****
110314 Sun Apr 22 20:26:17 2018
new/usr/src/cmd/mdb/common/modules/genunix/kmem.c
9525 kmem_dump_size is a corrupting influence
****

```
/*
 * 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.
 */

/*
 * Copyright 2018 Joyent, Inc.  All rights reserved.
 * Copyright 2011 Joyent, Inc.  All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#include <mdb/mdb_param.h>
#include <mdb/mdb_modapi.h>
#include <mdb/mdb_ctf.h>
#include <mdb/mdb_whatis.h>
#include <sys/cpuvar.h>
#include <sys/kmem_impl.h>
#include <sys/vmem_impl.h>
#include <sys/machelf.h>
#include <sys/modctl.h>
#include <sys/kobj.h>
#include <sys/panic.h>
#include <sys/stack.h>
#include <sys/sysmacros.h>
#include <vm/page.h>

#include "avl.h"
#include "combined.h"
#include "dist.h"
#include "kmem.h"
#include "list.h"

#define dprintf(x) if (mdb_debug_level) { \
        mdb_printf("kmem debug: ");  \
        /*CSTYLED*/\
        mdb_printf x ;\
}
```

_____unchanged_portion_omitted_

```
typedef struct kmem_verify {
        uint64_t *kmv_buf;              /* buffer to read cache contents into */
        size_t kmv_size;                /* number of bytes in kmv_buf */
        int kmv_corruption;             /* > 0 if corruption found. */
        uint_t kmv_flags;               /* dcmd flags */
        int kmv_besilent;               /* report actual corruption sites */
        struct kmem_cache kmv_cache;    /* the cache we're operating on */
} kmem_verify_t;
```


_____unchanged_portion_omitted_

```
/*
 * verify_free()
 *      verify the integrity of a free block of memory by checking
 *      that it is filled with 0xdeadbeef and that its buftag is sane.
 */
/*ARGSUSED1*/
static int
verify_free(uintptr_t addr, const void *data, void *private)
{
        kmem_verify_t *kmv = (kmem_verify_t *)private;
        uint64_t *buf = kmv->kmv_buf;   /* buf to validate */
        int64_t corrupt;                /* corruption offset */
        kmem_buftag_t *buftagp;         /* ptr to buftag */
        kmem_cache_t *cp = &kmv->kmv_cache;
        boolean_t besilent = !!(kmv->kmv_flags & (DCMD_LOOP | DCMD_PIPE_OUT));
        int besilent = kmv->kmv_besilent;

        /*LINTED*/
        buftagp = KMEM_BUFTAG(cp, buf);

        /*
         * Read the buffer to check.
         */
        if (mdb_vread(buf, kmv->kmv_size, addr) == -1) {
                if (!besilent)
                        mdb_warn("couldn't read %p", addr);
                return (WALK_NEXT);
        }

        if ((corrupt = verify_pattern(buf, cp->cache_verify,
            KMEM_FREE_PATTERN)) >= 0) {
                if (!besilent)
                        mdb_printf("buffer %p (free) seems corrupted, at %p\n",
                            addr, (uintptr_t)addr + corrupt);
                goto corrupt;
        }
        /*
         * When KMF_LITE is set, buftagp->bt_redzone is used to hold
         * the first bytes of the buffer, hence we cannot check for red
         * zone corruption.
         */
        if ((cp->cache_flags & (KMF_HASH | KMF_LITE)) == KMF_HASH &&
            buftagp->bt_redzone != KMEM_REDZONE_PATTERN) {
                if (!besilent)
                        mdb_printf("buffer %p (free) seems to "
                            "have a corrupt redzone pattern\n", addr);
                goto corrupt;
        }

        /*
         * confirm bufctl pointer integrity.
         */
        if (verify_buftag(buftagp, KMEM_BUFTAG_FREE) == -1) {
                if (!besilent)
                        mdb_printf("buffer %p (free) has a corrupt "
                            "buftag\n", addr);
                goto corrupt;
        }
```

```
3104         return (WALK_NEXT);
3105 corrupt:
3106         if (kmv->kmv_flags & DCMD_PIPE_OUT)
3107                 mdb_printf("%p\n", addr);
3108         kmv->kmv_corruption++;
3109         return (WALK_NEXT);
3110 }

3112 /*
3113  * verify_alloc()
3114  *      Verify that the buftag of an allocated buffer makes sense with respect
3115  *      to the buffer.
3116  */
3117 /*ARGSUSED1*/
3118 static int
3119 verify_alloc(uintptr_t addr, const void *data, void *private)
3120 {
3121         kmem_verify_t *kmv = (kmem_verify_t *)private;
3122         kmem_cache_t *cp = &kmv->kmv_cache;
3123         uint64_t *buf = kmv->kmv_buf;   /* buf to validate */
3124         /*LINTED*/
3125         kmem_buftag_t *buftagp = KMEM_BUFTAG(cp, buf);
3126         uint32_t *ip = (uint32_t *)buftagp;
3127         uint8_t *bp = (uint8_t *)buf;
3128         int looks_ok = 0, size_ok = 1;  /* flags for finding corruption */
3129         boolean_t besilent = !!(kmv->kmv_flags & (DCMD_LOOP | DCMD_PIPE_OUT));
3127         int besilent = kmv->kmv_besilent;

3131         /*
3132          * Read the buffer to check.
3133          */
3134         if (mdb_vread(buf, kmv->kmv_size, addr) == -1) {
3135                 if (!besilent)
3136                         mdb_warn("couldn't read %p", addr);
3137                 return (WALK_NEXT);
3138         }

3140         /*
3141          * There are two cases to handle:
3142          * 1. If the buf was alloc'd using kmem_cache_alloc, it will have
3143          *    0xfeedfacefeedface at the end of it
3144          * 2. If the buf was alloc'd using kmem_alloc, it will have
3145          *    0xbb just past the end of the region in use.  At the buftag,
3146          *    it will have 0xfeedface (or, if the whole buffer is in use,
3147          *    0xfeedface & bb000000 or 0xfeedfacf & 000000bb depending on
3148          *    endianness), followed by 32 bits containing the offset of the
3149          *    0xbb byte in the buffer.
3150          *
3151          * Finally, the two 32-bit words that comprise the second half of the
3152          * buftag should xor to KMEM_BUFTAG_ALLOC
3153          */

3155         if (buftagp->bt_redzone == KMEM_REDZONE_PATTERN)
3156                 looks_ok = 1;
3157         else if (!KMEM_SIZE_VALID(ip[1]))
3158                 size_ok = 0;
3159         else if (bp[KMEM_SIZE_DECODE(ip[1])] == KMEM_REDZONE_BYTE)
3160                 looks_ok = 1;
3161         else
3162                 size_ok = 0;

3164         if (!size_ok) {
3165                 if (!besilent)
3166                         mdb_printf("buffer %p (allocated) has a corrupt "
3167                             "redzone size encoding\n", addr);
```



```
3168                 goto corrupt;
3169         }

3171         if (!looks_ok) {
3172                 if (!besilent)
3173                         mdb_printf("buffer %p (allocated) has a corrupt "
3174                             "redzone signature\n", addr);
3175                 goto corrupt;
3176         }

3178         if (verify_buftag(buftagp, KMEM_BUFTAG_ALLOC) == -1) {
3179                 if (!besilent)
3180                         mdb_printf("buffer %p (allocated) has a "
3181                             "corrupt buftag\n", addr);
3182                 goto corrupt;
3183         }

3185         return (WALK_NEXT);
3186 corrupt:
3187         if (kmv->kmv_flags & DCMD_PIPE_OUT)
3188                 mdb_printf("%p\n", addr);

3190         kmv->kmv_corruption++;
3191         return (WALK_NEXT);
3192 }

3194 /*ARGSUSED2*/
3195 int
3196 kmem_verify(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
3197 {
3198         if (flags & DCMD_ADDRSPEC) {
3199                 int check_alloc = 0, check_free = 0;
3200                 kmem_verify_t kmv;

3202                 if (mdb_vread(&kmv.kmv_cache, sizeof (kmv.kmv_cache),
3203                     addr) == -1) {
3204                         mdb_warn("couldn't read kmem_cache %p", addr);
3205                         return (DCMD_ERR);
3206                 }

3208                 if ((kmv.kmv_cache.cache_dump.kd_unsafe ||
3209                     kmv.kmv_cache.cache_dump.kd_alloc_fails) &&
3210                     !(flags & (DCMD_LOOP | DCMD_PIPE_OUT))) {
3211                         mdb_warn("WARNING: cache was used during dump: "
3212                             "corruption may be incorrectly reported\n");
3213                 }

3215                 kmv.kmv_size = kmv.kmv_cache.cache_buftag +
3216                     sizeof (kmem_buftag_t);
3217                 kmv.kmv_buf = mdb_alloc(kmv.kmv_size, UM_SLEEP | UM_GC);
3218                 kmv.kmv_corruption = 0;
3219                 kmv.kmv_flags = flags;

3221                 if ((kmv.kmv_cache.cache_flags & KMF_REDZONE)) {
3222                         check_alloc = 1;
3223                         if (kmv.kmv_cache.cache_flags & KMF_DEADBEEF)
3224                                 check_free = 1;
3225                 } else {
3226                         if (!(flags & DCMD_LOOP)) {
3227                                 mdb_warn("cache %p (%s) does not have "
3228                                     "redzone checking enabled\n", addr,
3229                                     kmv.kmv_cache.cache_name);
3230                         }
3231                         return (DCMD_ERR);
3232                 }
```

```
3234                if (!(flags & (DCMD_LOOP | DCMD_PIPE_OUT))) {
3221                if (flags & DCMD_LOOP) {
3222                        /*
3223                         * table mode, don't print out every corrupt buffer
3224                         */
3225                        kmv.kmv_besilent = 1;
3226                } else {
3235                        mdb_printf("Summary for cache '%s'\n",
3236                            kmv.kmv_cache.cache_name);
3237                        mdb_inc_indent(2);
3230                        kmv.kmv_besilent = 0;
3238                }

3240                if (check_alloc)
3241                        (void) mdb_pwalk("kmem", verify_alloc, &kmv, addr);
3242                if (check_free)
3243                        (void) mdb_pwalk("freemem", verify_free, &kmv, addr);

3245                if (!(flags & DCMD_PIPE_OUT)) {
3246                        if (flags & DCMD_LOOP) {
3247                                if (kmv.kmv_corruption == 0) {
3248                                        mdb_printf("%-*s %?p clean\n",
3249                                            KMEM_CACHE_NAMELEN,
3250                                            kmv.kmv_cache.cache_name, addr);
3251                                } else {
3252                                        mdb_printf("%-*s %?p %d corrupt "
3253                                            "buffer%s\n", KMEM_CACHE_NAMELEN,
3244                                char *s = "";   /* optional s in "buffer[s]" */
3245                                if (kmv.kmv_corruption > 1)
3246                                        s = "s";

3248                                mdb_printf("%-*s %?p %d corrupt buffer%s\n",
3249                                    KMEM_CACHE_NAMELEN,
3254                                        kmv.kmv_cache.cache_name, addr,
3255                                        kmv.kmv_corruption,
3256                                        kmv.kmv_corruption > 1 ? "s" : "");
3251                                    kmv.kmv_corruption, s);
3257                                }
3258                        } else {
3259                                /*
3260                                 * This is the more verbose mode, when the user
3261                                 * typed addr::kmem_verify.  If the cache was
3262                                 * clean, nothing will have yet been printed. So
3263                                 * say something.
3255                                 * This is the more verbose mode, when the user has
3256                                 * type addr::kmem_verify.  If the cache was clean,
3257                                 * nothing will have yet been printed. So say something.
3264                                 */
3265                                if (kmv.kmv_corruption == 0)
3266                                        mdb_printf("clean\n");

3268                                mdb_dec_indent(2);
3269                        }
3270                }
3271        } else {
3272                /*
3273                 * If the user didn't specify a cache to verify, we'll walk all
3274                 * kmem_cache's, specifying ourself as a callback for each...
3275                 * this is the equivalent of '::walk kmem_cache .::kmem_verify'
3276                 */

3278                if (!(flags & DCMD_PIPE_OUT)) {
3279                        uintptr_t dump_curr;
3280                        uintptr_t dump_end;

3282                        if (mdb_readvar(&dump_curr, "kmem_dump_curr") != -1 &&
```



```
3283                            mdb_readvar(&dump_end, "kmem_dump_end") != -1 &&
3284                            dump_curr == dump_end) {
3285                                mdb_warn("WARNING: exceeded kmem_dump_size; "
3286                                    "corruption may be incorrectly reported\n");
3287                        }

3289                        mdb_printf("%<u>%-*s %-?s %-20s%</b>\n",
3290                            KMEM_CACHE_NAMELEN, "Cache Name", "Addr",
3291                            "Cache Integrity");
3292                }

3270                mdb_printf("%<u>%-*s %-?s %-20s%</b>\n", KMEM_CACHE_NAMELEN,
3271                    "Cache Name", "Addr", "Cache Integrity");
3294                (void) (mdb_walk_dcmd("kmem_cache", "kmem_verify", 0, NULL));
3295        }

3297        return (DCMD_OK);
3298 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   80856 Sun Apr 22 20:26:17 2018
new/usr/src/uts/common/os/dumpsubr.c
9525 kmem_dump_size is a corrupting influence
**********************************************************
```

```
/*
 * 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) 1998, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2018 Joyent, Inc.
 * Copyright 2016 Joyent, Inc.
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/vm.h>
#include <sys/proc.h>
#include <sys/file.h>
#include <sys/conf.h>
#include <sys/kmem.h>
#include <sys/mem.h>
#include <sys/mman.h>
#include <sys/vnode.h>
#include <sys/errno.h>
#include <sys/memlist.h>
#include <sys/dumphdr.h>
#include <sys/dumpadm.h>
#include <sys/ksyms.h>
#include <sys/compress.h>
#include <sys/stream.h>
#include <sys/strsun.h>
#include <sys/cmn_err.h>
#include <sys/bitmap.h>
#include <sys/modctl.h>
#include <sys/utsname.h>
#include <sys/systeminfo.h>
#include <sys/vmem.h>
#include <sys/log.h>
#include <sys/var.h>
#include <sys/debug.h>
#include <sys/sunddi.h>
#include <fs/fs_subr.h>
#include <sys/fs/snode.h>
#include <sys/ontrap.h>
#include <sys/panic.h>
#include <sys/dkio.h>
```



```
#include <sys/vtoc.h>
#include <sys/errorq.h>
#include <sys/fm/util.h>
#include <sys/fs/zfs.h>

#include <vm/hat.h>
#include <vm/as.h>
#include <vm/page.h>
#include <vm/pvn.h>
#include <vm/seg.h>
#include <vm/seg_kmem.h>
#include <sys/clock_impl.h>
#include <sys/hold_page.h>

#include <bzip2/bzlib.h>

#define ONE_GIG (1024 * 1024 * 1024UL)

/*
 * Crash dump time is dominated by disk write time.  To reduce this,
 * the stronger compression method bzip2 is applied to reduce the dump
 * size and hence reduce I/O time.  However, bzip2 is much more
 * computationally expensive than the existing lzjb algorithm, so to
 * avoid increasing compression time, CPUs that are otherwise idle
 * during panic are employed to parallelize the compression task.
 * Many helper CPUs are needed to prevent bzip2 from being a
 * bottleneck, and on systems with too few CPUs, the lzjb algorithm is
 * parallelized instead. Lastly, I/O and compression are performed by
 * different CPUs, and are hence overlapped in time, unlike the older
 * serial code.
 *
 * Another important consideration is the speed of the dump
 * device. Faster disks need less CPUs in order to benefit from
 * parallel lzjb versus parallel bzip2. Therefore, the CPU count
 * threshold for switching from parallel lzjb to paralled bzip2 is
 * elevated for faster disks. The dump device speed is adduced from
 * the setting for dumpbuf.iosize, see dump_update_clevel.
 */

/*
 * exported vars
 */
kmutex_t        dump_lock;              /* lock for dump configuration */
dumphdr_t       *dumphdr;               /* dump header */
int             dump_conflags = DUMP_KERNEL; /* dump configuration flags */
vnode_t         *dumpvp;                /* dump device vnode pointer */
u_offset_t      dumpvp_size;            /* size of dump device, in bytes */
char            *dumppath;              /* pathname of dump device */
int             dump_timeout = 120;     /* timeout for dumping pages */
int             dump_timeleft;          /* portion of dump_timeout remaining */
int             dump_ioerr;             /* dump i/o error */
int             dump_check_used;        /* enable check for used pages */
char        *dump_stack_scratch; /* scratch area for saving stack summary */

/*
 * Tunables for dump compression and parallelism. These can be set via
 * /etc/system.
 *
 * dump_ncpu_low        number of helpers for parallel lzjb
 *      This is also the minimum configuration.
 *
 * dump_bzip2_level     bzip2 compression level: 1-9
 *      Higher numbers give greater compression, but take more memory
 *      and time. Memory used per helper is ~(dump_bzip2_level * 1MB).
 *
 * dump_plat_mincpu     the cross-over limit for using bzip2 (per platform):
```

```
 127  *      if dump_plat_mincpu == 0, then always do single threaded dump
 128  *      if ncpu >= dump_plat_mincpu then try to use bzip2
 129  *
 130  * dump_metrics_on      if set, metrics are collected in the kernel, passed
 131  *      to savecore via the dump file, and recorded by savecore in
 132  *      METRICS.txt.
 133  */
 134 uint_t dump_ncpu_low = 4;       /* minimum config for parallel lzjb */
 135 uint_t dump_bzip2_level = 1;    /* bzip2 level (1-9) */

 137 /* Use dump_plat_mincpu_default unless this variable is set by /etc/system */
 138 #define MINCPU_NOT_SET  ((uint_t)-1)
 139 uint_t dump_plat_mincpu = MINCPU_NOT_SET;

 141 /* tunables for pre-reserved heap */
 142 uint_t dump_kmem_permap = 1024;
 143 uint_t dump_kmem_pages = 0;
 141 uint_t dump_kmem_pages = 8;

 145 /* Define multiple buffers per helper to avoid stalling */
 146 #define NCBUF_PER_HELPER        2
 147 #define NCMAP_PER_HELPER        4

 149 /* minimum number of helpers configured */
 150 #define MINHELPERS      (dump_ncpu_low)
 151 #define MINCBUFS        (MINHELPERS * NCBUF_PER_HELPER)

 153 /*
 154  * Define constant parameters.
 155  *
 156  * CBUF_SIZE            size of an output buffer
 157  *
 158  * CBUF_MAPSIZE         size of virtual range for mapping pages
 159  *
 160  * CBUF_MAPNP           size of virtual range in pages
 161  *
 162  */
 163 #define DUMP_1KB        ((size_t)1 << 10)
 164 #define DUMP_1MB        ((size_t)1 << 20)
 165 #define CBUF_SIZE       ((size_t)1 << 17)
 166 #define CBUF_MAPSHIFT   (22)
 167 #define CBUF_MAPSIZE    ((size_t)1 << CBUF_MAPSHIFT)
 168 #define CBUF_MAPNP      ((size_t)1 << (CBUF_MAPSHIFT - PAGESHIFT))

 170 /*
 171  * Compression metrics are accumulated nano-second subtotals. The
 172  * results are normalized by the number of pages dumped. A report is
 173  * generated when dumpsys() completes and is saved in the dump image
 174  * after the trailing dump header.
 175  *
 176  * Metrics are always collected. Set the variable dump_metrics_on to
 177  * cause metrics to be saved in the crash file, where savecore will
 178  * save it in the file METRICS.txt.
 179  */
 180 #define PERPAGES \
 181         PERPAGE(bitmap) PERPAGE(map) PERPAGE(unmap) \
 182         PERPAGE(copy) PERPAGE(compress) \
 183         PERPAGE(write) \
 184         PERPAGE(inwait) PERPAGE(outwait)

 186 typedef struct perpage {
 187 #define PERPAGE(x) hrtime_t x;
 188         PERPAGES
 189 #undef PERPAGE
 190 } perpage_t;
```

*unchanged_portion_omitted_*


```
 492 /*
 493  * dump_update_clevel is called when dumpadm configures the dump device.
 494  *      Calculate number of helpers and buffers.
 495  *      Allocate the minimum configuration for now.
 496  *
 497  * When the dump file is configured we reserve a minimum amount of
 498  * memory for use at crash time. But we reserve VA for all the memory
 499  * we really want in order to do the fastest dump possible. The VA is
 500  * backed by pages not being dumped, according to the bitmap. If
 501  * there is insufficient spare memory, however, we fall back to the
 502  * minimum.
 503  *
 504  * Live dump (savecore -L) always uses the minimum config.
 505  *
 506  * clevel 0 is single threaded lzjb
 507  * clevel 1 is parallel lzjb
 508  * clevel 2 is parallel bzip2
 509  *
 510  * The ncpu threshold is selected with dump_plat_mincpu.
 511  * On OPL, set_platform_defaults() overrides the sun4u setting.
 512  * The actual values are defined via DUMP_PLAT_*_MINCPU macros.
 513  *
 514  * Architecture         Threshold       Algorithm
 515  * sun4u                <  51           parallel lzjb
 516  * sun4u                >= 51           parallel bzip2(*)
 517  * sun4u OPL            <  8            parallel lzjb
 518  * sun4u OPL            >= 8            parallel bzip2(*)
 519  * sun4v                <  128          parallel lzjb
 520  * sun4v                >= 128          parallel bzip2(*)
 521  * x86                  < 11            parallel lzjb
 522  * x86                  >= 11           parallel bzip2(*)
 523  * 32-bit               N/A             single-threaded lzjb
 524  *
 525  * (*) bzip2 is only chosen if there is sufficient available
 526  * memory for buffers at dump time. See dumpsys_get_maxmem().
 527  *
 528  * Faster dump devices have larger I/O buffers. The threshold value is
 529  * increased according to the size of the dump I/O buffer, because
 530  * parallel lzjb performs better with faster disks. For buffers >= 1MB
 531  * the threshold is 3X; for buffers >= 256K threshold is 2X.
 532  *
 533  * For parallel dumps, the number of helpers is ncpu-1. The CPU
 534  * running panic runs the main task. For single-threaded dumps, the
 535  * panic CPU does lzjb compression (it is tagged as MAINHELPER.)
 536  *
 537  * Need multiple buffers per helper so that they do not block waiting
 538  * for the main task.
 539  *                                 parallel        single-threaded
 540  * Number of output buffers:    nhelper*2               1
 541  * Number of mapping buffers:   nhelper*4               1
 542  *
 543  */
 544 static void
 545 dump_update_clevel()
 546 {
 547         int tag;
 548         size_t bz2size;
 549         helper_t *hp, *hpend;
 550         cbuf_t *cp, *cpend;
 551         dumpcfg_t *old = &dumpcfg;
 552         dumpcfg_t newcfg = *old;
 553         dumpcfg_t *new = &newcfg;

 555         ASSERT(MUTEX_HELD(&dump_lock));
```

```
557         /*
558          * Free the previously allocated bufs and VM.
559          */
560         if (old->helper != NULL) {

562                 /* helpers */
563                 hpend = &old->helper[old->nhelper];
564                 for (hp = old->helper; hp != hpend; hp++) {
565                         if (hp->lzbuf != NULL)
566                                 kmem_free(hp->lzbuf, PAGESIZE);
567                         if (hp->page != NULL)
568                                 kmem_free(hp->page, PAGESIZE);
569                 }
570                 kmem_free(old->helper, old->nhelper * sizeof (helper_t));

572                 /* VM space for mapping pages */
573                 cpend = &old->cmap[old->ncmap];
574                 for (cp = old->cmap; cp != cpend; cp++)
575                         vmem_xfree(heap_arena, cp->buf, CBUF_MAPSIZE);
576                 kmem_free(old->cmap, old->ncmap * sizeof (cbuf_t));

578                 /* output bufs */
579                 cpend = &old->cbuf[old->ncbuf];
580                 for (cp = old->cbuf; cp != cpend; cp++)
581                         if (cp->buf != NULL)
582                                 kmem_free(cp->buf, cp->size);
583                 kmem_free(old->cbuf, old->ncbuf * sizeof (cbuf_t));

585                 /* reserved VM for dumpsys_get_maxmem */
586                 if (old->maxvmsize > 0)
587                         vmem_xfree(heap_arena, old->maxvm, old->maxvmsize);
588         }

590         /*
591          * Allocate memory and VM.
592          * One CPU runs dumpsys, the rest are helpers.
593          */
594         new->nhelper = ncpus - 1;
595         if (new->nhelper < 1)
596                 new->nhelper = 1;

598         if (new->nhelper > DUMP_MAX_NHELPER)
599                 new->nhelper = DUMP_MAX_NHELPER;

601         /* use platform default, unless /etc/system overrides */
602         if (dump_plat_mincpu == MINCPU_NOT_SET)
603                 dump_plat_mincpu = dump_plat_mincpu_default;

605         /* increase threshold for faster disks */
606         new->threshold = dump_plat_mincpu;
607         if (dumpbuf.iosize >= DUMP_1MB)
608                 new->threshold *= 3;
609         else if (dumpbuf.iosize >= (256 * DUMP_1KB))
610                 new->threshold *= 2;

612         /* figure compression level based upon the computed threshold. */
613         if (dump_plat_mincpu == 0 || new->nhelper < 2) {
614                 new->clevel = 0;
615                 new->nhelper = 1;
616         } else if ((new->nhelper + 1) >= new->threshold) {
617                 new->clevel = DUMP_CLEVEL_BZIP2;
618         } else {
619                 new->clevel = DUMP_CLEVEL_LZJB;
620         }

622         if (new->clevel == 0) {
```



```
623                 new->ncbuf = 1;
624                 new->ncmap = 1;
625         } else {
626                 new->ncbuf = NCBUF_PER_HELPER * new->nhelper;
627                 new->ncmap = NCMAP_PER_HELPER * new->nhelper;
628         }

630         /*
631          * Allocate new data structures and buffers for MINHELPERS,
632          * and also figure the max desired size.
633          */
634         bz2size = BZ2_bzCompressInitSize(dump_bzip2_level);
635         new->maxsize = 0;
636         new->maxvmsize = 0;
637         new->maxvm = NULL;
638         tag = 1;
639         new->helper = kmem_zalloc(new->nhelper * sizeof (helper_t), KM_SLEEP);
640         hpend = &new->helper[new->nhelper];
641         for (hp = new->helper; hp != hpend; hp++) {
642                 hp->tag = tag++;
643                 if (hp < &new->helper[MINHELPERS]) {
644                         hp->lzbuf = kmem_alloc(PAGESIZE, KM_SLEEP);
645                         hp->page = kmem_alloc(PAGESIZE, KM_SLEEP);
646                 } else if (new->clevel < DUMP_CLEVEL_BZIP2) {
647                         new->maxsize += 2 * PAGESIZE;
648                 } else {
649                         new->maxsize += PAGESIZE;
650                 }
651                 if (new->clevel >= DUMP_CLEVEL_BZIP2)
652                         new->maxsize += bz2size;
653         }

655         new->cbuf = kmem_zalloc(new->ncbuf * sizeof (cbuf_t), KM_SLEEP);
656         cpend = &new->cbuf[new->ncbuf];
657         for (cp = new->cbuf; cp != cpend; cp++) {
658                 cp->state = CBUF_FREEBUF;
659                 cp->size = CBUF_SIZE;
660                 if (cp < &new->cbuf[MINCBUFS])
661                         cp->buf = kmem_alloc(cp->size, KM_SLEEP);
662                 else
663                         new->maxsize += cp->size;
664         }

666         new->cmap = kmem_zalloc(new->ncmap * sizeof (cbuf_t), KM_SLEEP);
667         cpend = &new->cmap[new->ncmap];
668         for (cp = new->cmap; cp != cpend; cp++) {
669                 cp->state = CBUF_FREEMAP;
670                 cp->size = CBUF_MAPSIZE;
671                 cp->buf = vmem_xalloc(heap_arena, CBUF_MAPSIZE, CBUF_MAPSIZE,
672                     0, 0, NULL, NULL, VM_SLEEP);
673         }

675         /* reserve VA to be backed with spare pages at crash time */
676         if (new->maxsize > 0) {
677                 new->maxsize = P2ROUNDUP(new->maxsize, PAGESIZE);
678                 new->maxvmsize = P2ROUNDUP(new->maxsize, CBUF_MAPSIZE);
679                 new->maxvm = vmem_xalloc(heap_arena, new->maxvmsize,
680                     CBUF_MAPSIZE, 0, 0, NULL, NULL, VM_SLEEP);
681         }

683         /*
684          * Reserve memory for kmem allocation calls made during crash dump.  The
685          * hat layer allocates memory for each mapping created, and the I/O path
686          * allocates buffers and data structs.
687          *
688          * On larger systems, we easily exceed the lower amount, so we need some
```

```
 689          * more space; the cut-over point is relatively arbitrary.  If we run
 690          * out, the only impact is that kmem state in the dump becomes
 691          * inconsistent.
 682          * Reserve memory for kmem allocation calls made during crash
 683          * dump.  The hat layer allocates memory for each mapping
 684          * created, and the I/O path allocates buffers and data structs.
 685          * Add a few pages for safety.
 692          */

 694         if (dump_kmem_pages == 0) {
 695                 if (physmem > (16 * ONE_GIG) / PAGESIZE)
 696                         dump_kmem_pages = 20;
 697                 else
 698                         dump_kmem_pages = 8;
 699         }

 701         kmem_dump_init((new->ncmap * dump_kmem_permap) +
 702             (dump_kmem_pages * PAGESIZE));

 704         /* set new config pointers */
 705         *old = *new;
 706 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
  176134 Sun Apr 22 20:26:18 2018
new/usr/src/uts/common/os/kmem.c
9525 kmem_dump_size is a corrupting influence
**********************************************************
______unchanged_portion_omitted_

2210 #define KMEM_DUMPCTL(cp, buf)   \
2211         ((kmem_dumpctl_t *)P2ROUNDUP((uintptr_t)(buf) + (cp)->cache_bufsize, \
2212             sizeof (void *)))

2214 /* Keep some simple stats. */
2215 #define KMEM_DUMP_LOGS  (100)

2217 typedef struct kmem_dump_log {
2218         kmem_cache_t    *kdl_cache;
2219         uint_t          kdl_allocs;             /* # of dump allocations */
2220         uint_t          kdl_frees;              /* # of dump frees */
2221         uint_t          kdl_alloc_fails;        /* # of allocation failures */
2222         uint_t          kdl_free_nondump;       /* # of non-dump frees */
2223         uint_t          kdl_unsafe;             /* cache was used, but unsafe */
2224 } kmem_dump_log_t;

2226 static kmem_dump_log_t *kmem_dump_log;
2227 static int kmem_dump_log_idx;

2229 #define KDI_LOG(cp, stat) {                                             \
2230         kmem_dump_log_t *kdl;                                          \
2231         if ((kdl = (kmem_dump_log_t *)((cp)->cache_dumplog)) != NULL) { \
2232                 kdl->stat++;                                           \
2233         } else if (kmem_dump_log_idx < KMEM_DUMP_LOGS) {               \
2234                 kdl = &kmem_dump_log[kmem_dump_log_idx++];             \
2235                 kdl->stat++;                                           \
2236                 kdl->kdl_cache = (cp);                                 \
2237                 (cp)->cache_dumplog = kdl;                             \
2238         }                                                              \
2239 }

2214 /* set non zero for full report */
2215 uint_t kmem_dump_verbose = 0;

2217 /* stats for overize heap */
2218 uint_t kmem_dump_oversize_allocs = 0;
2219 uint_t kmem_dump_oversize_max = 0;

2221 static void
2222 kmem_dumppr(char **pp, char *e, const char *format, ...)
2223 {
2224         char *p = *pp;

2226         if (p < e) {
2227                 int n;
2228                 va_list ap;

2230                 va_start(ap, format);
2231                 n = vsnprintf(p, e - p, format, ap);
2232                 va_end(ap);
2233                 *pp = p + n;
2234         }
2235 }

2237 /*
2238  * Called when dumpadm(1M) configures dump parameters.
2239  */
2240 void
2241 kmem_dump_init(size_t size)
```



```
2242 {
2243         /* Our caller ensures size is always set. */
2244         ASSERT3U(size, >, 0);

2246         if (kmem_dump_start != NULL)
2247                 kmem_free(kmem_dump_start, kmem_dump_size);

2273         if (kmem_dump_log == NULL)
2274                 kmem_dump_log = (kmem_dump_log_t *)kmem_zalloc(KMEM_DUMP_LOGS *
2275                     sizeof (kmem_dump_log_t), KM_SLEEP);

2249         kmem_dump_start = kmem_alloc(size, KM_SLEEP);

2279         if (kmem_dump_start != NULL) {
2250         kmem_dump_size = size;
2251         kmem_dump_curr = kmem_dump_start;
2252         kmem_dump_end = (void *)((char *)kmem_dump_start + size);
2253         copy_pattern(KMEM_UNINITIALIZED_PATTERN, kmem_dump_start, size);
2284         } else {
2285                 kmem_dump_size = 0;
2286                 kmem_dump_curr = NULL;
2287                 kmem_dump_end = NULL;
2288         }
2254 }

2256 /*
2257  * Set flag for each kmem_cache_t if is safe to use alternate dump
2258  * memory. Called just before panic crash dump starts. Set the flag
2259  * for the calling CPU.
2260  */
2261 void
2262 kmem_dump_begin(void)
2263 {
2299         ASSERT(panicstr != NULL);
2300         if (kmem_dump_start != NULL) {
2264         kmem_cache_t *cp;

2266         ASSERT(panicstr != NULL);

2268         for (cp = list_head(&kmem_caches); cp != NULL;
2269             cp = list_next(&kmem_caches, cp)) {
2270                 kmem_cpu_cache_t *ccp = KMEM_CPU_CACHE(cp);

2272                 if (cp->cache_arena->vm_cflags & VMC_DUMPSAFE) {
2273                         cp->cache_flags |= KMF_DUMPDIVERT;
2274                         ccp->cc_flags |= KMF_DUMPDIVERT;
2275                         ccp->cc_dump_rounds = ccp->cc_rounds;
2276                         ccp->cc_dump_prounds = ccp->cc_prounds;
2277                         ccp->cc_rounds = ccp->cc_prounds = -1;
2278                 } else {
2279                         cp->cache_flags |= KMF_DUMPUNSAFE;
2280                         ccp->cc_flags |= KMF_DUMPUNSAFE;
2281                 }
2282         }
2318         }
2283 }

2285 /*
2286  * finished dump intercept
2287  * print any warnings on the console
2288  * return verbose information to dumpsys() in the given buffer
2289  */
2290 size_t
2291 kmem_dump_finish(char *buf, size_t size)
2292 {
2329         int kdi_idx;
```

```
2330         int kdi_end = kmem_dump_log_idx;
2293         int percent = 0;
2332         int header = 0;
2333         int warn = 0;
2294         size_t used;
2335         kmem_cache_t *cp;
2336         kmem_dump_log_t *kdl;
2295         char *e = buf + size;
2296         char *p = buf;

2298         if (kmem_dump_curr == kmem_dump_end) {
2299                 cmn_err(CE_WARN, "exceeded kmem_dump space of %lu "
2300                     "bytes: kmem state in dump may be inconsistent",
2301                     kmem_dump_size);
2302         }

2304         if (kmem_dump_verbose == 0)
2340         if (kmem_dump_size == 0 || kmem_dump_verbose == 0)
2305                 return (0);

2307         used = (char *)kmem_dump_curr - (char *)kmem_dump_start;
2308         percent = (used * 100) / kmem_dump_size;

2310         kmem_dumppr(&p, e, "%% heap used,%d\n", percent);
2311         kmem_dumppr(&p, e, "used bytes,%ld\n", used);
2312         kmem_dumppr(&p, e, "heap size,%ld\n", kmem_dump_size);
2313         kmem_dumppr(&p, e, "Oversize allocs,%d\n",
2314             kmem_dump_oversize_allocs);
2315         kmem_dumppr(&p, e, "Oversize max size,%ld\n",
2316             kmem_dump_oversize_max);

2354         for (kdi_idx = 0; kdi_idx < kdi_end; kdi_idx++) {
2355                 kdl = &kmem_dump_log[kdi_idx];
2356                 cp = kdl->kdl_cache;
2357                 if (cp == NULL)
2358                         break;
2359                 if (kdl->kdl_alloc_fails)
2360                         ++warn;
2361                 if (header == 0) {
2362                         kmem_dumppr(&p, e,
2363                             "Cache Name,Allocs,Frees,Alloc Fails,"
2364                             "Nondump Frees,Unsafe Allocs/Frees\n");
2365                         header = 1;
2366                 }
2367                 kmem_dumppr(&p, e, "%s,%d,%d,%d,%d,%d\n",
2368                     cp->cache_name, kdl->kdl_allocs, kdl->kdl_frees,
2369                     kdl->kdl_alloc_fails, kdl->kdl_free_nondump,
2370                     kdl->kdl_unsafe);
2371         }

2318         /* return buffer size used */
2319         if (p < e)
2320                 bzero(p, e - p);
2321         return (p - buf);
2322 }

2324 /*
2325  * Allocate a constructed object from alternate dump memory.
2326  */
2327 void *
2328 kmem_cache_alloc_dump(kmem_cache_t *cp, int kmflag)
2329 {
2330         void *buf;
2331         void *curr;
2332         char *bufend;
```



```
2334         /* return a constructed object */
2335         if ((buf = cp->cache_dump.kd_freelist) != NULL) {
2336                 cp->cache_dump.kd_freelist = KMEM_DUMPCTL(cp, buf)->kdc_next;
2390         if ((buf = cp->cache_dumpfreelist) != NULL) {
2391                 cp->cache_dumpfreelist = KMEM_DUMPCTL(cp, buf)->kdc_next;
2392                 KDI_LOG(cp, kdl_allocs);
2337                 return (buf);
2338         }

2340         /* create a new constructed object */
2341         curr = kmem_dump_curr;
2342         buf = (void *)P2ROUNDUP((uintptr_t)curr, cp->cache_align);
2343         bufend = (char *)KMEM_DUMPCTL(cp, buf) + sizeof (kmem_dumpctl_t);

2345         /* hat layer objects cannot cross a page boundary */
2346         if (cp->cache_align < PAGESIZE) {
2347                 char *page = (char *)P2ROUNDUP((uintptr_t)buf, PAGESIZE);
2348                 if (bufend > page) {
2349                         bufend += page - (char *)buf;
2350                         buf = (void *)page;
2351                 }
2352         }

2354         /* fall back to normal alloc if reserved area is used up */
2355         if (bufend > (char *)kmem_dump_end) {
2356                 kmem_dump_curr = kmem_dump_end;
2357                 cp->cache_dump.kd_alloc_fails++;
2413                 KDI_LOG(cp, kdl_alloc_fails);
2358                 return (NULL);
2359         }

2361         /*
2362          * Must advance curr pointer before calling a constructor that
2363          * may also allocate memory.
2364          */
2365         kmem_dump_curr = bufend;

2367         /* run constructor */
2368         if (cp->cache_constructor != NULL &&
2369             cp->cache_constructor(buf, cp->cache_private, kmflag)
2370             != 0) {
2371 #ifdef DEBUG
2372                 printf("name='%s' cache=0x%p: kmem cache constructor failed\n",
2373                     cp->cache_name, (void *)cp);
2374 #endif
2375                 /* reset curr pointer iff no allocs were done */
2376                 if (kmem_dump_curr == bufend)
2377                         kmem_dump_curr = curr;

2379                 cp->cache_dump.kd_alloc_fails++;
2380                 /* fall back to normal alloc if the constructor fails */
2436                 KDI_LOG(cp, kdl_alloc_fails);
2381                 return (NULL);
2382         }

2440         KDI_LOG(cp, kdl_allocs);
2384         return (buf);
2385 }

2387 /*
2388  * Free a constructed object in alternate dump memory.
2389  */
2390 int
2391 kmem_cache_free_dump(kmem_cache_t *cp, void *buf)
2392 {
2393         /* save constructed buffers for next time */
```

```
2394         if ((char *)buf >= (char *)kmem_dump_start &&
2395             (char *)buf < (char *)kmem_dump_end) {
2396                 KMEM_DUMPCTL(cp, buf)->kdc_next = cp->cache_dump.kd_freelist;
2397                 cp->cache_dump.kd_freelist = buf;
2453                 KMEM_DUMPCTL(cp, buf)->kdc_next = cp->cache_dumpfreelist;
2454                 cp->cache_dumpfreelist = buf;
2455                 KDI_LOG(cp, kdl_frees);
2398                 return (0);
2399         }

2459         /* count all non-dump buf frees */
2460         KDI_LOG(cp, kdl_free_nondump);

2401         /* just drop buffers that were allocated before dump started */
2402         if (kmem_dump_curr < kmem_dump_end)
2403                 return (0);

2405         /* fall back to normal free if reserved area is used up */
2406         return (1);
2407 }

2409 /*
2410  * Allocate a constructed object from cache cp.
2411  */
2412 void *
2413 kmem_cache_alloc(kmem_cache_t *cp, int kmflag)
2414 {
2415         kmem_cpu_cache_t *ccp = KMEM_CPU_CACHE(cp);
2416         kmem_magazine_t *fmp;
2417         void *buf;

2419         mutex_enter(&ccp->cc_lock);
2420         for (;;) {
2421                 /*
2422                  * If there's an object available in the current CPU's
2423                  * loaded magazine, just take it and return.
2424                  */
2425                 if (ccp->cc_rounds > 0) {
2426                         buf = ccp->cc_loaded->mag_round[--ccp->cc_rounds];
2427                         ccp->cc_alloc++;
2428                         mutex_exit(&ccp->cc_lock);
2429                         if (ccp->cc_flags & (KMF_BUFTAG | KMF_DUMPUNSAFE)) {
2430                                 if (ccp->cc_flags & KMF_DUMPUNSAFE) {
2431                                         ASSERT(!(ccp->cc_flags &
2432                                             KMF_DUMPDIVERT));
2433                                         cp->cache_dump.kd_unsafe++;
2494                                         KDI_LOG(cp, kdl_unsafe);
2434                                 }
2435                                 if ((ccp->cc_flags & KMF_BUFTAG) &&
2436                                     kmem_cache_alloc_debug(cp, buf, kmflag, 0,
2437                                     caller()) != 0) {
2438                                         if (kmflag & KM_NOSLEEP)
2439                                                 return (NULL);
2440                                         mutex_enter(&ccp->cc_lock);
2441                                         continue;
2442                                 }
2443                         }
2444                         return (buf);
2445                 }

2447                 /*
2448                  * The loaded magazine is empty.  If the previously loaded
2449                  * magazine was full, exchange them and try again.
2450                  */
2451                 if (ccp->cc_prounds > 0) {
2452                         kmem_cpu_reload(ccp, ccp->cc_ploaded, ccp->cc_prounds);
```



```
2453                         continue;
2454                 }

2456                 /*
2457                  * Return an alternate buffer at dump time to preserve
2458                  * the heap.
2459                  */
2460                 if (ccp->cc_flags & (KMF_DUMPDIVERT | KMF_DUMPUNSAFE)) {
2461                         if (ccp->cc_flags & KMF_DUMPUNSAFE) {
2462                                 ASSERT(!(ccp->cc_flags & KMF_DUMPDIVERT));
2463                                 /* log it so that we can warn about it */
2464                                 cp->cache_dump.kd_unsafe++;
2525                                 KDI_LOG(cp, kdl_unsafe);
2465                         } else {
2466                                 if ((buf = kmem_cache_alloc_dump(cp, kmflag)) !=
2467                                     NULL) {
2468                                         mutex_exit(&ccp->cc_lock);
2469                                         return (buf);
2470                                 }
2471                                 break;          /* fall back to slab layer */
2472                         }
2473                 }

2475                 /*
2476                  * If the magazine layer is disabled, break out now.
2477                  */
2478                 if (ccp->cc_magsize == 0)
2479                         break;

2481                 /*
2482                  * Try to get a full magazine from the depot.
2483                  */
2484                 fmp = kmem_depot_alloc(cp, &cp->cache_full);
2485                 if (fmp != NULL) {
2486                         if (ccp->cc_ploaded != NULL)
2487                                 kmem_depot_free(cp, &cp->cache_empty,
2488                                     ccp->cc_ploaded);
2489                         kmem_cpu_reload(ccp, fmp, ccp->cc_magsize);
2490                         continue;
2491                 }

2493                 /*
2494                  * There are no full magazines in the depot,
2495                  * so fall through to the slab layer.
2496                  */
2497                 break;
2498         }
2499         mutex_exit(&ccp->cc_lock);

2501         /*
2502          * We couldn't allocate a constructed object from the magazine layer,
2503          * so get a raw buffer from the slab layer and apply its constructor.
2504          */
2505         buf = kmem_slab_alloc(cp, kmflag);

2507         if (buf == NULL)
2508                 return (NULL);

2510         if (cp->cache_flags & KMF_BUFTAG) {
2511                 /*
2512                  * Make kmem_cache_alloc_debug() apply the constructor for us.
2513                  */
2514                 int rc = kmem_cache_alloc_debug(cp, buf, kmflag, 1, caller());
2515                 if (rc != 0) {
2516                         if (kmflag & KM_NOSLEEP)
2517                                 return (NULL);
```

```
2518                        /*
2519                         * kmem_cache_alloc_debug() detected corruption
2520                         * but didn't panic (kmem_panic <= 0). We should not be
2521                         * here because the constructor failed (indicated by a
2522                         * return code of 1). Try again.
2523                         */
2524                        ASSERT(rc == -1);
2525                        return (kmem_cache_alloc(cp, kmflag));
2526                }
2527                return (buf);
2528        }

2530        if (cp->cache_constructor != NULL &&
2531            cp->cache_constructor(buf, cp->cache_private, kmflag) != 0) {
2532                atomic_inc_64(&cp->cache_alloc_fail);
2533                kmem_slab_free(cp, buf);
2534                return (NULL);
2535        }

2537        return (buf);
2538 }
```

*unchanged_portion_omitted_*

```
2639 /*
2640  * Free a constructed object to cache cp.
2641  */
2642 void
2643 kmem_cache_free(kmem_cache_t *cp, void *buf)
2644 {
2645        kmem_cpu_cache_t *ccp = KMEM_CPU_CACHE(cp);

2647        /*
2648         * The client must not free either of the buffers passed to the move
2649         * callback function.
2650         */
2651        ASSERT(cp->cache_defrag == NULL ||
2652            cp->cache_defrag->kmd_thread != curthread ||
2653            (buf != cp->cache_defrag->kmd_from_buf &&
2654            buf != cp->cache_defrag->kmd_to_buf));

2656        if (ccp->cc_flags & (KMF_BUFTAG | KMF_DUMPDIVERT | KMF_DUMPUNSAFE)) {
2657                if (ccp->cc_flags & KMF_DUMPUNSAFE) {
2658                        ASSERT(!(ccp->cc_flags & KMF_DUMPDIVERT));
2659                        /* log it so that we can warn about it */
2660                        cp->cache_dump.kd_unsafe++;
2721                        KDI_LOG(cp, kdl_unsafe);
2661                } else if (KMEM_DUMPCC(ccp) && !kmem_cache_free_dump(cp, buf)) {
2662                        return;
2663                }
2664                if (ccp->cc_flags & KMF_BUFTAG) {
2665                        if (kmem_cache_free_debug(cp, buf, caller()) == -1)
2666                                return;
2667                }
2668        }

2670        mutex_enter(&ccp->cc_lock);
2671        /*
2672         * Any changes to this logic should be reflected in kmem_slab_prefill()
2673         */
2674        for (;;) {
2675                /*
2676                 * If there's a slot available in the current CPU's
2677                 * loaded magazine, just put the object there and return.
2678                 */
2679                if ((uint_t)ccp->cc_rounds < ccp->cc_magsize) {
2680                        ccp->cc_loaded->mag_round[ccp->cc_rounds++] = buf;
```



```
2681                        ccp->cc_free++;
2682                        mutex_exit(&ccp->cc_lock);
2683                        return;
2684                }

2686                /*
2687                 * The loaded magazine is full.  If the previously loaded
2688                 * magazine was empty, exchange them and try again.
2689                 */
2690                if (ccp->cc_prounds == 0) {
2691                        kmem_cpu_reload(ccp, ccp->cc_ploaded, ccp->cc_prounds);
2692                        continue;
2693                }

2695                /*
2696                 * If the magazine layer is disabled, break out now.
2697                 */
2698                if (ccp->cc_magsize == 0)
2699                        break;

2701                if (!kmem_cpucache_magazine_alloc(ccp, cp)) {
2702                        /*
2703                         * We couldn't free our constructed object to the
2704                         * magazine layer, so apply its destructor and free it
2705                         * to the slab layer.
2706                         */
2707                        break;
2708                }
2709        }
2710        mutex_exit(&ccp->cc_lock);
2711        kmem_slab_free_constructed(cp, buf, B_TRUE);
2712 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   16219 Sun Apr 22 20:26:18 2018
new/usr/src/uts/common/sys/kmem_impl.h
9525 kmem_dump_size is a corrupting influence
**********************************************************
```

```
   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  */

  22 /*
  23  * Copyright (c) 1994, 2010, Oracle and/or its affiliates. All rights reserved.
  24  * Copyright 2018 Joyent, Inc.
  25  */

  27 #ifndef _SYS_KMEM_IMPL_H
  28 #define _SYS_KMEM_IMPL_H

  30 #include <sys/kmem.h>
  31 #include <sys/vmem.h>
  32 #include <sys/thread.h>
  33 #include <sys/t_lock.h>
  34 #include <sys/time.h>
  35 #include <sys/kstat.h>
  36 #include <sys/cpuvar.h>
  37 #include <sys/systm.h>
  38 #include <vm/page.h>
  39 #include <sys/avl.h>
  40 #include <sys/list.h>

  42 #ifdef  __cplusplus
  43 extern "C" {
  44 #endif

  46 /*
  47  * kernel memory allocator: implementation-private data structures
  48  *
  49  * Lock order:
  50  * 1. cache_lock
  51  * 2. cc_lock in order by CPU ID
  52  * 3. cache_depot_lock
  53  *
  54  * Do not call kmem_cache_alloc() or taskq_dispatch() while holding any of the
  55  * above locks.
  56  */

  58 #define KMF_AUDIT       0x00000001      /* transaction auditing */
  59 #define KMF_DEADBEEF    0x00000002      /* deadbeef checking */
  60 #define KMF_REDZONE     0x00000004      /* redzone checking */
  61 #define KMF_CONTENTS    0x00000008      /* freed-buffer content logging */
```



```
  62 #define KMF_STICKY      0x00000010      /* if set, override /etc/system */
  63 #define KMF_NOMAGAZINE  0x00000020      /* disable per-cpu magazines */
  64 #define KMF_FIREWALL    0x00000040      /* put all bufs before unmapped pages */
  65 #define KMF_LITE        0x00000100      /* lightweight debugging */

  67 #define KMF_HASH        0x00000200      /* cache has hash table */
  68 #define KMF_RANDOMIZE   0x00000400      /* randomize other kmem_flags */

  70 #define KMF_DUMPDIVERT  0x00001000      /* use alternate memory at dump time */
  71 #define KMF_DUMPUNSAFE  0x00002000      /* flag caches used at dump time */
  72 #define KMF_PREFILL     0x00004000      /* Prefill the slab when created. */

  74 #define KMF_BUFTAG      (KMF_DEADBEEF | KMF_REDZONE)
  75 #define KMF_TOUCH       (KMF_BUFTAG | KMF_LITE | KMF_CONTENTS)
  76 #define KMF_RANDOM      (KMF_TOUCH | KMF_AUDIT | KMF_NOMAGAZINE)
  77 #define KMF_DEBUG       (KMF_RANDOM | KMF_FIREWALL)

  79 #define KMEM_STACK_DEPTH        15

  81 #define KMEM_FREE_PATTERN               0xdeadbeefdeadbeefULL
  82 #define KMEM_UNINITIALIZED_PATTERN      0xbaddcafebaddcafeULL
  83 #define KMEM_REDZONE_PATTERN            0xfeedfacefeedfaceULL
  84 #define KMEM_REDZONE_BYTE               0xbb

  86 /*
  87  * Redzone size encodings for kmem_alloc() / kmem_free().  We encode the
  88  * allocation size, rather than storing it directly, so that kmem_free()
  89  * can distinguish frees of the wrong size from redzone violations.
  90  *
  91  * A size of zero is never valid.
  92  */
  93 #define KMEM_SIZE_ENCODE(x)     (251 * (x) + 1)
  94 #define KMEM_SIZE_DECODE(x)     ((x) / 251)
  95 #define KMEM_SIZE_VALID(x)      ((x) % 251 == 1 && (x) != 1)


  98 #define KMEM_ALIGN              8       /* min guaranteed alignment */
  99 #define KMEM_ALIGN_SHIFT        3       /* log2(KMEM_ALIGN) */
 100 #define KMEM_VOID_FRACTION      8       /* never waste more than 1/8 of slab */

 102 #define KMEM_SLAB_IS_PARTIAL(sp)                \
 103         ((sp)->slab_refcnt > 0 && (sp)->slab_refcnt < (sp)->slab_chunks)
 104 #define KMEM_SLAB_IS_ALL_USED(sp)               \
 105         ((sp)->slab_refcnt == (sp)->slab_chunks)

 107 /*
 108  * The bufctl (buffer control) structure keeps some minimal information
 109  * about each buffer: its address, its slab, and its current linkage,
 110  * which is either on the slab's freelist (if the buffer is free), or
 111  * on the cache's buf-to-bufctl hash table (if the buffer is allocated).
 112  * In the case of non-hashed, or "raw", caches (the common case), only
 113  * the freelist linkage is necessary: the buffer address is at a fixed
 114  * offset from the bufctl address, and the slab is at the end of the page.
 115  *
 116  * NOTE: bc_next must be the first field; raw buffers have linkage only.
 117  */
 118 typedef struct kmem_bufctl {
 119         struct kmem_bufctl      *bc_next;       /* next bufctl struct */
 120         void                    *bc_addr;       /* address of buffer */
 121         struct kmem_slab        *bc_slab;       /* controlling slab */
 122 } kmem_bufctl_t;
_____unchanged_portion_omitted_

 332 typedef struct kmem_dump {
 333         void            *kd_freelist;           /* heap during crash dump */
 334         uint_t          kd_alloc_fails;         /* # of allocation failures */
```

```
335         uint_t          kd_unsafe;              /* cache was used, but unsafe */
336 } kmem_dump_t;

338 #define KMEM_CACHE_NAMELEN      31

340 struct kmem_cache {
341         /*
342          * Statistics
343          */
344         uint64_t        cache_slab_create;      /* slab creates */
345         uint64_t        cache_slab_destroy;     /* slab destroys */
346         uint64_t        cache_slab_alloc;       /* slab layer allocations */
347         uint64_t        cache_slab_free;        /* slab layer frees */
348         uint64_t        cache_alloc_fail;       /* total failed allocations */
349         uint64_t        cache_buftotal;         /* total buffers */
350         uint64_t        cache_bufmax;           /* max buffers ever */
351         uint64_t        cache_bufslab;          /* buffers free in slab layer */
352         uint64_t        cache_reap;             /* cache reaps */
353         uint64_t        cache_rescale;          /* hash table rescales */
354         uint64_t        cache_lookup_depth;     /* hash lookup depth */
355         uint64_t        cache_depot_contention; /* mutex contention count */
356         uint64_t        cache_depot_contention_prev; /* previous snapshot */

358         /*
359          * Cache properties
360          */
361         char            cache_name[KMEM_CACHE_NAMELEN + 1];
362         size_t          cache_bufsize;          /* object size */
363         size_t          cache_align;            /* object alignment */
364         int             (*cache_constructor)(void *, void *, int);
365         void            (*cache_destructor)(void *, void *);
366         void            (*cache_reclaim)(void *);
367         kmem_cbrc_t     (*cache_move)(void *, void *, size_t, void *);
368         void            *cache_private;         /* opaque arg to callbacks */
369         vmem_t          *cache_arena;           /* vmem source for slabs */
370         int             cache_cflags;           /* cache creation flags */
371         int             cache_flags;            /* various cache state info */
372         uint32_t        cache_mtbf;             /* induced alloc failure rate */
373         uint32_t        cache_pad1;             /* compiler padding */
374         kstat_t         *cache_kstat;           /* exported statistics */
375         list_node_t     cache_link;             /* cache linkage */

377         /*
378          * Slab layer
379          */
380         kmutex_t        cache_lock;             /* protects slab layer */
381         size_t          cache_chunksize;        /* buf + alignment [+ debug] */
382         size_t          cache_slabsize;         /* size of a slab */
383         size_t          cache_maxchunks;        /* max buffers per slab */
384         size_t          cache_bufctl;           /* buf-to-bufctl distance */
385         size_t          cache_buftag;           /* buf-to-buftag distance */
386         size_t          cache_verify;           /* bytes to verify */
387         size_t          cache_contents;         /* bytes of saved content */
388         size_t          cache_color;            /* next slab color */
389         size_t          cache_mincolor;         /* maximum slab color */
390         size_t          cache_maxcolor;         /* maximum slab color */
391         size_t          cache_hash_shift;       /* get to interesting bits */
392         size_t          cache_hash_mask;        /* hash table mask */
393         list_t          cache_complete_slabs;   /* completely allocated slabs */
394         size_t          cache_complete_slab_count;
395         avl_tree_t      cache_partial_slabs;    /* partial slab freelist */
396         size_t          cache_partial_binshift; /* for AVL sort bins */
397         kmem_cache_t    *cache_bufctl_cache;    /* source of bufctls */
398         kmem_bufctl_t   **cache_hash_table;     /* hash table base */
399         kmem_defrag_t   *cache_defrag;          /* slab consolidator fields */
```



```
401         /*
402          * Depot layer
403          */
404         kmutex_t        cache_depot_lock;       /* protects depot */
405         kmem_magtype_t  *cache_magtype;         /* magazine type */
406         kmem_maglist_t  cache_full;             /* full magazines */
407         kmem_maglist_t  cache_empty;            /* empty magazines */
408         kmem_dump_t     cache_dump;             /* used during crash dump */
401         void            *cache_dumpfreelist;    /* heap during crash dump */
402         void            *cache_dumplog;         /* log entry during dump */

410         /*
411          * Per-CPU layer
412          */
413         kmem_cpu_cache_t cache_cpu[1];          /* max_ncpus actual elements */
414 };
______unchanged_portion_omitted_
```