
*****
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/system.h>
42 #include <sys/archsystem.h>
43 #include <sys/machsystem.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 *
```

new/usr/src/uts/i86pc/os/x_call.c

2

```
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
68 * atomic_cas_ptr(), atomic_cas_32() and atomic_dec_32() for the message
69 * passing are implemented with LOCK prefix instructions which are
70 * equivalent to mfence.
71 *
72 * One interesting aspect of this implementation is that it allows 2 or more
73 * CPUs to initiate cross calls to intersecting sets of CPUs at the same time.
74 * The cross call processing by the CPUs will happen in any order with only
75 * a guarantee, for xc_call() and xc_sync(), that an initiator won't return
76 * from cross calls before all slaves have invoked the function.
77 *
78 * The reason for this asynchronous approach is to allow for fast global
79 * TLB shutdowns. If all CPUs, say N, tried to do a global TLB invalidation
80 * on a different Virtual Address at the same time. The old code required
81 * N squared IPIs. With this method, depending on timing, it could happen
82 * with just N IPIs.
83 */

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 interesting 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 * than 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);
146     int old;
147     do {
148         old = mcpu->xc_work_cnt;
149     } while (atomic_cas_32(&mcpu->xc_work_cnt, old, old + 1) != old);
150     return (old);
151 }
152
153 _____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_*
*****
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 2010 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 #include <sys/systm.h>
27 #include <sys/platform_module.h>
28 #include <sys/sysmacros.h>
29 #include <sys/atomic.h>
30 #include <sys/memlist.h>
31 #include <sys/memnode.h>
32 #include <vm/vm_dep.h>

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

36 struct mem_node_conf mem_node_config[MAX_MEM_NODES];
37 int mem_node_pfn_shift;
38 /*
39  * num_memnodes should be updated atomically and always >=
40  * the number of bits in memnodes_mask or the algorithm may fail.
41  */
42 uint16_t num_memnodes;
43 mnodeset_t memnodes_mask; /* assumes 8*(sizeof(mnodeset_t)) >= MAX_MEM_NODES */

45 /*
46  * If set, mem_node_physalign should be a power of two, and
47  * should reflect the minimum address alignment of each node.
48  */
49 uint64_t mem_node_physalign;

51 /*
52  * Platform hooks we will need.
53  */

55 #pragma weak plat_build_mem_nodes
56 #pragma weak plat_slice_add
57 #pragma weak plat_slice_del

59 /*
60  * Adjust the memnode config after a DR operation.
61  */

```

```

62  * It is rather tricky to do these updates since we can't
63  * protect the memnode structures with locks, so we must
64  * be mindful of the order in which updates and reads to
65  * these values can occur.
66  */
67 void
68 mem_node_add_slice(pfn_t start, pfn_t end)
69 {
70     int mnode;
71     mnodeset_t newmask, oldmask;

72     /*
73      * DR will pass us the first pfn that is allocatable.
74      * We need to round down to get the real start of
75      * the slice.
76      */
77     if (mem_node_physalign) {
78         start &= ~(btop(mem_node_physalign) - 1);
79         end = roundup(end, btop(mem_node_physalign)) - 1;
80     }

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

85     if (atomic_cas_32((uint32_t *)&mem_node_config[mnode].exists, 0, 1)) {
86         /*
87          * Add slice to existing node.
88          */
89         if (start < mem_node_config[mnode].physbase)
90             mem_node_config[mnode].physbase = start;
91         if (end > mem_node_config[mnode].physmax)
92             mem_node_config[mnode].physmax = end;
93     } else {
94         mem_node_config[mnode].physbase = start;
95         mem_node_config[mnode].physmax = end;
96         atomic_inc_16(&num_memnodes);
97         atomic_or_64(&memnodes_mask, 1ull << mnode);
98         do {
99             oldmask = memnodes_mask;
100            newmask = memnodes_mask | (1ull << mnode);
101            } while (atomic_cas_64(&memnodes_mask, oldmask, newmask) !=
102                oldmask);
103     }

105     /*
106      * Remove a PFN range from a memnode. On some platforms,
107      * the memnode will be created with physbase at the first
108      * allocatable PFN, but later deleted with the MC slice
109      * base address converted to a PFN, in which case we need
110      * to assume physbase and up.
111      */
112     void
113     mem_node_del_slice(pfn_t start, pfn_t end)
114     {
115         int mnode;
116         pgcnt_t delta_pgcnt, node_size;
117         mnodeset_t omask, nmask;

118         if (mem_node_physalign) {
119             start &= ~(btop(mem_node_physalign) - 1);
120             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));
154         do {
155             omask = memnodes_mask;
156             nmask = omask & ~(1ull << mnode);
157         } while (atomic_cas_64(&memnodes_mask, omask, nmask) != omask);
158         atomic_dec_16(&num_memnodes);
159         mem_node_config[mnode].exists = 0;
160     }
161 }
162 }
163 }
164 }
165 }
166 }
167 }
168 }
169 }
170 }
171 }
172 }
173 }
174 }
175 }
176 }
177 }
178 }
179 }
180 }
181 }
182 }
183 }
184 }
185 }
186 }
187 }
188 }
189 }
190 }
191 }
192 }
193 }
194 }
195 }
196 }
197 }
198 }
199 }
200 }
201 /*
202 * Allocate an unassigned memnode.
203 */
204 int
205 mem_node_alloc()
206 {
207     int mnode;
208     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);
225     do {
226         oldmask = memnodes_mask;
227         newmask = memnodes_mask | (1ull << mnode);
228     } while (atomic_cas_64(&memnodes_mask, oldmask, newmask) != oldmask);

229     return (mnode);
230 }
231 }
232 }
233 }
234 }
235 }
236 }
237 }
238 }
239 }
240 }
241 }
242 }
243 }
244 }
245 }
246 }
247 }
248 }
249 }
250 }
251 }
252 }
253 }
254 }
255 }
256 }
257 }
258 }
259 }
260 }
261 }
262 }
263 }
264 }
265 }
266 }
267 }
268 }
269 }
270 }
271 }
272 }
273 }
274 }
275 }
276 }
277 }
278 }
279 }
280 }
281 }
282 }
283 }
284 }
285 }
286 }
287 }
288 }
289 }
290 }
291 }
292 }
293 }
294 }
295 }
296 }
297 }
298 }
299 }
300 }
301 }
302 }
303 }
304 }
305 }
306 }
307 }
308 }
309 }
310 }
311 }
312 }
313 }
314 }
315 }
316 }
317 }
318 }
319 }
320 }
321 }
322 }
323 }
324 }
325 }
326 }
327 }
328 }
329 }
330 }
331 }
332 }
333 }
334 }
335 }
336 }
337 }
338 }
339 }
340 }
341 }
342 }
343 }
344 }
345 }
346 }
347 }
348 }
349 }
350 }
351 }
352 }
353 }
354 }
355 }
356 }
357 }
358 }
359 }
360 }
361 }
362 }
363 }
364 }
365 }
366 }
367 }
368 }
369 }
370 }
371 }
372 }
373 }
374 }
375 }
376 }
377 }
378 }
379 }
380 }
381 }
382 }
383 }
384 }
385 }
386 }
387 }
388 }
389 }
390 }
391 }
392 }
393 }
394 }
395 }
396 }
397 }
398 }
399 }
400 }
401 }
402 }
403 }
404 }
405 }
406 }
407 }
408 }
409 }
410 }
411 }
412 }
413 }
414 }
415 }
416 }
417 }
418 }
419 }
420 }
421 }
422 }
423 }
424 }
425 }
426 }
427 }
428 }
429 }
430 }
431 }
432 }
433 }
434 }
435 }
436 }
437 }
438 }
439 }
440 }
441 }
442 }
443 }
444 }
445 }
446 }
447 }
448 }
449 }
450 }
451 }
452 }
453 }
454 }
455 }
456 }
457 }
458 }
459 }
460 }
461 }
462 }
463 }
464 }
465 }
466 }
467 }
468 }
469 }
470 }
471 }
472 }
473 }
474 }
475 }
476 }
477 }
478 }
479 }
480 }
481 }
482 }
483 }
484 }
485 }
486 }
487 }
488 }
489 }
490 }
491 }
492 }
493 }
494 }
495 }
496 }
497 }
498 }
499 }
500 }
501 }
502 }
503 }
504 }
505 }
506 }
507 }
508 }
509 }
510 }
511 }
512 }
513 }
514 }
515 }
516 }
517 }
518 }
519 }
520 }
521 }
522 }
523 }
524 }
525 }
526 }
527 }
528 }
529 }
530 }
531 }
532 }
533 }
534 }
535 }
536 }
537 }
538 }
539 }
540 }
541 }
542 }
543 }
544 }
545 }
546 }
547 }
548 }
549 }
550 }
551 }
552 }
553 }
554 }
555 }
556 }
557 }
558 }
559 }
560 }
561 }
562 }
563 }
564 }
565 }
566 }
567 }
568 }
569 }
570 }
571 }
572 }
573 }
574 }
575 }
576 }
577 }
578 }
579 }
580 }
581 }
582 }
583 }
584 }
585 }
586 }
587 }
588 }
589 }
590 }
591 }
592 }
593 }
594 }
595 }
596 }
597 }
598 }
599 }
600 }
601 }
602 }
603 }
604 }
605 }
606 }
607 }
608 }
609 }
610 }
611 }
612 }
613 }
614 }
615 }
616 }
617 }
618 }
619 }
620 }
621 }
622 }
623 }
624 }
625 }
626 }
627 }
628 }
629 }
630 }
631 }
632 }
633 }
634 }
635 }
636 }
637 }
638 }
639 }
640 }
641 }
642 }
643 }
644 }
645 }
646 }
647 }
648 }
649 }
650 }
651 }
652 }
653 }
654 }
655 }
656 }
657 }
658 }
659 }
660 }
661 }
662 }
663 }
664 }
665 }
666 }
667 }
668 }
669 }
670 }
671 }
672 }
673 }
674 }
675 }
676 }
677 }
678 }
679 }
680 }
681 }
682 }
683 }
684 }
685 }
686 }
687 }
688 }
689 }
690 }
691 }
692 }
693 }
694 }
695 }
696 }
697 }
698 }
699 }
700 }
701 }
702 }
703 }
704 }
705 }
706 }
707 }
708 }
709 }
710 }
711 }
712 }
713 }
714 }
715 }
716 }
717 }
718 }
719 }
720 }
721 }
722 }
723 }
724 }
725 }
726 }
727 }
728 }
729 }
730 }
731 }
732 }
733 }
734 }
735 }
736 }
737 }
738 }
739 }
740 }
741 }
742 }
743 }
744 }
745 }
746 }
747 }
748 }
749 }
750 }
751 }
752 }
753 }
754 }
755 }
756 }
757 }
758 }
759 }
760 }
761 }
762 }
763 }
764 }
765 }
766 }
767 }
768 }
769 }
770 }
771 }
772 }
773 }
774 }
775 }
776 }
777 }
778 }
779 }
780 }
781 }
782 }
783 }
784 }
785 }
786 }
787 }
788 }
789 }
790 }
791 }
792 }
793 }
794 }
795 }
796 }
797 }
798 }
799 }
800 }
801 }
802 }
803 }
804 }
805 }
806 }
807 }
808 }
809 }
810 }
811 }
812 }
813 }
814 }
815 }
816 }
817 }
818 }
819 }
820 }
821 }
822 }
823 }
824 }
825 }
826 }
827 }
828 }
829 }
830 }
831 }
832 }
833 }
834 }
835 }
836 }
837 }
838 }
839 }
840 }
841 }
842 }
843 }
844 }
845 }
846 }
847 }
848 }
849 }
850 }
851 }
852 }
853 }
854 }
855 }
856 }
857 }
858 }
859 }
860 }
861 }
862 }
863 }
864 }
865 }
866 }
867 }
868 }
869 }
870 }
871 }
872 }
873 }
874 }
875 }
876 }
877 }
878 }
879 }
880 }
881 }
882 }
883 }
884 }
885 }
886 }
887 }
888 }
889 }
890 }
891 }
892 }
893 }
894 }
895 }
896 }
897 }
898 }
899 }
900 }
901 }
902 }
903 }
904 }
905 }
906 }
907 }
908 }
909 }
910 }
911 }
912 }
913 }
914 }
915 }
916 }
917 }
918 }
919 }
920 }
921 }
922 }
923 }
924 }
925 }
926 }
927 }
928 }
929 }
930 }
931 }
932 }
933 }
934 }
935 }
936 }
937 }
938 }
939 }
940 }
941 }
942 }
943 }
944 }
945 }
946 }
947 }
948 }
949 }
950 }
951 }
952 }
953 }
954 }
955 }
956 }
957 }
958 }
959 }
960 }
961 }
962 }
963 }
964 }
965 }
966 }
967 }
968 }
969 }
970 }
971 }
972 }
973 }
974 }
975 }
976 }
977 }
978 }
979 }
980 }
981 }
982 }
983 }
984 }
985 }
986 }
987 }
988 }
989 }
990 }
991 }
992 }
993 }
994 }
995 }
996 }
997 }
998 }
999 }
1000 }

```

_____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_____
```