```
**********************************************************
   36430 Fri Oct 26 17:54:26 2012
new/usr/src/uts/common/os/dumpsubr.c
[mq]: core-v2
**********************************************************
```

```
/*
 * 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 2012, Josef 'Jeff' Sipek <jeffpc@31bits.net>. All rights reserved.
 */

#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>

/*
 * 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.  These can be set via /etc/system.
 * 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):
 *      if dump_plat_mincpu == 0, then always do single threaded dump
 *      if ncpu >= dump_plat_mincpu then try to use bzip2
```

```
 126  *
  92  * dump_metrics_on      if set, metrics are collected in the kernel, passed
  93  *      to savecore via the dump file, and recorded by savecore in
  94  *      METRICS.txt.
  95  */
 131 uint_t dump_ncpu_low = 4;       /* minimum config for parallel lzjb */
 132 uint_t dump_bzip2_level = 1;    /* bzip2 level (1-9) */

 134 /* Use dump_plat_mincpu_default unless this variable is set by /etc/system */
 135 #define MINCPU_NOT_SET  ((uint_t)-1)
 136 uint_t dump_plat_mincpu = MINCPU_NOT_SET;

  97 /* tunables for pre-reserved heap */
  98 uint_t dump_kmem_permap = 1024;
  99 uint_t dump_kmem_pages = 8;

 142 /* Define multiple buffers per helper to avoid stalling */
 143 #define NCBUF_PER_HELPER        2
 144 #define NCMAP_PER_HELPER        4

 146 /* minimum number of helpers configured */
 147 #define MINHELPERS      (dump_ncpu_low)
 148 #define MINCBUFS        (MINHELPERS * NCBUF_PER_HELPER)

 101 /*
 151  * Define constant parameters.
 152  *
 153  * CBUF_SIZE            size of an output buffer
 154  *
 155  * CBUF_MAPSIZE         size of virtual range for mapping pages
 156  *
 157  * CBUF_MAPNP           size of virtual range in pages
 158  *
 159  */
 160 #define DUMP_1KB        ((size_t)1 << 10)
 161 #define DUMP_1MB        ((size_t)1 << 20)
 162 #define CBUF_SIZE       ((size_t)1 << 17)
 163 #define CBUF_MAPSHIFT   (22)
 164 #define CBUF_MAPSIZE    ((size_t)1 << CBUF_MAPSHIFT)
 165 #define CBUF_MAPNP      ((size_t)1 << (CBUF_MAPSHIFT - PAGESHIFT))

 167 /*
 102  * Compression metrics are accumulated nano-second subtotals. The
 103  * results are normalized by the number of pages dumped. A report is
 104  * generated when dumpsys() completes and is saved in the dump image
 105  * after the trailing dump header.
 106  *
 107  * Metrics are always collected. Set the variable dump_metrics_on to
 108  * cause metrics to be saved in the crash file, where savecore will
 109  * save it in the file METRICS.txt.
 110  */
 111 #define PERPAGES \
 112         PERPAGE(bitmap) PERPAGE(map) PERPAGE(unmap) \
 113         PERPAGE(copy) PERPAGE(compress) \
 114         PERPAGE(write) \
 115         PERPAGE(inwait) PERPAGE(outwait)

 117 typedef struct perpage {
 118 #define PERPAGE(x) hrtime_t x;
 119         PERPAGES
 120 #undef PERPAGE
 121 } perpage_t;

 123 /*
 124  * This macro controls the code generation for collecting dump
 125  * performance information. By default, the code is generated, but
```



```
 126  * automatic saving of the information is disabled. If dump_metrics_on
 127  * is set to 1, the timing information is passed to savecore via the
 128  * crash file, where it is appended to the file dump-dir/METRICS.txt.
 129  */
 130 #define COLLECT_METRICS

 132 #ifdef COLLECT_METRICS
 133 uint_t dump_metrics_on = 0;     /* set to 1 to enable recording metrics */

 135 #define HRSTART(v, m)           v##ts.m = gethrtime()
 136 #define HRSTOP(v, m)            v.m += gethrtime() - v##ts.m
 137 #define HRBEGIN(v, m, s)        v##ts.m = gethrtime(); v.size += s
 138 #define HREND(v, m)             v.m += gethrtime() - v##ts.m
 139 #define HRNORM(v, m, n)         v.m /= (n)

 141 #else
 142 #define HRSTART(v, m)
 143 #define HRSTOP(v, m)
 144 #define HRBEGIN(v, m, s)
 145 #define HREND(v, m)
 146 #define HRNORM(v, m, n)
 147 #endif  /* COLLECT_METRICS */

 215 /*
 216  * Buffers for copying and compressing memory pages.
 217  *
 218  * cbuf_t buffer controllers: used for both input and output.
 219  *
 220  * The buffer state indicates how it is being used:
 221  *
 222  * CBUF_FREEMAP: CBUF_MAPSIZE virtual address range is available for
 223  * mapping input pages.
 224  *
 225  * CBUF_INREADY: input pages are mapped and ready for compression by a
 226  * helper.
 227  *
 228  * CBUF_USEDMAP: mapping has been consumed by a helper. Needs unmap.
 229  *
 230  * CBUF_FREEBUF: CBUF_SIZE output buffer, which is available.
 231  *
 232  * CBUF_WRITE: CBUF_SIZE block of compressed pages from a helper,
 233  * ready to write out.
 234  *
 235  * CBUF_ERRMSG: CBUF_SIZE block of error messages from a helper
 236  * (reports UE errors.)
 237  */

 239 typedef enum cbufstate {
 240         CBUF_FREEMAP,
 241         CBUF_INREADY,
 242         CBUF_USEDMAP,
 243         CBUF_FREEBUF,
 244         CBUF_WRITE,
 245         CBUF_ERRMSG
 246 } cbufstate_t;

 248 typedef struct cbuf cbuf_t;

 250 struct cbuf {
 251         cbuf_t *next;                   /* next in list */
 252         cbufstate_t state;              /* processing state */
 253         size_t used;                    /* amount used */
 254         size_t size;                    /* mem size */
 255         char *buf;                      /* kmem or vmem */
 256         pgcnt_t pagenum;                /* index to pfn map */
 257         pgcnt_t bitnum;                 /* first set bitnum */
```

```
258         pfn_t pfn;                      /* first pfn in mapped range */
259         int off;                        /* byte offset to first pfn */
260 };

149 static char dump_osimage_uuid[36 + 1];

151 #define isdigit(ch)     ((ch) >= '0' && (ch) <= '9')
152 #define isxdigit(ch)    (isdigit(ch) || ((ch) >= 'a' && (ch) <= 'f') || \
153                         ((ch) >= 'A' && (ch) <= 'F'))

155 /*
269  * cqueue_t queues: a uni-directional channel for communication
270  * from the master to helper tasks or vice-versa using put and
271  * get primitives. Both mappings and data buffers are passed via
272  * queues. Producers close a queue when done. The number of
273  * active producers is reference counted so the consumer can
274  * detect end of data. Concurrent access is mediated by atomic
275  * operations for panic dump, or mutex/cv for live dump.
276  *
277  * There a four queues, used as follows:
278  *
279  * Queue                Dataflow                NewState
280  * --------------------------------------------------
281  * mainq                master -> master        FREEMAP
282  * master has initialized or unmapped an input buffer
283  * --------------------------------------------------
284  * helperq              master -> helper        INREADY
285  * master has mapped input for use by helper
286  * --------------------------------------------------
287  * mainq                master <- helper        USEDMAP
288  * helper is done with input
289  * --------------------------------------------------
290  * freebufq             master -> helper        FREEBUF
291  * master has initialized or written an output buffer
292  * --------------------------------------------------
293  * mainq                master <- helper        WRITE
294  * block of compressed pages from a helper
295  * --------------------------------------------------
296  * mainq                master <- helper        ERRMSG
297  * error messages from a helper (memory error case)
298  * --------------------------------------------------
299  * writerq              master <- master        WRITE
300  * non-blocking queue of blocks to write
301  * --------------------------------------------------
302  */
303 typedef struct cqueue {
304         cbuf_t *volatile first;         /* first in list */
305         cbuf_t *last;                   /* last in list */
306         hrtime_t ts;                    /* timestamp */
307         hrtime_t empty;                 /* total time empty */
308         kmutex_t mutex;                 /* live state lock */
309         kcondvar_t cv;                  /* live wait var */
310         lock_t spinlock;                /* panic mode spin lock */
311         volatile uint_t open;           /* producer ref count */
312 } cqueue_t;

314 /*
315  * Convenience macros for using the cqueue functions
316  * Note that the caller must have defined "dumpsync_t *ds"
317  */
318 #define CQ_IS_EMPTY(q)                                  \
319         (ds->q.first == NULL)

321 #define CQ_OPEN(q)                                      \
322         atomic_inc_uint(&ds->q.open)
```



```
324 #define CQ_CLOSE(q)                                     \
325         dumpsys_close_cq(&ds->q, ds->live)

327 #define CQ_PUT(q, cp, st)                               \
328         dumpsys_put_cq(&ds->q, cp, st, ds->live)

330 #define CQ_GET(q)                                       \
331         dumpsys_get_cq(&ds->q, ds->live)

333 /*
156  * Dynamic state when dumpsys() is running.
157  */
158 typedef struct dumpsync {
159         pgcnt_t npages;                 /* subtotal of pages dumped */
160         pgcnt_t pages_mapped;           /* subtotal of pages mapped */
161         pgcnt_t pages_used;             /* subtotal of pages used per map */
162         size_t nwrite;                  /* subtotal of bytes written */
341         uint_t live;                    /* running live dump */
342         uint_t neednl;                  /* will need to print a newline */
163         uint_t percent;                 /* dump progress */
164         uint_t percent_done;            /* dump progress reported */
345         cqueue_t freebufq;              /* free kmem bufs for writing */
346         cqueue_t mainq;                 /* input for main task */
347         cqueue_t helperq;               /* input for helpers */
348         cqueue_t writerq;               /* input for writer */
165         hrtime_t start;                 /* start time */
166         hrtime_t elapsed;               /* elapsed time when completed */
167         hrtime_t iotime;                /* time spent writing nwrite bytes */
168         hrtime_t iowait;                /* time spent waiting for output */
169         hrtime_t iowaitts;              /* iowait timestamp */
170         perpage_t perpage;              /* metrics */
171         perpage_t perpagets;
356         int dumpcpu;                    /* master cpu */
172 } dumpsync_t;

174 static dumpsync_t dumpsync;             /* synchronization vars */

176 /*
177  * configuration vars for dumpsys
362  * helper_t helpers: contains the context for a stream. CPUs run in
363  * parallel at dump time; each CPU creates a single stream of
364  * compression data.  Stream data is divided into CBUF_SIZE blocks.
365  * The blocks are written in order within a stream. But, blocks from
366  * multiple streams can be interleaved. Each stream is identified by a
367  * unique tag.
178  */
179 typedef struct dumpcfg {
369 typedef struct helper {
370         int helper;                     /* bound helper id */
371         int tag;                        /* compression stream tag */
180         perpage_t perpage;      /* per page metrics */
181         perpage_t perpagets;    /* per page metrics (timestamps) */
374         taskqid_t taskqid;              /* live dump task ptr */
375         int in, out;                    /* buffer offsets */
376         cbuf_t *cpin, *cpout, *cperr;   /* cbuf objects in process */
377         dumpsync_t *ds;                 /* pointer to sync vars */
378         size_t used;                    /* counts input consumed */
182         char *page;             /* buffer for page copy */
183         char *lzbuf;            /* lzjb output */
381         bz_stream bzstream;             /* bzip2 state */
382 } helper_t;

185         char *cmap;             /* array of input (map) buffers */
384 #define MAINHELPER      (-1)            /* helper is also the main task */
385 #define FREEHELPER      (-2)            /* unbound helper */
386 #define DONEHELPER      (-3)            /* helper finished */
```

```
388 /*
389  * configuration vars for dumpsys
390  */
391 typedef struct dumpcfg {
392         int     threshold;      /* ncpu threshold for bzip2 */
393         int     nhelper;        /* number of helpers */
394         int     nhelper_used;   /* actual number of helpers used */
395         int     ncmap;          /* number VA pages for compression */
396         int     ncbuf;          /* number of bufs for compression */
397         int     ncbuf_used;     /* number of bufs in use */
398         uint_t  clevel;         /* dump compression level */
399         helper_t *helper;       /* array of helpers */
400         cbuf_t  *cmap;          /* array of input (map) buffers */
401         cbuf_t  *cbuf;          /* array of output  buffers */
402         ulong_t *helpermap;     /* set of dumpsys helper CPU ids */
186         ulong_t *bitmap;        /* bitmap for marking pages to dump */
404         ulong_t *rbitmap;       /* bitmap for used CBUF_MAPSIZE ranges */
187         pgcnt_t bitmapsize;     /* size of bitmap */
406         pgcnt_t rbitmapsize;    /* size of bitmap for ranges */
407         pgcnt_t found4m;        /* number ranges allocated by dump */
408         pgcnt_t foundsm;        /* number small pages allocated by dump */
188         pid_t *pids;            /* list of process IDs at dump time */
410         size_t  maxsize;        /* memory size needed at dump time */
411         size_t  maxvmsize;      /* size of reserved VM */
412         char    *maxvm;         /* reserved VM for spare pages */
413         lock_t  helper_lock;    /* protect helper state */
414         char    helpers_wanted; /* flag to enable parallelism */
189 } dumpcfg_t;
      unchanged_portion_omitted_

210 static dumpbuf_t dumpbuf;       /* I/O buffer */
436 dumpbuf_t dumpbuf;              /* I/O buffer */

212 /*
213  * The dump I/O buffer must be at least one page, at most xfer_size
214  * bytes, and should scale with physmem in between.  The transfer size
215  * passed in will either represent a global default (maxphys) or the
216  * best size for the device.  The size of the dumpbuf I/O buffer is
217  * limited by dumpbuf_limit (8MB by default) because the dump
218  * performance saturates beyond a certain size.  The default is to
219  * select 1/4096 of the memory.
220  */
221 static int      dumpbuf_fraction = 12;  /* memory size scale factor */
222 static size_t   dumpbuf_limit = 8 << 20;        /* max I/O buf size */
448 static size_t   dumpbuf_limit = 8 * DUMP_1MB;   /* max I/O buf size */

224 static size_t
225 dumpbuf_iosize(size_t xfer_size)
226 {
227         size_t iosize = ptob(physmem >> dumpbuf_fraction);

229         if (iosize < PAGESIZE)
230                 iosize = PAGESIZE;
231         else if (iosize > xfer_size)
232                 iosize = xfer_size;
233         if (iosize > dumpbuf_limit)
234                 iosize = dumpbuf_limit;
235         return (iosize & PAGEMASK);
236 }
      unchanged_portion_omitted_

262 /*
263  * dump_update_clevel is called when dumpadm configures the dump device.
490  *      Calculate number of helpers and buffers.
264  *      Allocate the minimum configuration for now.
```



```
265  *
266  * When the dump file is configured we reserve a minimum amount of
267  * memory for use at crash time. But we reserve VA for all the memory
268  * we really want in order to do the fastest dump possible. The VA is
269  * backed by pages not being dumped, according to the bitmap. If
270  * there is insufficient spare memory, however, we fall back to the
271  * minimum.
272  *
273  * Live dump (savecore -L) always uses the minimum config.
274  *
275  * For single-threaded dumps, the panic CPU does lzjb compression.
502  * clevel 0 is single threaded lzjb
503  * clevel 1 is parallel lzjb
504  * clevel 2 is parallel bzip2
276  *
506  * The ncpu threshold is selected with dump_plat_mincpu.
507  * On OPL, set_platform_defaults() overrides the sun4u setting.
508  * The actual values are defined via DUMP_PLAT_*_MINCPU macros.
509  *
510  * Architecture         Threshold       Algorithm
511  * sun4u                <  51           parallel lzjb
512  * sun4u                >= 51           parallel bzip2(*)
513  * sun4u OPL            <  8            parallel lzjb
514  * sun4u OPL            >= 8            parallel bzip2(*)
515  * sun4v                <  128          parallel lzjb
516  * sun4v                >= 128          parallel bzip2(*)
517  * x86                  < 11            parallel lzjb
518  * x86                  >= 11           parallel bzip2(*)
519  * 32-bit               N/A             single-threaded lzjb
520  *
521  * (*) bzip2 is only chosen if there is sufficient available
522  * memory for buffers at dump time. See dumpsys_get_maxmem().
523  *
524  * Faster dump devices have larger I/O buffers. The threshold value is
525  * increased according to the size of the dump I/O buffer, because
526  * parallel lzjb performs better with faster disks. For buffers >= 1MB
527  * the threshold is 3X; for buffers >= 256K threshold is 2X.
528  *
529  * For parallel dumps, the number of helpers is ncpu-1. The CPU
530  * running panic runs the main task. For single-threaded dumps, the
531  * panic CPU does lzjb compression (it is tagged as MAINHELPER.)
532  *
533  * Need multiple buffers per helper so that they do not block waiting
534  * for the main task.
535  *                              parallel        single-threaded
536  * Number of output buffers:    nhelper*2               1
537  * Number of mapping buffers:   nhelper*4               1
538  *
277  */
278 static void
279 dump_update_clevel()
280 {
543         int tag;
544         size_t bz2size;
545         helper_t *hp, *hpend;
546         cbuf_t *cp, *cpend;
281         dumpcfg_t *old = &dumpcfg;
282         dumpcfg_t newcfg = *old;
283         dumpcfg_t *new = &newcfg;

285         ASSERT(MUTEX_HELD(&dump_lock));

287         /*
288          * Free the previously allocated bufs and VM.
289          */
290         if (old->lzbuf)
```

```
291                 kmem_free(old->lzbuf, PAGESIZE);
292         if (old->page)
293                 kmem_free(old->page, PAGESIZE);
556         if (old->helper != NULL) {

295         if (old->cmap)
558                 /* helpers */
559                 hpend = &old->helper[old->nhelper];
560                 for (hp = old->helper; hp != hpend; hp++) {
561                         if (hp->lzbuf != NULL)
562                                 kmem_free(hp->lzbuf, PAGESIZE);
563                         if (hp->page != NULL)
564                                 kmem_free(hp->page, PAGESIZE);
565                 }
566                 kmem_free(old->helper, old->nhelper * sizeof (helper_t));

296                 /* VM space for mapping pages */
297                 vmem_xfree(heap_arena, old->cmap, PAGESIZE);
569                 cpend = &old->cmap[old->ncmap];
570                 for (cp = old->cmap; cp != cpend; cp++)
571                         vmem_xfree(heap_arena, cp->buf, CBUF_MAPSIZE);
572                 kmem_free(old->cmap, old->ncmap * sizeof (cbuf_t));

574                 /* output bufs */
575                 cpend = &old->cbuf[old->ncbuf];
576                 for (cp = old->cbuf; cp != cpend; cp++)
577                         if (cp->buf != NULL)
578                                 kmem_free(cp->buf, cp->size);
579                 kmem_free(old->cbuf, old->ncbuf * sizeof (cbuf_t));

581                 /* reserved VM for dumpsys_get_maxmem */
582                 if (old->maxvmsize > 0)
583                         vmem_xfree(heap_arena, old->maxvm, old->maxvmsize);
584         }

299         /*
300          * Allocate new data structures and buffers, and also figure the max
301          * desired size.
587          * Allocate memory and VM.
588          * One CPU runs dumpsys, the rest are helpers.
302          */
303         new->lzbuf = kmem_alloc(PAGESIZE, KM_SLEEP);
304         new->page = kmem_alloc(PAGESIZE, KM_SLEEP);
590         new->nhelper = ncpus - 1;
591         if (new->nhelper < 1)
592                 new->nhelper = 1;

306         new->cmap = vmem_xalloc(heap_arena, PAGESIZE, PAGESIZE,
594         if (new->nhelper > DUMP_MAX_NHELPER)
595                 new->nhelper = DUMP_MAX_NHELPER;

597         /* use platform default, unless /etc/system overrides */
598         if (dump_plat_mincpu == MINCPU_NOT_SET)
599                 dump_plat_mincpu = dump_plat_mincpu_default;

601         /* increase threshold for faster disks */
602         new->threshold = dump_plat_mincpu;
603         if (dumpbuf.iosize >= DUMP_1MB)
604                 new->threshold *= 3;
605         else if (dumpbuf.iosize >= (256 * DUMP_1KB))
606                 new->threshold *= 2;

608         /* figure compression level based upon the computed threshold. */
609         if (dump_plat_mincpu == 0 || new->nhelper < 2) {
610                 new->clevel = 0;
611                 new->nhelper = 1;
```



```
612         } else if ((new->nhelper + 1) >= new->threshold) {
613                 new->clevel = DUMP_CLEVEL_BZIP2;
614         } else {
615                 new->clevel = DUMP_CLEVEL_LZJB;
616         }

618         if (new->clevel == 0) {
619                 new->ncbuf = 1;
620                 new->ncmap = 1;
621         } else {
622                 new->ncbuf = NCBUF_PER_HELPER * new->nhelper;
623                 new->ncmap = NCMAP_PER_HELPER * new->nhelper;
624         }

626         /*
627          * Allocate new data structures and buffers for MINHELPERS,
628          * and also figure the max desired size.
629          */
630         bz2size = BZ2_bzCompressInitSize(dump_bzip2_level);
631         new->maxsize = 0;
632         new->maxvmsize = 0;
633         new->maxvm = NULL;
634         tag = 1;
635         new->helper = kmem_zalloc(new->nhelper * sizeof (helper_t), KM_SLEEP);
636         hpend = &new->helper[new->nhelper];
637         for (hp = new->helper; hp != hpend; hp++) {
638                 hp->tag = tag++;
639                 if (hp < &new->helper[MINHELPERS]) {
640                         hp->lzbuf = kmem_alloc(PAGESIZE, KM_SLEEP);
641                         hp->page = kmem_alloc(PAGESIZE, KM_SLEEP);
642                 } else if (new->clevel < DUMP_CLEVEL_BZIP2) {
643                         new->maxsize += 2 * PAGESIZE;
644                 } else {
645                         new->maxsize += PAGESIZE;
646                 }
647                 if (new->clevel >= DUMP_CLEVEL_BZIP2)
648                         new->maxsize += bz2size;
649         }

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

662         new->cmap = kmem_zalloc(new->ncmap * sizeof (cbuf_t), KM_SLEEP);
663         cpend = &new->cmap[new->ncmap];
664         for (cp = new->cmap; cp != cpend; cp++) {
665                 cp->state = CBUF_FREEMAP;
666                 cp->size = CBUF_MAPSIZE;
667                 cp->buf = vmem_xalloc(heap_arena, CBUF_MAPSIZE, CBUF_MAPSIZE,
307                                       0, 0, NULL, NULL, VM_SLEEP);
669         }

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

```
309         /*
310          * Reserve memory for kmem allocation calls made during crash
311          * dump.  The hat layer allocates memory for each mapping
312          * created, and the I/O path allocates buffers and data structs.
313          * Add a few pages for safety.
314          */
315         kmem_dump_init(dump_kmem_permap + (dump_kmem_pages * PAGESIZE));
685         kmem_dump_init((new->ncmap * dump_kmem_permap) +
686             (dump_kmem_pages * PAGESIZE));

317         /* set new config pointers */
318         *old = *new;
319 }
```

*unchanged_portion_omitted_*

```
756 /*
757  * Set/test bitmap for a CBUF_MAPSIZE range which includes pfn. The
758  * mapping of pfn to range index is imperfect because pfn and bitnum
759  * do not have the same phase. To make sure a CBUF_MAPSIZE range is
760  * covered, call this for both ends:
761  *      dump_set_used(base)
762  *      dump_set_used(base+CBUF_MAPNP-1)
763  *
764  * This is used during a panic dump to mark pages allocated by
765  * dumpsys_get_maxmem(). The macro IS_DUMP_PAGE(pp) is used by
766  * page_get_mnode_freelist() to make sure pages used by dump are never
767  * allocated.
768  */
769 #define CBUF_MAPP2R(pfn)        ((pfn) >> (CBUF_MAPSHIFT - PAGESHIFT))

385 static void
772 dump_set_used(pfn_t pfn)
773 {

775         pgcnt_t bitnum, rbitnum;

777         bitnum = dump_pfn_to_bitnum(pfn);
778         ASSERT(bitnum != (pgcnt_t)-1);

780         rbitnum = CBUF_MAPP2R(bitnum);
781         ASSERT(rbitnum < dumpcfg.rbitmapsize);

783         BT_SET(dumpcfg.rbitmap, rbitnum);
784 }

786 int
787 dump_test_used(pfn_t pfn)
788 {
789         pgcnt_t bitnum, rbitnum;

791         bitnum = dump_pfn_to_bitnum(pfn);
792         ASSERT(bitnum != (pgcnt_t)-1);

794         rbitnum = CBUF_MAPP2R(bitnum);
795         ASSERT(rbitnum < dumpcfg.rbitmapsize);

797         return (BT_TEST(dumpcfg.rbitmap, rbitnum));
798 }

800 /*
801  * dumpbzalloc and dumpbzfree are callbacks from the bzip2 library.
802  * dumpsys_get_maxmem() uses them for BZ2_bzCompressInit().
803  */
804 static void *
805 dumpbzalloc(void *opaque, int items, int size)
```



```
806 {
807         size_t *sz;
808         char *ret;

810         ASSERT(opaque != NULL);
811         sz = opaque;
812         ret = dumpcfg.maxvm + *sz;
813         *sz += items * size;
814         *sz = P2ROUNDUP(*sz, BZ2_BZALLOC_ALIGN);
815         ASSERT(*sz <= dumpcfg.maxvmsize);
816         return (ret);
817 }

819 /*ARGSUSED*/
820 static void
821 dumpbzfree(void *opaque, void *addr)
822 {
823 }

825 /*
826  * Perform additional checks on the page to see if we can really use
827  * it. The kernel (kas) pages are always set in the bitmap. However,
828  * boot memory pages (prom_ppages or P_BOOTPAGES) are not in the
829  * bitmap. So we check for them.
830  */
831 static inline int
832 dump_pfn_check(pfn_t pfn)
833 {
834         page_t *pp = page_numtopp_nolock(pfn);
835         if (pp == NULL || pp->p_pagenum != pfn ||
836 #if defined(__sparc)
837             pp->p_vnode == &promvp ||
838 #else
839             PP_ISBOOTPAGES(pp) ||
840 #endif
841             pp->p_toxic != 0)
842                 return (0);
843         return (1);
844 }

846 /*
847  * Check a range to see if all contained pages are available and
848  * return non-zero if the range can be used.
849  */
850 static inline int
851 dump_range_check(pgcnt_t start, pgcnt_t end, pfn_t pfn)
852 {
853         for (; start < end; start++, pfn++) {
854                 if (BT_TEST(dumpcfg.bitmap, start))
855                         return (0);
856                 if (!dump_pfn_check(pfn))
857                         return (0);
858         }
859         return (1);
860 }

862 /*
863  * dumpsys_get_maxmem() is called during panic. Find unused ranges
864  * and use them for buffers. If we find enough memory switch to
865  * parallel bzip2, otherwise use parallel lzjb.
866  *
867  * It searches the dump bitmap in 2 passes. The first time it looks
868  * for CBUF_MAPSIZE ranges. On the second pass it uses small pages.
869  */
870 static void
871 dumpsys_get_maxmem()
```

```
872 {
873         dumpcfg_t *cfg = &dumpcfg;
874         cbuf_t *endcp = &cfg->cbuf[cfg->ncbuf];
875         helper_t *endhp = &cfg->helper[cfg->nhelper];
876         pgcnt_t bitnum, end;
877         size_t sz, endsz, bz2size;
878         pfn_t pfn, off;
879         cbuf_t *cp;
880         helper_t *hp, *ohp;
881         dumpmlw_t mlw;
882         int k;
883
884         /*
885          * Setting dump_plat_mincpu to 0 at any time forces a serial
886          * dump.
887          */
888         if (dump_plat_mincpu == 0) {
889                 cfg->clevel = 0;
890                 return;
891         }
892
893         /*
894          * There may be no point in looking for spare memory. If
895          * dumping all memory, then none is spare. If doing a serial
896          * dump, then already have buffers.
897          */
898         if (cfg->maxsize == 0 || cfg->clevel < DUMP_CLEVEL_LZJB ||
899             (dump_conflags & DUMP_ALL) != 0) {
900                 if (cfg->clevel > DUMP_CLEVEL_LZJB)
901                         cfg->clevel = DUMP_CLEVEL_LZJB;
902                 return;
903         }
904
905         sz = 0;
906         cfg->found4m = 0;
907         cfg->foundsm = 0;
908
909         /* bitmap of ranges used to estimate which pfns are being used */
910         bzero(dumpcfg.rbitmap, BT_SIZEOFMAP(dumpcfg.rbitmapsize));
911
912         /* find ranges that are not being dumped to use for buffers */
913         dump_init_memlist_walker(&mlw);
914         for (bitnum = 0; bitnum < dumpcfg.bitmapsize; bitnum = end) {
915                 dump_timeleft = dump_timeout;
916                 end = bitnum + CBUF_MAPNP;
917                 pfn = dump_bitnum_to_pfn(bitnum, &mlw);
918                 ASSERT(pfn != PFN_INVALID);
919
920                 /* skip partial range at end of mem segment */
921                 if (mlw.mpleft < CBUF_MAPNP) {
922                         end = bitnum + mlw.mpleft;
923                         continue;
924                 }
925
926                 /* skip non aligned pages */
927                 off = P2PHASE(pfn, CBUF_MAPNP);
928                 if (off != 0) {
929                         end -= off;
930                         continue;
931                 }
932
933                 if (!dump_range_check(bitnum, end, pfn))
934                         continue;
935
936                 ASSERT((sz + CBUF_MAPSIZE) <= cfg->maxvmsize);
937                 hat_devload(kas.a_hat, cfg->maxvm + sz, CBUF_MAPSIZE, pfn,
```



```
938                     PROT_READ | PROT_WRITE, HAT_LOAD_NOCONSIST);
939                 sz += CBUF_MAPSIZE;
940                 cfg->found4m++;
941
942                 /* set the bitmap for both ends to be sure to cover the range */
943                 dump_set_used(pfn);
944                 dump_set_used(pfn + CBUF_MAPNP - 1);
945
946                 if (sz >= cfg->maxsize)
947                         goto foundmax;
948         }
949
950         /* Add small pages if we can't find enough large pages. */
951         dump_init_memlist_walker(&mlw);
952         for (bitnum = 0; bitnum < dumpcfg.bitmapsize; bitnum = end) {
953                 dump_timeleft = dump_timeout;
954                 end = bitnum + CBUF_MAPNP;
955                 pfn = dump_bitnum_to_pfn(bitnum, &mlw);
956                 ASSERT(pfn != PFN_INVALID);
957
958                 /* Find any non-aligned pages at start and end of segment. */
959                 off = P2PHASE(pfn, CBUF_MAPNP);
960                 if (mlw.mpleft < CBUF_MAPNP) {
961                         end = bitnum + mlw.mpleft;
962                 } else if (off != 0) {
963                         end -= off;
964                 } else if (cfg->found4m && dump_test_used(pfn)) {
965                         continue;
966                 }
967
968                 for (; bitnum < end; bitnum++, pfn++) {
969                         dump_timeleft = dump_timeout;
970                         if (BT_TEST(dumpcfg.bitmap, bitnum))
971                                 continue;
972                         if (!dump_pfn_check(pfn))
973                                 continue;
974                         ASSERT((sz + PAGESIZE) <= cfg->maxvmsize);
975                         hat_devload(kas.a_hat, cfg->maxvm + sz, PAGESIZE, pfn,
976                             PROT_READ | PROT_WRITE, HAT_LOAD_NOCONSIST);
977                         sz += PAGESIZE;
978                         cfg->foundsm++;
979                         dump_set_used(pfn);
980                         if (sz >= cfg->maxsize)
981                                 goto foundmax;
982                 }
983         }
984
985         /* Fall back to lzjb if we did not get enough memory for bzip2. */
986         endsz = (cfg->maxsize * cfg->threshold) / cfg->nhelper;
987         if (sz < endsz) {
988                 cfg->clevel = DUMP_CLEVEL_LZJB;
989         }
990
991         /* Allocate memory for as many helpers as we can. */
992 foundmax:
993
994         /* Byte offsets into memory found and mapped above */
995         endsz = sz;
996         sz = 0;
997
998         /* Set the size for bzip2 state. Only bzip2 needs it. */
999         bz2size = BZ2_bzCompressInitSize(dump_bzip2_level);
1000
1001         /* Skip the preallocate output buffers. */
1002         cp = &cfg->cbuf[MINCBUFS];
```

```
1004         /* Use this to move memory up from the preallocated helpers. */
1005         ohp = cfg->helper;
1006
1007         /* Loop over all helpers and allocate memory. */
1008         for (hp = cfg->helper; hp < endhp; hp++) {
1009
1010                 /* Skip preallocated helpers by checking hp->page. */
1011                 if (hp->page == NULL) {
1012                         if (cfg->clevel <= DUMP_CLEVEL_LZJB) {
1013                                 /* lzjb needs 2 1-page buffers */
1014                                 if ((sz + (2 * PAGESIZE)) > endsz)
1015                                         break;
1016                                 hp->page = cfg->maxvm + sz;
1017                                 sz += PAGESIZE;
1018                                 hp->lzbuf = cfg->maxvm + sz;
1019                                 sz += PAGESIZE;
1020
1021                         } else if (ohp->lzbuf != NULL) {
1022                                 /* re-use the preallocted lzjb page for bzip2 */
1023                                 hp->page = ohp->lzbuf;
1024                                 ohp->lzbuf = NULL;
1025                                 ++ohp;
1026
1027                         } else {
1028                                 /* bzip2 needs a 1-page buffer */
1029                                 if ((sz + PAGESIZE) > endsz)
1030                                         break;
1031                                 hp->page = cfg->maxvm + sz;
1032                                 sz += PAGESIZE;
1033                         }
1034                 }
1035
1036                 /*
1037                  * Add output buffers per helper. The number of
1038                  * buffers per helper is determined by the ratio of
1039                  * ncbuf to nhelper.
1040                  */
1041                 for (k = 0; cp < endcp && (sz + CBUF_SIZE) <= endsz &&
1042                     k < NCBUF_PER_HELPER; k++) {
1043                         cp->state = CBUF_FREEBUF;
1044                         cp->size = CBUF_SIZE;
1045                         cp->buf = cfg->maxvm + sz;
1046                         sz += CBUF_SIZE;
1047                         ++cp;
1048                 }
1049
1050                 /*
1051                  * bzip2 needs compression state. Use the dumpbzalloc
1052                  * and dumpbzfree callbacks to allocate the memory.
1053                  * bzip2 does allocation only at init time.
1054                  */
1055                 if (cfg->clevel >= DUMP_CLEVEL_BZIP2) {
1056                         if ((sz + bz2size) > endsz) {
1057                                 hp->page = NULL;
1058                                 break;
1059                         } else {
1060                                 hp->bzstream.opaque = &sz;
1061                                 hp->bzstream.bzalloc = dumpbzalloc;
1062                                 hp->bzstream.bzfree = dumpbzfree;
1063                                 (void) BZ2_bzCompressInit(&hp->bzstream,
1064                                     dump_bzip2_level, 0, 0);
1065                                 hp->bzstream.opaque = NULL;
1066                         }
1067                 }
1068         }
```



```
1070         /* Finish allocating output buffers */
1071         for (; cp < endcp && (sz + CBUF_SIZE) <= endsz; cp++) {
1072                 cp->state = CBUF_FREEBUF;
1073                 cp->size = CBUF_SIZE;
1074                 cp->buf = cfg->maxvm + sz;
1075                 sz += CBUF_SIZE;
1076         }
1077
1078         /* Enable IS_DUMP_PAGE macro, which checks for pages we took. */
1079         if (cfg->found4m || cfg->foundsm)
1080                 dump_check_used = 1;
1081
1082         ASSERT(sz <= endsz);
1083 }
1084
1085 static void
 386 dumphdr_init(void)
 387 {
 388         pgcnt_t npages;
1088         pgcnt_t npages = 0;
1089
 390         ASSERT(MUTEX_HELD(&dump_lock));
1091
 392         if (dumphdr == NULL) {
 393                 dumphdr = kmem_zalloc(sizeof (dumphdr_t), KM_SLEEP);
 394                 dumphdr->dump_magic = DUMP_MAGIC;
 395                 dumphdr->dump_version = DUMP_VERSION;
 396                 dumphdr->dump_wordsize = DUMP_WORDSIZE;
 397                 dumphdr->dump_pageshift = PAGESHIFT;
 398                 dumphdr->dump_pagesize = PAGESIZE;
 399                 dumphdr->dump_utsname = utsname;
 400                 (void) strcpy(dumphdr->dump_platform, platform);
 401                 dumpbuf.size = dumpbuf_iosize(maxphys);
 402                 dumpbuf.start = kmem_alloc(dumpbuf.size, KM_SLEEP);
 403                 dumpbuf.end = dumpbuf.start + dumpbuf.size;
 404                 dumpcfg.pids = kmem_alloc(v.v_proc * sizeof (pid_t), KM_SLEEP);
1105                 dumpcfg.helpermap = kmem_zalloc(BT_SIZEOFMAP(NCPU), KM_SLEEP);
1106                 LOCK_INIT_HELD(&dumpcfg.helper_lock);
 405                 dump_stack_scratch = kmem_alloc(STACK_BUF_SIZE, KM_SLEEP);
 406                 (void) strncpy(dumphdr->dump_uuid, dump_get_uuid(),
 407                     sizeof (dumphdr->dump_uuid));
 408         }
1109
 410         npages = num_phys_pages();
1111
 412         if (dumpcfg.bitmapsize != npages) {
1115                 size_t rlen = CBUF_MAPP2R(P2ROUNDUP(npages, CBUF_MAPNP));
 413                 void *map = kmem_alloc(BT_SIZEOFMAP(npages), KM_SLEEP);
1117                 void *rmap = kmem_alloc(BT_SIZEOFMAP(rlen), KM_SLEEP);
1118
 415                 if (dumpcfg.bitmap != NULL)
 416                         kmem_free(dumpcfg.bitmap, BT_SIZEOFMAP(dumpcfg.
 417                             bitmapsize));
1122                 if (dumpcfg.rbitmap != NULL)
1123                         kmem_free(dumpcfg.rbitmap, BT_SIZEOFMAP(dumpcfg.
1124                             rbitmapsize));
 418                 dumpcfg.bitmap = map;
 419                 dumpcfg.bitmapsize = npages;
1127                 dumpcfg.rbitmap = rmap;
1128                 dumpcfg.rbitmapsize = rlen;
 420         }
 421 }
_____unchanged_portion_omitted_

 848 /*
1558  * The following functions are called on multiple CPUs during dump.
```

```
 * They must not use most kernel services, because all cross-calls are
 * disabled during panic. Therefore, blocking locks and cache flushes
 * will not work.
 */

/*
 * Copy pages, trapping ECC errors. Also, for robustness, trap data
 * access in case something goes wrong in the hat layer and the
 * mapping is broken.
 */
static int
dump_pagecopy(void *src, void *dst)
{
        long *wsrc = (long *)src;
        long *wdst = (long *)dst;
        const ulong_t ncopies = PAGESIZE / sizeof (long);
        volatile int w = 0;
        volatile int ueoff = -1;
        on_trap_data_t otd;

        if (on_trap(&otd, OT_DATA_EC | OT_DATA_ACCESS)) {
                if (ueoff == -1)
                        ueoff = w * sizeof (long);
                /* report "bad ECC" or "bad address" */
#ifdef _LP64
                if (otd.ot_trap & OT_DATA_EC)
                        wdst[w++] = 0x00badecc00badecc;
                else
                        wdst[w++] = 0x00badadd00badadd;
#else
                if (otd.ot_trap & OT_DATA_EC)
                        wdst[w++] = 0x00badecc;
                else
                        wdst[w++] = 0x00badadd;
#endif
        }
        while (w < ncopies) {
                wdst[w] = wsrc[w];
                w++;
        }
        no_trap();
        return (ueoff);
}

static void
dumpsys_close_cq(cqueue_t *cq, int live)
{
        if (live) {
                mutex_enter(&cq->mutex);
                atomic_dec_uint(&cq->open);
                cv_signal(&cq->cv);
                mutex_exit(&cq->mutex);
        } else {
                atomic_dec_uint(&cq->open);
        }
}

static inline void
dumpsys_spinlock(lock_t *lp)
{
        uint_t backoff = 0;
        int loop_count = 0;

        while (LOCK_HELD(lp) || !lock_spin_try(lp)) {
                if (++loop_count >= ncpus) {
                        backoff = mutex_lock_backoff(0);
```



```
                        loop_count = 0;
                } else {
                        backoff = mutex_lock_backoff(backoff);
                }
                mutex_lock_delay(backoff);
        }
}

static inline void
dumpsys_spinunlock(lock_t *lp)
{
        lock_clear(lp);
}

static inline void
dumpsys_lock(cqueue_t *cq, int live)
{
        if (live)
                mutex_enter(&cq->mutex);
        else
                dumpsys_spinlock(&cq->spinlock);
}

static inline void
dumpsys_unlock(cqueue_t *cq, int live, int signal)
{
        if (live) {
                if (signal)
                        cv_signal(&cq->cv);
                mutex_exit(&cq->mutex);
        } else {
                dumpsys_spinunlock(&cq->spinlock);
        }
}

static void
dumpsys_wait_cq(cqueue_t *cq, int live)
{
        if (live) {
                cv_wait(&cq->cv, &cq->mutex);
        } else {
                dumpsys_spinunlock(&cq->spinlock);
                while (cq->open)
                        if (cq->first)
                                break;
                dumpsys_spinlock(&cq->spinlock);
        }
}

static void
dumpsys_put_cq(cqueue_t *cq, cbuf_t *cp, int newstate, int live)
{
        if (cp == NULL)
                return;

        dumpsys_lock(cq, live);

        if (cq->ts != 0) {
                cq->empty += gethrtime() - cq->ts;
                cq->ts = 0;
        }

        cp->state = newstate;
        cp->next = NULL;
        if (cq->last == NULL)
                cq->first = cp;
```

```
1691         else
1692                 cq->last->next = cp;
1693         cq->last = cp;
1694
1695         dumpsys_unlock(cq, live, 1);
1696 }
1697
1698 static cbuf_t *
1699 dumpsys_get_cq(cqueue_t *cq, int live)
1700 {
1701         cbuf_t *cp;
1702         hrtime_t now = gethrtime();
1703
1704         dumpsys_lock(cq, live);
1705
1706         /* CONSTCOND */
1707         while (1) {
1708                 cp = (cbuf_t *)cq->first;
1709                 if (cp == NULL) {
1710                         if (cq->open == 0)
1711                                 break;
1712                         dumpsys_wait_cq(cq, live);
1713                         continue;
1714                 }
1715                 cq->first = cp->next;
1716                 if (cq->first == NULL) {
1717                         cq->last = NULL;
1718                         cq->ts = now;
1719                 }
1720                 break;
1721         }
1722
1723         dumpsys_unlock(cq, live, cq->first != NULL || cq->open == 0);
1724         return (cp);
1725 }
1726
1727 /*
1728  * Send an error message to the console. If the main task is running
1729  * just write the message via uprintf. If a helper is running the
1730  * message has to be put on a queue for the main task. Setting fmt to
1731  * NULL means flush the error message buffer. If fmt is not NULL, just
1732  * add the text to the existing buffer.
1733  */
1734 static void
1735 dumpsys_errmsg(helper_t *hp, const char *fmt, ...)
1736 {
1737         dumpsync_t *ds = hp->ds;
1738         cbuf_t *cp = hp->cperr;
1739         va_list adx;
1740
1741         if (hp->helper == MAINHELPER) {
1742                 if (fmt != NULL) {
1743                         if (ds->neednl) {
1744                                 uprintf("\n");
1745                                 ds->neednl = 0;
1746                         }
1747                         va_start(adx, fmt);
1748                         vuprintf(fmt, adx);
1749                         va_end(adx);
1750                 }
1751         } else if (fmt == NULL) {
1752                 if (cp != NULL) {
1753                         CQ_PUT(mainq, cp, CBUF_ERRMSG);
1754                         hp->cperr = NULL;
1755                 }
1756         } else {
```



```
1757                 if (hp->cperr == NULL) {
1758                         cp = CQ_GET(freebufq);
1759                         hp->cperr = cp;
1760                         cp->used = 0;
1761                 }
1762                 va_start(adx, fmt);
1763                 cp->used += vsnprintf(cp->buf + cp->used, cp->size - cp->used,
1764                     fmt, adx);
1765                 va_end(adx);
1766                 if ((cp->used + LOG_MSGSIZE) > cp->size) {
1767                         CQ_PUT(mainq, cp, CBUF_ERRMSG);
1768                         hp->cperr = NULL;
1769                 }
1770         }
1771 }
1772
1773 /*
1774  * Write an output buffer to the dump file. If the main task is
1775  * running just write the data. If a helper is running the output is
1776  * placed on a queue for the main task.
1777  */
1778 static void
1779 dumpsys_swrite(helper_t *hp, cbuf_t *cp, size_t used)
1780 {
1781         dumpsync_t *ds = hp->ds;
1782
1783         if (hp->helper == MAINHELPER) {
1784                 HRSTART(ds->perpage, write);
1785                 dumpvp_write(cp->buf, used);
1786                 HRSTOP(ds->perpage, write);
1787                 CQ_PUT(freebufq, cp, CBUF_FREEBUF);
1788         } else {
1789                 cp->used = used;
1790                 CQ_PUT(mainq, cp, CBUF_WRITE);
1791         }
1792 }
1793
1794 /*
1795  * Copy one page within the mapped range. The offset starts at 0 and
1796  * is relative to the first pfn. cp->buf + cp->off is the address of
1797  * the first pfn. If dump_pagecopy returns a UE offset, create an
1798  * error message.  Returns the offset to the next pfn in the range
1799  * selected by the bitmap.
1800  */
1801 static int
1802 dumpsys_copy_page(helper_t *hp, int offset)
1803 {
1804         cbuf_t *cp = hp->cpin;
1805         int ueoff;
1806
1807         ASSERT(cp->off + offset + PAGESIZE <= cp->size);
1808         ASSERT(BT_TEST(dumpcfg.bitmap, cp->bitnum));
1809
1810         ueoff = dump_pagecopy(cp->buf + cp->off + offset, hp->page);
1811
1812         /* ueoff is the offset in the page to a UE error */
1813         if (ueoff != -1) {
1814                 uint64_t pa = ptob(cp->pfn) + offset + ueoff;
1815
1816                 dumpsys_errmsg(hp, "cpu %d: memory error at PA 0x%08x.%08x\n",
1817                     CPU->cpu_id, (uint32_t)(pa >> 32), (uint32_t)pa);
1818         }
1819
1820         /*
1821          * Advance bitnum and offset to the next input page for the
1822          * next call to this function.
```

```
1823          */
1824         offset += PAGESIZE;
1825         cp->bitnum++;
1826         while (cp->off + offset < cp->size) {
1827                 if (BT_TEST(dumpcfg.bitmap, cp->bitnum))
1828                         break;
1829                 offset += PAGESIZE;
1830                 cp->bitnum++;
1831         }

1833         return (offset);
1834 }

1836 /*
1837  * Read the helper queue, and copy one mapped page. Return 0 when
1838  * done. Return 1 when a page has been copied into hp->page.
1839  */
1840 static int
1841 dumpsys_sread(helper_t *hp)
1842 {
1843         dumpsync_t *ds = hp->ds;

1845         /* CONSTCOND */
1846         while (1) {

1848                 /* Find the next input buffer. */
1849                 if (hp->cpin == NULL) {
1850                         HRSTART(hp->perpage, inwait);

1852                         /* CONSTCOND */
1853                         while (1) {
1854                                 hp->cpin = CQ_GET(helperq);
1855                                 dump_timeleft = dump_timeout;

1857                                 /*
1858                                  * NULL return means the helper queue
1859                                  * is closed and empty.
1860                                  */
1861                                 if (hp->cpin == NULL)
1862                                         break;

1864                                 /* Have input, check for dump I/O error. */
1865                                 if (!dump_ioerr)
1866                                         break;

1868                                 /*
1869                                  * If an I/O error occurs, stay in the
1870                                  * loop in order to empty the helper
1871                                  * queue. Return the buffers to the
1872                                  * main task to unmap and free it.
1873                                  */
1874                                 hp->cpin->used = 0;
1875                                 CQ_PUT(mainq, hp->cpin, CBUF_USEDMAP);
1876                         }
1877                         HRSTOP(hp->perpage, inwait);

1879                         /* Stop here when the helper queue is closed. */
1880                         if (hp->cpin == NULL)
1881                                 break;

1883                         /* Set the offset=0 to get the first pfn. */
1884                         hp->in = 0;

1886                         /* Set the total processed to 0 */
1887                         hp->used = 0;
1888                 }
```



```
1890                 /* Process the next page. */
1891                 if (hp->used < hp->cpin->used) {

1893                         /*
1894                          * Get the next page from the input buffer and
1895                          * return a copy.
1896                          */
1897                         ASSERT(hp->in != -1);
1898                         HRSTART(hp->perpage, copy);
1899                         hp->in = dumpsys_copy_page(hp, hp->in);
1900                         hp->used += PAGESIZE;
1901                         HRSTOP(hp->perpage, copy);
1902                         break;

1904                 } else {

1906                         /*
1907                          * Done with the input. Flush the VM and
1908                          * return the buffer to the main task.
1909                          */
1910                         if (panicstr && hp->helper != MAINHELPER)
1911                                 hat_flush_range(kas.a_hat,
1912                                     hp->cpin->buf, hp->cpin->size);
1913                         dumpsys_errmsg(hp, NULL);
1914                         CQ_PUT(mainq, hp->cpin, CBUF_USEDMAP);
1915                         hp->cpin = NULL;
1916                 }
1917         }

1919         return (hp->cpin != NULL);
1920 }

1922 /*
1923  * Compress size bytes starting at buf with bzip2
1924  * mode:
1925  *      BZ_RUN          add one more compressed page
1926  *      BZ_FINISH       no more input, flush the state
1927  */
1928 static void
1929 dumpsys_bzrun(helper_t *hp, void *buf, size_t size, int mode)
1930 {
1931         dumpsync_t *ds = hp->ds;
1932         const int CSIZE = sizeof (dumpcsize_t);
1933         bz_stream *ps = &hp->bzstream;
1934         int rc = 0;
1935         uint32_t csize;
1936         dumpcsize_t cs;

1938         /* Set input pointers to new input page */
1939         if (size > 0) {
1940                 ps->avail_in = size;
1941                 ps->next_in = buf;
1942         }

1944         /* CONSTCOND */
1945         while (1) {

1947                 /* Quit when all input has been consumed */
1948                 if (ps->avail_in == 0 && mode == BZ_RUN)
1949                         break;

1951                 /* Get a new output buffer */
1952                 if (hp->cpout == NULL) {
1953                         HRSTART(hp->perpage, outwait);
1954                         hp->cpout = CQ_GET(freebufq);
```

```
1955                        HRSTOP(hp->perpage, outwait);
1956                        ps->avail_out = hp->cpout->size - CSIZE;
1957                        ps->next_out = hp->cpout->buf + CSIZE;
1958                }

1960                /* Compress input, or finalize */
1961                HRSTART(hp->perpage, compress);
1962                rc = BZ2_bzCompress(ps, mode);
1963                HRSTOP(hp->perpage, compress);

1965                /* Check for error */
1966                if (mode == BZ_RUN && rc != BZ_RUN_OK) {
1967                        dumpsys_errmsg(hp, "%d: BZ_RUN error %s at page %lx\n",
1968                            hp->helper, BZ2_bzErrorString(rc),
1969                            hp->cpin->pagenum);
1970                        break;
1971                }

1973                /* Write the buffer if it is full, or we are flushing */
1974                if (ps->avail_out == 0 || mode == BZ_FINISH) {
1975                        csize = hp->cpout->size - CSIZE - ps->avail_out;
1976                        cs = DUMP_SET_TAG(csize, hp->tag);
1977                        if (csize > 0) {
1978                                (void) memcpy(hp->cpout->buf, &cs, CSIZE);
1979                                dumpsys_swrite(hp, hp->cpout, csize + CSIZE);
1980                                hp->cpout = NULL;
1981                        }
1982                }

1984                /* Check for final complete */
1985                if (mode == BZ_FINISH) {
1986                        if (rc == BZ_STREAM_END)
1987                                break;
1988                        if (rc != BZ_FINISH_OK) {
1989                                dumpsys_errmsg(hp, "%d: BZ_FINISH error %s\n",
1990                                    hp->helper, BZ2_bzErrorString(rc));
1991                                break;
1992                        }
1993                }
1994        }

1996        /* Cleanup state and buffers */
1997        if (mode == BZ_FINISH) {

1999                /* Reset state so that it is re-usable. */
2000                (void) BZ2_bzCompressReset(&hp->bzstream);

2002                /* Give any unused outout buffer to the main task */
2003                if (hp->cpout != NULL) {
2004                        hp->cpout->used = 0;
2005                        CQ_PUT(mainq, hp->cpout, CBUF_ERRMSG);
2006                        hp->cpout = NULL;
2007                }
2008        }
2009 }

2011 static void
2012 dumpsys_bz2compress(helper_t *hp)
2013 {
2014        dumpsync_t *ds = hp->ds;
2015        dumpstreamhdr_t sh;

2017        (void) strcpy(sh.stream_magic, DUMP_STREAM_MAGIC);
2018        sh.stream_pagenum = (pgcnt_t)-1;
2019        sh.stream_npages = 0;
2020        hp->cpin = NULL;
```



```
2021        hp->cpout = NULL;
2022        hp->cperr = NULL;
2023        hp->in = 0;
2024        hp->out = 0;
2025        hp->bzstream.avail_in = 0;

2027        /* Bump reference to mainq while we are running */
2028        CQ_OPEN(mainq);

2030        /* Get one page at a time */
2031        while (dumpsys_sread(hp)) {
2032                if (sh.stream_pagenum != hp->cpin->pagenum) {
2033                        sh.stream_pagenum = hp->cpin->pagenum;
2034                        sh.stream_npages = btop(hp->cpin->used);
2035                        dumpsys_bzrun(hp, &sh, sizeof (sh), BZ_RUN);
2036                }
2037                dumpsys_bzrun(hp, hp->page, PAGESIZE, 0);
2038        }

2040        /* Done with input, flush any partial buffer */
2041        if (sh.stream_pagenum != (pgcnt_t)-1) {
2042                dumpsys_bzrun(hp, NULL, 0, BZ_FINISH);
2043                dumpsys_errmsg(hp, NULL);
2044        }

2046        ASSERT(hp->cpin == NULL && hp->cpout == NULL && hp->cperr == NULL);

2048        /* Decrement main queue count, we are done */
2049        CQ_CLOSE(mainq);
2050 }

2052 /*
2053  * Compress with lzjb
2054  * write stream block if full or size==0
2055  * if csize==0 write stream header, else write <csize, data>
2056  * size==0 is a call to flush a buffer
2057  * hp->cpout is the buffer we are flushing or filling
2058  * hp->out is the next index to fill data
2059  * osize is either csize+data, or the size of a stream header
2060  */
2061 static void
2062 dumpsys_lzjbrun(helper_t *hp, size_t csize, void *buf, size_t size)
2063 {
2064        dumpsync_t *ds = hp->ds;
2065        const int CSIZE = sizeof (dumpcsize_t);
2066        dumpcsize_t cs;
2067        size_t osize = csize > 0 ? CSIZE + size : size;

2069        /* If flush, and there is no buffer, just return */
2070        if (size == 0 && hp->cpout == NULL)
2071                return;

2073        /* If flush, or cpout is full, write it out */
2074        if (size == 0 ||
2075            hp->cpout != NULL && hp->out + osize > hp->cpout->size) {

2077                /* Set tag+size word at the front of the stream block. */
2078                cs = DUMP_SET_TAG(hp->out - CSIZE, hp->tag);
2079                (void) memcpy(hp->cpout->buf, &cs, CSIZE);

2081                /* Write block to dump file. */
2082                dumpsys_swrite(hp, hp->cpout, hp->out);

2084                /* Clear pointer to indicate we need a new buffer */
2085                hp->cpout = NULL;
```

```
2087                 /* flushing, we are done */
2088                 if (size == 0)
2089                         return;
2090         }

2092         /* Get an output buffer if we dont have one. */
2093         if (hp->cpout == NULL) {
2094                 HRSTART(hp->perpage, outwait);
2095                 hp->cpout = CQ_GET(freebufq);
2096                 HRSTOP(hp->perpage, outwait);
2097                 hp->out = CSIZE;
2098         }

2100         /* Store csize word. This is the size of compressed data. */
2101         if (csize > 0) {
2102                 cs = DUMP_SET_TAG(csize, 0);
2103                 (void) memcpy(hp->cpout->buf + hp->out, &cs, CSIZE);
2104                 hp->out += CSIZE;
2105         }

2107         /* Store the data. */
2108         (void) memcpy(hp->cpout->buf + hp->out, buf, size);
2109         hp->out += size;
2110 }

2112 static void
2113 dumpsys_lzjbcompress(helper_t *hp)
2114 {
2115         dumpsync_t *ds = hp->ds;
2116         size_t csize;
2117         dumpstreamhdr_t sh;

2119         (void) strcpy(sh.stream_magic, DUMP_STREAM_MAGIC);
2120         sh.stream_pagenum = (pfn_t)-1;
2121         sh.stream_npages = 0;
2122         hp->cpin = NULL;
2123         hp->cpout = NULL;
2124         hp->cperr = NULL;
2125         hp->in = 0;
2126         hp->out = 0;

2128         /* Bump reference to mainq while we are running */
2129         CQ_OPEN(mainq);

2131         /* Get one page at a time */
2132         while (dumpsys_sread(hp)) {

2134                 /* Create a stream header for each new input map */
2135                 if (sh.stream_pagenum != hp->cpin->pagenum) {
2136                         sh.stream_pagenum = hp->cpin->pagenum;
2137                         sh.stream_npages = btop(hp->cpin->used);
2138                         dumpsys_lzjbrun(hp, 0, &sh, sizeof (sh));
2139                 }

2141                 /* Compress one page */
2142                 HRSTART(hp->perpage, compress);
2143                 csize = compress(hp->page, hp->lzbuf, PAGESIZE);
2144                 HRSTOP(hp->perpage, compress);

2146                 /* Add csize+data to output block */
2147                 ASSERT(csize > 0 && csize <= PAGESIZE);
2148                 dumpsys_lzjbrun(hp, csize, hp->lzbuf, csize);
2149         }

2151         /* Done with input, flush any partial buffer */
2152         if (sh.stream_pagenum != (pfn_t)-1) {
```



```
2153                 dumpsys_lzjbrun(hp, 0, NULL, 0);
2154                 dumpsys_errmsg(hp, NULL);
2155         }

2157         ASSERT(hp->cpin == NULL && hp->cpout == NULL && hp->cperr == NULL);

2159         /* Decrement main queue count, we are done */
2160         CQ_CLOSE(mainq);
2161 }

2163 /*
2164  * Dump helper called from panic_idle() to compress pages.  CPUs in
2165  * this path must not call most kernel services.
2166  *
2167  * During panic, all but one of the CPUs is idle. These CPUs are used
2168  * as helpers working in parallel to copy and compress memory
2169  * pages. During a panic, however, these processors cannot call any
2170  * kernel services. This is because mutexes become no-ops during
2171  * panic, and, cross-call interrupts are inhibited.  Therefore, during
2172  * panic dump the helper CPUs communicate with the panic CPU using
2173  * memory variables. All memory mapping and I/O is performed by the
2174  * panic CPU.
2175  *
2176  * At dump configuration time, helper_lock is set and helpers_wanted
2177  * is 0. dumpsys() decides whether to set helpers_wanted before
2178  * clearing helper_lock.
2179  *
2180  * At panic time, idle CPUs spin-wait on helper_lock, then alternately
2181  * take the lock and become a helper, or return.
2182  */
2183 void
2184 dumpsys_helper()
2185 {
2186         dumpsys_spinlock(&dumpcfg.helper_lock);
2187         if (dumpcfg.helpers_wanted) {
2188                 helper_t *hp, *hpend = &dumpcfg.helper[dumpcfg.nhelper];

2190                 for (hp = dumpcfg.helper; hp != hpend; hp++) {
2191                         if (hp->helper == FREEHELPER) {
2192                                 hp->helper = CPU->cpu_id;
2193                                 BT_SET(dumpcfg.helpermap, CPU->cpu_seqid);

2195                                 dumpsys_spinunlock(&dumpcfg.helper_lock);

2197                                 if (dumpcfg.clevel < DUMP_CLEVEL_BZIP2)
2198                                         dumpsys_lzjbcompress(hp);
2199                                 else
2200                                         dumpsys_bz2compress(hp);

2202                                 hp->helper = DONEHELPER;
2203                                 return;
2204                         }
2205                 }

2207                 /* No more helpers are needed. */
2208                 dumpcfg.helpers_wanted = 0;

2210         }
2211         dumpsys_spinunlock(&dumpcfg.helper_lock);
2212 }

2214 /*
2215  * No-wait helper callable in spin loops.
2216  *
2217  * Do not wait for helper_lock. Just check helpers_wanted. The caller
2218  * may decide to continue. This is the "c)ontinue, s)ync, r)eset? s"
```

```
 * case.
 */
void
dumpsys_helper_nw()
{
        if (dumpcfg.helpers_wanted)
                dumpsys_helper();
}

/*
 * Dump helper for live dumps.
 * These run as a system task.
 */
static void
dumpsys_live_helper(void *arg)
{
        helper_t *hp = arg;

        BT_ATOMIC_SET(dumpcfg.helpermap, CPU->cpu_seqid);
        if (dumpcfg.clevel < DUMP_CLEVEL_BZIP2)
                dumpsys_lzjbcompress(hp);
        else
                dumpsys_bz2compress(hp);
}

/*
 * Compress one page with lzjb (single threaded case)
 */
static void
dumpsys_lzjb_page(helper_t *hp, cbuf_t *cp)
{
        dumpsync_t *ds = hp->ds;
        uint32_t csize;

        hp->helper = MAINHELPER;
        hp->in = 0;
        hp->used = 0;
        hp->cpin = cp;
        while (hp->used < cp->used) {
                HRSTART(hp->perpage, copy);
                hp->in = dumpsys_copy_page(hp, hp->in);
                hp->used += PAGESIZE;
                HRSTOP(hp->perpage, copy);

                HRSTART(hp->perpage, compress);
                csize = compress(hp->page, hp->lzbuf, PAGESIZE);
                HRSTOP(hp->perpage, compress);

                HRSTART(hp->perpage, write);
                dumpvp_write(&csize, sizeof (csize));
                dumpvp_write(hp->lzbuf, csize);
                HRSTOP(hp->perpage, write);
        }
        CQ_PUT(mainq, hp->cpin, CBUF_USEDMAP);
        hp->cpin = NULL;
}

/*
 * Main task to dump pages. This is called on the dump CPU.
 */
static void
dumpsys_main_task(void *arg)
{
        dumpsync_t *ds = arg;
        pgcnt_t pagenum = 0, bitnum = 0, hibitnum;
        dumpmlw_t mlw;
```



```
        cbuf_t *cp;
        pgcnt_t baseoff, pfnoff;
        pfn_t base, pfn;
        int sec, i, dumpserial;

        /*
         * Fall back to serial mode if there are no helpers.
         * dump_plat_mincpu can be set to 0 at any time.
         * dumpcfg.helpermap must contain at least one member.
         */
        dumpserial = 1;

        if (dump_plat_mincpu != 0 && dumpcfg.clevel != 0) {
                for (i = 0; i < BT_BITOUL(NCPU); ++i) {
                        if (dumpcfg.helpermap[i] != 0) {
                                dumpserial = 0;
                                break;
                        }
                }
        }

        if (dumpserial) {
                dumpcfg.clevel = 0;
                if (dumpcfg.helper[0].lzbuf == NULL)
                        dumpcfg.helper[0].lzbuf = dumpcfg.helper[1].page;
        }

        dump_init_memlist_walker(&mlw);

        /* CONSTCOND */
        while (1) {

                if (ds->percent > ds->percent_done) {
                        ds->percent_done = ds->percent;
                        sec = (gethrtime() - ds->start) / 1000 / 1000 / 1000;
                        uprintf("^\r%2d:%02d %3d%% done",
                            sec / 60, sec % 60, ds->percent);
                        ds->neednl = 1;
                }

                while (CQ_IS_EMPTY(mainq) && !CQ_IS_EMPTY(writerq)) {

                        /* the writerq never blocks */
                        cp = CQ_GET(writerq);
                        if (cp == NULL)
                                break;

                        dump_timeleft = dump_timeout;

                        HRSTART(ds->perpage, write);
                        dumpvp_write(cp->buf, cp->used);
                        HRSTOP(ds->perpage, write);

                        CQ_PUT(freebufq, cp, CBUF_FREEBUF);
                }

                /*
                 * Wait here for some buffers to process. Returns NULL
                 * when all helpers have terminated and all buffers
                 * have been processed.
                 */
                cp = CQ_GET(mainq);

                if (cp == NULL) {

                        /* Drain the write queue. */
```

```
2351                    if (!CQ_IS_EMPTY(writerq))
2352                            continue;
2353
2354                    /* Main task exits here. */
2355                    break;
2356            }
2357
2358            dump_timeleft = dump_timeout;
2359
2360            switch (cp->state) {
2361
2362            case CBUF_FREEMAP:
2363
2364                    /*
2365                     * Note that we drop CBUF_FREEMAP buffers on
2366                     * the floor (they will not be on any cqueue)
2367                     * when we no longer need them.
2368                     */
2369                    if (bitnum >= dumpcfg.bitmapsize)
2370                            break;
2371
2372                    if (dump_ioerr) {
2373                            bitnum = dumpcfg.bitmapsize;
2374                            CQ_CLOSE(helperq);
2375                            break;
2376                    }
2377
2378                    HRSTART(ds->perpage, bitmap);
2379                    for (; bitnum < dumpcfg.bitmapsize; bitnum++)
2380                            if (BT_TEST(dumpcfg.bitmap, bitnum))
2381                                    break;
2382                    HRSTOP(ds->perpage, bitmap);
2383                    dump_timeleft = dump_timeout;
2384
2385                    if (bitnum >= dumpcfg.bitmapsize) {
2386                            CQ_CLOSE(helperq);
2387                            break;
2388                    }
2389
2390                    /*
2391                     * Try to map CBUF_MAPSIZE ranges. Can't
2392                     * assume that memory segment size is a
2393                     * multiple of CBUF_MAPSIZE. Can't assume that
2394                     * the segment starts on a CBUF_MAPSIZE
2395                     * boundary.
2396                     */
2397                    pfn = dump_bitnum_to_pfn(bitnum, &mlw);
2398                    ASSERT(pfn != PFN_INVALID);
2399                    ASSERT(bitnum + mlw.mpleft <= dumpcfg.bitmapsize);
2400
2401                    base = P2ALIGN(pfn, CBUF_MAPNP);
2402                    if (base < mlw.mpaddr) {
2403                            base = mlw.mpaddr;
2404                            baseoff = P2PHASE(base, CBUF_MAPNP);
2405                    } else {
2406                            baseoff = 0;
2407                    }
2408
2409                    pfnoff = pfn - base;
2410                    if (pfnoff + mlw.mpleft < CBUF_MAPNP) {
2411                            hibitnum = bitnum + mlw.mpleft;
2412                            cp->size = ptob(pfnoff + mlw.mpleft);
2413                    } else {
2414                            hibitnum = bitnum - pfnoff + CBUF_MAPNP -
2415                                baseoff;
2416                            cp->size = CBUF_MAPSIZE - ptob(baseoff);
```



```
2417                    }
2418
2419                    cp->pfn = pfn;
2420                    cp->bitnum = bitnum++;
2421                    cp->pagenum = pagenum++;
2422                    cp->off = ptob(pfnoff);
2423
2424                    for (; bitnum < hibitnum; bitnum++)
2425                            if (BT_TEST(dumpcfg.bitmap, bitnum))
2426                                    pagenum++;
2427
2428                    dump_timeleft = dump_timeout;
2429                    cp->used = ptob(pagenum - cp->pagenum);
2430
2431                    HRSTART(ds->perpage, map);
2432                    hat_devload(kas.a_hat, cp->buf, cp->size, base,
2433                        PROT_READ, HAT_LOAD_NOCONSIST);
2434                    HRSTOP(ds->perpage, map);
2435
2436                    ds->pages_mapped += btop(cp->size);
2437                    ds->pages_used += pagenum - cp->pagenum;
2438
2439                    CQ_OPEN(mainq);
2440
2441                    /*
2442                     * If there are no helpers the main task does
2443                     * non-streams lzjb compress.
2444                     */
2445                    if (dumpserial) {
2446                            dumpsys_lzjb_page(dumpcfg.helper, cp);
2447                            break;
2448                    }
2449
2450                    /* pass mapped pages to a helper */
2451                    CQ_PUT(helperq, cp, CBUF_INREADY);
2452
2453                    /* the last page was done */
2454                    if (bitnum >= dumpcfg.bitmapsize)
2455                            CQ_CLOSE(helperq);
2456
2457                    break;
2458
2459            case CBUF_USEDMAP:
2460
2461                    ds->npages += btop(cp->used);
2462
2463                    HRSTART(ds->perpage, unmap);
2464                    hat_unload(kas.a_hat, cp->buf, cp->size, HAT_UNLOAD);
2465                    HRSTOP(ds->perpage, unmap);
2466
2467                    if (bitnum < dumpcfg.bitmapsize)
2468                            CQ_PUT(mainq, cp, CBUF_FREEMAP);
2469                    CQ_CLOSE(mainq);
2470
2471                    ASSERT(ds->npages <= dumphdr->dump_npages);
2472                    ds->percent = ds->npages * 100LL / dumphdr->dump_npages;
2473                    break;
2474
2475            case CBUF_WRITE:
2476
2477                    CQ_PUT(writerq, cp, CBUF_WRITE);
2478                    break;
2479
2480            case CBUF_ERRMSG:
2481
2482                    if (cp->used > 0) {
```

```
2483                                cp->buf[cp->size - 2] = '\n';
2484                                cp->buf[cp->size - 1] = '\0';
2485                                if (ds->neednl) {
2486                                        uprintf("\n%s", cp->buf);
2487                                        ds->neednl = 0;
2488                                } else {
2489                                        uprintf("%s", cp->buf);
2490                                }
2491                                /* wait for console output */
2492                                drv_usecwait(200000);
2493                                dump_timeleft = dump_timeout;
2494                        }
2495                        CQ_PUT(freebufq, cp, CBUF_FREEBUF);
2496                        break;

2498                default:
2499                        uprintf("dump: unexpected buffer state %d, "
2500                            "buffer will be lost\n", cp->state);
2501                        break;

2503                } /* end switch */

2505        } /* end while(1) */
2506 }

 887 #ifdef  COLLECT_METRICS
 888 size_t
 889 dumpsys_metrics(dumpsync_t *ds, char *buf, size_t size)
 890 {
 891         dumpcfg_t *cfg = &dumpcfg;
 892         int myid = CPU->cpu_seqid;
 893         int i, compress_ratio;
 894         int sec, iorate;
2516         helper_t *hp, *hpend = &cfg->helper[cfg->nhelper];
 895         char *e = buf + size;
 896         char *p = buf;

 898         sec = ds->elapsed / (1000 * 1000 * 1000ULL);
 899         if (sec < 1)
 900                 sec = 1;

 902         if (ds->iotime < 1)
 903                 ds->iotime = 1;
 904         iorate = (ds->nwrite * 100000ULL) / ds->iotime;

 906         compress_ratio = 100LL * ds->npages / btopr(ds->nwrite + 1);

 908 #define P(...) (p += p < e ? snprintf(p, e - p, __VA_ARGS__) : 0)

 910         P("Master cpu_seqid,%d\n", CPU->cpu_seqid);
 911         P("Master cpu_id,%d\n", CPU->cpu_id);
 912         P("dump_flags,0x%x\n", dumphdr->dump_flags);
 913         P("dump_ioerr,%d\n", dump_ioerr);

 915         P("Compression type,serial lzjb\n");
2537         P("Helpers:\n");
2538         for (i = 0; i < ncpus; i++) {
2539                 if ((i & 15) == 0)
2540                         P(",,%03d,", i);
2541                 if (i == myid)
2542                         P("   M");
2543                 else if (BT_TEST(cfg->helpermap, i))
2544                         P("%4d", cpu_seq[i]->cpu_id);
2545                 else
2546                         P("   *");
2547                 if ((i & 15) == 15)
```



```
2548                        P("\n");
2549        }

2551        P("ncbuf_used,%d\n", cfg->ncbuf_used);
2552        P("ncmap,%d\n", cfg->ncmap);

2554        P("Found %ldM ranges,%ld\n", (CBUF_MAPSIZE / DUMP_1MB), cfg->found4m);
2555        P("Found small pages,%ld\n", cfg->foundsm);

2557        P("Compression level,%d\n", cfg->clevel);
2558        P("Compression type,%s %s\n", cfg->clevel == 0 ? "serial" : "parallel",
2559            cfg->clevel >= DUMP_CLEVEL_BZIP2 ? "bzip2" : "lzjb");
 916        P("Compression ratio,%d.%02d\n", compress_ratio / 100, compress_ratio %
 917            100);
2562        P("nhelper_used,%d\n", cfg->nhelper_used);

 919        P("Dump I/O rate MBS,%d.%02d\n", iorate / 100, iorate % 100);
 920        P("..total bytes,%lld\n", (u_longlong_t)ds->nwrite);
 921        P("..total nsec,%lld\n", (u_longlong_t)ds->iotime);
 922        P("dumpbuf.iosize,%ld\n", dumpbuf.iosize);
 923        P("dumpbuf.size,%ld\n", dumpbuf.size);

 925        P("Dump pages/sec,%llu\n", (u_longlong_t)ds->npages / sec);
 926        P("Dump pages,%llu\n", (u_longlong_t)ds->npages);
 927        P("Dump time,%d\n", sec);

 929        if (ds->pages_mapped > 0)
 930                P("per-cent map utilization,%d\n", (int)((100 * ds->pages_used)
 931                    / ds->pages_mapped));

 933        P("\nPer-page metrics:\n");
 934        if (ds->npages > 0) {
 935 #define PERPAGE(x)      ds->perpage.x += cfg->perpage.x;
2580                for (hp = cfg->helper; hp != hpend; hp++) {
2581 #define PERPAGE(x)      ds->perpage.x += hp->perpage.x;
 936                PERPAGES;
 937 #undef PERPAGE
2584                }
 938 #define PERPAGE(x) \
 939                P("%s nsec/page,%d\n", #x, (int)(ds->perpage.x / ds->npages));
 940                PERPAGES;
 941 #undef PERPAGE
2589                P("freebufq.empty,%d\n", (int)(ds->freebufq.empty /
2590                    ds->npages));
2591                P("helperq.empty,%d\n", (int)(ds->helperq.empty /
2592                    ds->npages));
2593                P("writerq.empty,%d\n", (int)(ds->writerq.empty /
2594                    ds->npages));
2595                P("mainq.empty,%d\n", (int)(ds->mainq.empty / ds->npages));

 943                P("I/O wait nsec/page,%llu\n", (u_longlong_t)(ds->iowait /
 944                    ds->npages));
 945        }
 946 #undef P
 947        if (p < e)
 948                bzero(p, e - p);
 949        return (p - buf);
 950 }
 951 #endif  /* COLLECT_METRICS */

 953 /*
 954  * Dump the system.
 955  */
 956 void
 957 dumpsys(void)
 958 {
```

```
 959         dumpsync_t *ds = &dumpsync;
2614         taskq_t *livetaskq = NULL;
 960         pfn_t pfn;
 961         pgcnt_t bitnum;
 962         proc_t *p;
2618         helper_t *hp, *hpend = &dumpcfg.helper[dumpcfg.nhelper];
2619         cbuf_t *cp;
 963         pid_t npids, pidx;
 964         char *content;
 965         char *buf;
 966         size_t size;
2624         int save_dump_clevel;
 967         dumpmlw_t mlw;
 968         dumpcsize_t datatag;
 969         dumpdatahdr_t datahdr;

 971         if (dumpvp == NULL || dumphdr == NULL) {
 972                 uprintf("skipping system dump - no dump device configured\n");
2631                 if (panicstr) {
2632                         dumpcfg.helpers_wanted = 0;
2633                         dumpsys_spinunlock(&dumpcfg.helper_lock);
2634                 }
 973                 return;
 974         }
 975         dumpbuf.cur = dumpbuf.start;

 977         /* clear the sync variables */
2640         ASSERT(dumpcfg.nhelper > 0);
 978         bzero(ds, sizeof (*ds));
2642         ds->dumpcpu = CPU->cpu_id;

 980         /*
 981          * Calculate the starting block for dump.  If we're dumping on a
 982          * swap device, start 1/5 of the way in; otherwise, start at the
 983          * beginning.  And never use the first page -- it may be a disk label.
 984          */
 985         if (dumpvp->v_flag & VISSWAP)
 986                 dumphdr->dump_start = P2ROUNDUP(dumpvp_size / 5, DUMP_OFFSET);
 987         else
 988                 dumphdr->dump_start = DUMP_OFFSET;

 990         dumphdr->dump_flags = DF_VALID | DF_COMPLETE | DF_LIVE | DF_COMPRESSED;
 991         dumphdr->dump_crashtime = gethrestime_sec();
 992         dumphdr->dump_npages = 0;
 993         dumphdr->dump_nvtop = 0;
 994         bzero(dumpcfg.bitmap, BT_SIZEOFMAP(dumpcfg.bitmapsize));
 995         dump_timeleft = dump_timeout;

 997         if (panicstr) {
 998                 dumphdr->dump_flags &= ~DF_LIVE;
 999                 (void) VOP_DUMPCTL(dumpvp, DUMP_FREE, NULL, NULL);
1000                 (void) VOP_DUMPCTL(dumpvp, DUMP_ALLOC, NULL, NULL);
1001                 (void) vsnprintf(dumphdr->dump_panicstring, DUMP_PANICSIZE,
1002                     panicstr, panicargs);

1004         }

1006         if (dump_conflags & DUMP_ALL)
1007                 content = "all";
1008         else if (dump_conflags & DUMP_CURPROC)
1009                 content = "kernel + curproc";
1010         else
1011                 content = "kernel";
1012         uprintf("dumping to %s, offset %lld, content: %s\n", dumppath,
1013             dumphdr->dump_start, content);
```



```
1015         /* Make sure nodename is current */
1016         bcopy(utsname.nodename, dumphdr->dump_utsname.nodename, SYS_NMLN);

1018         /*
1019          * If this is a live dump, try to open a VCHR vnode for better
1020          * performance. We must take care to flush the buffer cache
1021          * first.
1022          */
1023         if (!panicstr) {
1024                 vnode_t *cdev_vp, *cmn_cdev_vp;

1026                 ASSERT(dumpbuf.cdev_vp == NULL);
1027                 cdev_vp = makespecvp(VTOS(dumpvp)->s_dev, VCHR);
1028                 if (cdev_vp != NULL) {
1029                         cmn_cdev_vp = common_specvp(cdev_vp);
1030                         if (VOP_OPEN(&cmn_cdev_vp, FREAD | FWRITE, kcred, NULL)
1031                             == 0) {
1032                                 if (vn_has_cached_data(dumpvp))
1033                                         (void) pvn_vplist_dirty(dumpvp, 0, NULL,
1034                                             B_INVAL | B_TRUNC, kcred);
1035                                 dumpbuf.cdev_vp = cmn_cdev_vp;
1036                         } else {
1037                                 VN_RELE(cdev_vp);
1038                         }
1039                 }
1040         }

1042         /*
1043          * Store a hires timestamp so we can look it up during debugging.
1044          */
1045         lbolt_debug_entry();

1047         /*
1048          * Leave room for the message and ereport save areas and terminal dump
1049          * header.
1050          */
1051         dumpbuf.vp_limit = dumpvp_size - DUMP_LOGSIZE - DUMP_OFFSET -
1052             DUMP_ERPTSIZE;

1054         /*
1055          * Write out the symbol table.  It's no longer compressed,
1056          * so its 'size' and 'csize' are equal.
1057          */
1058         dumpbuf.vp_off = dumphdr->dump_ksyms = dumphdr->dump_start + PAGESIZE;
1059         dumphdr->dump_ksyms_size = dumphdr->dump_ksyms_csize =
1060             ksyms_snapshot(dumpvp_ksyms_write, NULL, LONG_MAX);

1062         /*
1063          * Write out the translation map.
1064          */
1065         dumphdr->dump_map = dumpvp_flush();
1066         dump_as(&kas);
1067         dumphdr->dump_nvtop += dump_plat_addr();

1069         /*
1070          * call into hat, which may have unmapped pages that also need to
1071          * be in the dump
1072          */
1073         hat_dump();

1075         if (dump_conflags & DUMP_ALL) {
1076                 mutex_enter(&pidlock);

1078                 for (npids = 0, p = practive; p != NULL; p = p->p_next)
1079                         dumpcfg.pids[npids++] = p->p_pid;
```

```
1081                mutex_exit(&pidlock);

1083                for (pidx = 0; pidx < npids; pidx++)
1084                        (void) dump_process(dumpcfg.pids[pidx]);

1086                dump_init_memlist_walker(&mlw);
1087                for (bitnum = 0; bitnum < dumpcfg.bitmapsize; bitnum++) {
1088                        dump_timeleft = dump_timeout;
1089                        pfn = dump_bitnum_to_pfn(bitnum, &mlw);
1090                        /*
1091                         * Some hypervisors do not have all pages available to
1092                         * be accessed by the guest OS.  Check for page
1093                         * accessibility.
1094                         */
1095                        if (plat_hold_page(pfn, PLAT_HOLD_NO_LOCK, NULL) !=
1096                            PLAT_HOLD_OK)
1097                                continue;
1098                        BT_SET(dumpcfg.bitmap, bitnum);
1099                }
1100                dumphdr->dump_npages = dumpcfg.bitmapsize;
1101                dumphdr->dump_flags |= DF_ALL;

1103        } else if (dump_conflags & DUMP_CURPROC) {
1104                /*
1105                 * Determine which pid is to be dumped.  If we're panicking, we
1106                 * dump the process associated with panic_thread (if any).  If
1107                 * this is a live dump, we dump the process associated with
1108                 * curthread.
1109                 */
1110                npids = 0;
1111                if (panicstr) {
1112                        if (panic_thread != NULL &&
1113                            panic_thread->t_procp != NULL &&
1114                            panic_thread->t_procp != &p0) {
1115                                dumpcfg.pids[npids++] =
1116                                    panic_thread->t_procp->p_pid;
1117                        }
1118                } else {
1119                        dumpcfg.pids[npids++] = curthread->t_procp->p_pid;
1120                }

1122                if (npids && dump_process(dumpcfg.pids[0]) == 0)
1123                        dumphdr->dump_flags |= DF_CURPROC;
1124                else
1125                        dumphdr->dump_flags |= DF_KERNEL;

1127        } else {
1128                dumphdr->dump_flags |= DF_KERNEL;
1129        }

1131        dumphdr->dump_hashmask = (1 << highbit(dumphdr->dump_nvtop - 1)) - 1;

1133        /*
1134         * Write out the pfn table.
1135         */
1136        dumphdr->dump_pfn = dumpvp_flush();
1137        dump_init_memlist_walker(&mlw);
1138        for (bitnum = 0; bitnum < dumpcfg.bitmapsize; bitnum++) {
1139                dump_timeleft = dump_timeout;
1140                if (!BT_TEST(dumpcfg.bitmap, bitnum))
1141                        continue;
1142                pfn = dump_bitnum_to_pfn(bitnum, &mlw);
1143                ASSERT(pfn != PFN_INVALID);
1144                dumpvp_write(&pfn, sizeof (pfn_t));
1145        }
1146        dump_plat_pfn();
```



```
1148        /*
1149         * Write out all the pages.
1150         * Map pages, copy them handling UEs, compress, and write them out.
2815         * Cooperate with any helpers running on CPUs in panic_idle().
1151         */
1152        dumphdr->dump_data = dumpvp_flush();

1154        ASSERT(dumpcfg.page);
1155        bzero(&dumpcfg.perpage, sizeof (dumpcfg.perpage));
2819        bzero(dumpcfg.helpermap, BT_SIZEOFMAP(NCPU));
2820        ds->live = dumpcfg.clevel > 0 &&
2821            (dumphdr->dump_flags & DF_LIVE) != 0;

1157        ds->start = gethrtime();
1158        ds->iowaitts = ds->start;

2823        save_dump_clevel = dumpcfg.clevel;
1160        if (panicstr)
1161                kmem_dump_begin();
2825                dumpsys_get_maxmem();
2826        else if (dumpcfg.clevel >= DUMP_CLEVEL_BZIP2)
2827                dumpcfg.clevel = DUMP_CLEVEL_LZJB;

1163        dump_init_memlist_walker(&mlw);
1164        for (bitnum = 0; bitnum < dumpcfg.bitmapsize; bitnum++) {
1165                size_t csize;

1167                dump_timeleft = dump_timeout;
1168                HRSTART(ds->perpage, bitmap);
1169                if (!BT_TEST(dumpcfg.bitmap, bitnum)) {
1170                        HRSTOP(ds->perpage, bitmap);
2829        dumpcfg.nhelper_used = 0;
2830        for (hp = dumpcfg.helper; hp != hpend; hp++) {
2831                if (hp->page == NULL) {
2832                        hp->helper = DONEHELPER;
1171                        continue;
1172                }
1173                HRSTOP(ds->perpage, bitmap);
2835                ++dumpcfg.nhelper_used;
2836                hp->helper = FREEHELPER;
2837                hp->taskqid = NULL;
2838                hp->ds = ds;
2839                bzero(&hp->perpage, sizeof (hp->perpage));
2840                if (dumpcfg.clevel >= DUMP_CLEVEL_BZIP2)
2841                        (void) BZ2_bzCompressReset(&hp->bzstream);
2842        }

1175                pfn = dump_bitnum_to_pfn(bitnum, &mlw);
1176                ASSERT(pfn != PFN_INVALID);
2844        CQ_OPEN(freebufq);
2845        CQ_OPEN(helperq);

1178                HRSTART(ds->perpage, map);
1179                hat_devload(kas.a_hat, dumpcfg.cmap, PAGESIZE, pfn, PROT_READ,
1180                            HAT_LOAD_NOCONSIST);
1181                HRSTOP(ds->perpage, map);
2847        dumpcfg.ncbuf_used = 0;
2848        for (cp = dumpcfg.cbuf; cp != &dumpcfg.cbuf[dumpcfg.ncbuf]; cp++) {
2849                if (cp->buf != NULL) {
2850                        CQ_PUT(freebufq, cp, CBUF_FREEBUF);
2851                        ++dumpcfg.ncbuf_used;
2852                }
2853        }

1183                dump_pagecopy(dumpcfg.cmap, dumpcfg.page);
```

```
2855         for (cp = dumpcfg.cmap; cp != &dumpcfg.cmap[dumpcfg.ncmap]; cp++)
2856                 CQ_PUT(mainq, cp, CBUF_FREEMAP);

1185                 HRSTART(ds->perpage, unmap);
1186                 hat_unload(kas.a_hat, dumpcfg.cmap, PAGESIZE, HAT_UNLOAD);
1187                 HRSTOP(ds->perpage, unmap);
2858         ds->start = gethrtime();
2859         ds->iowaitts = ds->start;

1189                 HRSTART(dumpcfg.perpage, compress);
1190                 csize = compress(dumpcfg.page, dumpcfg.lzbuf, PAGESIZE);
1191                 HRSTOP(dumpcfg.perpage, compress);
2861         /* start helpers */
2862         if (ds->live) {
2863                 int n = dumpcfg.nhelper_used;
2864                 int pri = MINCLSYSPRI - 25;

1193                 HRSTART(dumpcfg.perpage, write);
1194                 dumpvp_write(&csize, sizeof (csize));
1195                 dumpvp_write(dumpcfg.lzbuf, csize);
1196                 HRSTOP(dumpcfg.perpage, write);

1198                 if (dump_ioerr) {
1199                         dumphdr->dump_flags &= ~DF_COMPLETE;
1200                         dumphdr->dump_npages = ds->npages;
1201                         break;
2866                 livetaskq = taskq_create("LiveDump", n, pri, n, n,
2867                     TASKQ_PREPOPULATE);
2868                 for (hp = dumpcfg.helper; hp != hpend; hp++) {
2869                         if (hp->page == NULL)
2870                                 continue;
2871                         hp->helper = hp - dumpcfg.helper;
2872                         hp->taskqid = taskq_dispatch(livetaskq,
2873                             dumpsys_live_helper, (void *)hp, TQ_NOSLEEP);
1202                 }
1203                 if (++ds->npages * 100LL / dumphdr->dump_npages > ds->percent_do
1204                         int sec;

1206                         sec = (gethrtime() - ds->start) / 1000 / 1000 / 1000;
1207                         uprintf("^\r%2d:%02d %3d%% done", sec / 60, sec % 60,
1208                                 ++ds->percent_done);
1209                         if (!panicstr)
1210                                 delay(1);       /* let the output be sent */
2876         } else {
2877                 if (panicstr)
2878                         kmem_dump_begin();
2879                 dumpcfg.helpers_wanted = dumpcfg.clevel > 0;
2880                 dumpsys_spinunlock(&dumpcfg.helper_lock);
2881                 }
1212         }

2883         /* run main task */
2884         dumpsys_main_task(ds);

1214         ds->elapsed = gethrtime() - ds->start;
1215         if (ds->elapsed < 1)
1216                 ds->elapsed = 1;

2890         if (livetaskq != NULL)
2891                 taskq_destroy(livetaskq);

2893         if (ds->neednl) {
2894                 uprintf("\n");
2895                 ds->neednl = 0;
2896         }
```



```
1218         /* record actual pages dumped */
1219         dumphdr->dump_npages = ds->npages;

1221         /* platform-specific data */
1222         dumphdr->dump_npages += dump_plat_data(dumpcfg.page);
2902         dumphdr->dump_npages += dump_plat_data(dumpcfg.cbuf[0].buf);

1224         /* note any errors by clearing DF_COMPLETE */
1225         if (dump_ioerr || ds->npages < dumphdr->dump_npages)
1226                 dumphdr->dump_flags &= ~DF_COMPLETE;

1228         /* end of stream blocks */
1229         datatag = 0;
1230         dumpvp_write(&datatag, sizeof (datatag));

1232         bzero(&datahdr, sizeof (datahdr));

1234         /* buffer for metrics */
1235         buf = dumpcfg.page;
1236         size = MIN(PAGESIZE, DUMP_OFFSET - sizeof (dumphdr_t) -
2915         buf = dumpcfg.cbuf[0].buf;
2916         size = MIN(dumpcfg.cbuf[0].size, DUMP_OFFSET - sizeof (dumphdr_t) -
1237             sizeof (dumpdatahdr_t));

1239         /* finish the kmem intercepts, collect kmem verbose info */
1240         if (panicstr) {
1241                 datahdr.dump_metrics = kmem_dump_finish(buf, size);
1242                 buf += datahdr.dump_metrics;
1243                 size -= datahdr.dump_metrics;
1244         }

1246         /* record in the header whether this is a fault-management panic */
1247         if (panicstr)
1248                 dumphdr->dump_fm_panic = is_fm_panic();

1250         /* compression info in data header */
1251         datahdr.dump_datahdr_magic = DUMP_DATAHDR_MAGIC;
1252         datahdr.dump_datahdr_version = DUMP_DATAHDR_VERSION;
1253         datahdr.dump_maxcsize = PAGESIZE;
1254         datahdr.dump_maxrange = 1;
1255         datahdr.dump_nstreams = 1;
1256         datahdr.dump_clevel = 0;
2933         datahdr.dump_maxcsize = CBUF_SIZE;
2934         datahdr.dump_maxrange = CBUF_MAPSIZE / PAGESIZE;
2935         datahdr.dump_nstreams = dumpcfg.nhelper_used;
2936         datahdr.dump_clevel = dumpcfg.clevel;
1257 #ifdef COLLECT_METRICS
1258         if (dump_metrics_on)
1259                 datahdr.dump_metrics += dumpsys_metrics(ds, buf, size);
1260 #endif
1261         datahdr.dump_data_csize = dumpvp_flush() - dumphdr->dump_data;

1263         /*
1264          * Write out the initial and terminal dump headers.
1265          */
1266         dumpbuf.vp_off = dumphdr->dump_start;
1267         dumpvp_write(dumphdr, sizeof (dumphdr_t));
1268         (void) dumpvp_flush();

1270         dumpbuf.vp_limit = dumpvp_size;
1271         dumpbuf.vp_off = dumpbuf.vp_limit - DUMP_OFFSET;
1272         dumpvp_write(dumphdr, sizeof (dumphdr_t));
1273         dumpvp_write(&datahdr, sizeof (dumpdatahdr_t));
1274         dumpvp_write(dumpcfg.page, datahdr.dump_metrics);
2954         dumpvp_write(dumpcfg.cbuf[0].buf, datahdr.dump_metrics);
```

```
1276         (void) dumpvp_flush();

1278         uprintf("\r%3d%% done: %llu pages dumped, ",
1279             ds->percent_done, (u_longlong_t)ds->npages);

1281         if (dump_ioerr == 0) {
1282                 uprintf("dump succeeded\n");
1283         } else {
1284                 uprintf("dump failed: error %d\n", dump_ioerr);
1285 #ifdef DEBUG
1286                 if (panicstr)
1287                         debug_enter("dump failed");
1288 #endif
1289         }

1291         /*
1292          * Write out all undelivered messages.  This has to be the *last*
1293          * thing we do because the dump process itself emits messages.
1294          */
1295         if (panicstr) {
1296                 dump_summary();
1297                 dump_ereports();
1298                 dump_messages();
1299         }

1301         delay(2 * hz);  /* let people see the 'done' message */
1302         dump_timeleft = 0;
1303         dump_ioerr = 0;

1305         /* restore settings after live dump completes */
1306         if (!panicstr) {
2987                 dumpcfg.clevel = save_dump_clevel;

1307                 /* release any VCHR open of the dump device */
1308                 if (dumpbuf.cdev_vp != NULL) {
1309                         (void) VOP_CLOSE(dumpbuf.cdev_vp, FREAD | FWRITE, 1, 0,
1310                             kcred, NULL);
1311                         VN_RELE(dumpbuf.cdev_vp);
1312                         dumpbuf.cdev_vp = NULL;
1313                 }
1314         }
1315 }
```

_____***unchanged_portion_omitted_***

****
7653 Fri Oct 26 17:54:27 2012
new/usr/src/uts/common/sys/dumphdr.h
[mq]: core-v2
****
_____unchanged_portion_omitted_

```
 163 #define DUMP_DATAHDR_MAGIC      ('d' << 24 | 'h' << 16 | 'd' << 8 | 'r')

 165 #define DUMP_DATAHDR_VERSION    1
 166 #define DUMP_CLEVEL_LZJB        1       /* parallel lzjb compression */
 167 #define DUMP_CLEVEL_BZIP2       2       /* parallel bzip2 level 1 */

 169 #ifdef _KERNEL

 171 extern kmutex_t dump_lock;
 172 extern struct vnode *dumpvp;
 173 extern u_offset_t dumpvp_size;
 174 extern struct dumphdr *dumphdr;
 175 extern int dump_conflags;
 176 extern char *dumppath;

 178 extern int dump_timeout;
 179 extern int dump_timeleft;
 180 extern int dump_ioerr;
 181 extern int sync_timeout;
 182 extern int sync_timeleft;

 184 extern int dumpinit(struct vnode *, char *, int);
 185 extern void dumpfini(void);
 186 extern void dump_resize(void);
 187 extern void dump_page(pfn_t);
 188 extern void dump_addpage(struct as *, void *, pfn_t);
 189 extern void dumpsys(void);
 190 extern void dumpsys_helper(void);
 191 extern void dumpsys_helper_nw(void);
 190 extern void dump_messages(void);
 191 extern void dump_ereports(void);
 192 extern void dumpvp_write(const void *, size_t);
 193 extern int dumpvp_resize(void);
 194 extern int dump_plat_addr(void);
 195 extern void dump_plat_pfn(void);
 196 extern int dump_plat_data(void *);
 197 extern int dump_set_uuid(const char *);
 198 extern const char *dump_get_uuid(void);

 200 /*
 201  * Define a CPU count threshold that determines when to employ
 202  * bzip2. This value is defined per-platform.
 203  */
 204 extern uint_t dump_plat_mincpu_default;

 206 #define DUMP_PLAT_SUN4U_MINCPU          51
 207 #define DUMP_PLAT_SUN4U_OPL_MINCPU      8
 208 #define DUMP_PLAT_SUN4V_MINCPU          128
 209 #define DUMP_PLAT_X86_64_MINCPU         11
 210 #define DUMP_PLAT_X86_32_MINCPU         0

 212 /*
 213  * Override the per-platform default by setting this variable with
 214  * /etc/system.  The value 0 disables parallelism, and the old format
 215  * dump is produced.
 216  */
 217 extern uint_t dump_plat_mincpu;

 221 /*
```



```
 222  * Pages may be stolen at dump time. Prevent the pages from ever being
 223  * allocated while dump is running.
 224  */
 225 #define IS_DUMP_PAGE(pp) (dump_check_used && dump_test_used((pp)->p_pagenum))

 227 extern int dump_test_used(pfn_t);
 228 extern int dump_check_used;

 219 #endif /* _KERNEL */

 221 #ifdef  __cplusplus
 222 }
```

_____unchanged_portion_omitted_

```
**********************************************************
  116337 Fri Oct 26 17:54:27 2012
new/usr/src/uts/common/vm/vm_pagelist.c
[mq]: core-v2
**********************************************************
______unchanged_portion_omitted_
```

```
2917 page_t *
2918 page_get_mnode_freelist(int mnode, uint_t bin, int mtype, uchar_t szc,
2919     uint_t flags)
2920 {
2921         kmutex_t                *pcm;
2922         page_t                  *pp, *first_pp;
2923         uint_t                  sbin;
2924         int                     plw_initialized;
2925         page_list_walker_t      plw;

2927         ASSERT(szc < mmu_page_sizes);

2929         VM_STAT_ADD(vmm_vmstats.pgmf_alloc[szc]);

2931         MTYPE_START(mnode, mtype, flags);
2932         if (mtype < 0) {        /* mnode does not have memory in mtype range */
2933                 VM_STAT_ADD(vmm_vmstats.pgmf_allocempty[szc]);
2934                 return (NULL);
2935         }
2936 try_again:

2938         plw_initialized = 0;
2939         plw.plw_ceq_dif = 1;

2941         /*
2942          * Only hold one freelist lock at a time, that way we
2943          * can start anywhere and not have to worry about lock
2944          * ordering.
2945          */
2946         for (plw.plw_count = 0;
2947             plw.plw_count < plw.plw_ceq_dif; plw.plw_count++) {
2948                 sbin = bin;
2949                 do {
2950                         if (!PAGE_FREELISTS(mnode, szc, bin, mtype))
2951                                 goto bin_empty_1;

2953                         pcm = PC_BIN_MUTEX(mnode, bin, PG_FREE_LIST);
2954                         mutex_enter(pcm);
2955                         pp = PAGE_FREELISTS(mnode, szc, bin, mtype);
2956                         if (pp == NULL)
2957                                 goto bin_empty_0;

2959                         /*
2960                          * These were set before the page
2961                          * was put on the free list,
2962                          * they must still be set.
2963                          */
2964                         ASSERT(PP_ISFREE(pp));
2965                         ASSERT(PP_ISAGED(pp));
2966                         ASSERT(pp->p_vnode == NULL);
2967                         ASSERT(pp->p_hash == NULL);
2968                         ASSERT(pp->p_offset == (u_offset_t)-1);
2969                         ASSERT(pp->p_szc == szc);
2970                         ASSERT(PFN_2_MEM_NODE(pp->p_pagenum) == mnode);

2972                         /*
2973                          * Walk down the hash chain.
2974                          * 8k pages are linked on p_next
2975                          * and p_prev fields. Large pages
```



```
2976                          * are a contiguous group of
2977                          * constituent pages linked together
2978                          * on their p_next and p_prev fields.
2979                          * The large pages are linked together
2980                          * on the hash chain using p_vpnext
2981                          * p_vpprev of the base constituent
2982                          * page of each large page.
2983                          */
2984                         first_pp = pp;
2985                         while (!page_trylock_cons(pp, SE_EXCL)) {
2985                         while (IS_DUMP_PAGE(pp) || !page_trylock_cons(pp,
2986                             SE_EXCL)) {
2986                                 if (szc == 0) {
2987                                         pp = pp->p_next;
2988                                 } else {
2989                                         pp = pp->p_vpnext;
2990                                 }

2992                                 ASSERT(PP_ISFREE(pp));
2993                                 ASSERT(PP_ISAGED(pp));
2994                                 ASSERT(pp->p_vnode == NULL);
2995                                 ASSERT(pp->p_hash == NULL);
2996                                 ASSERT(pp->p_offset == (u_offset_t)-1);
2997                                 ASSERT(pp->p_szc == szc);
2998                                 ASSERT(PFN_2_MEM_NODE(pp->p_pagenum) == mnode);

3000                                 if (pp == first_pp)
3001                                         goto bin_empty_0;
3002                         }

3004                         ASSERT(pp != NULL);
3005                         ASSERT(mtype == PP_2_MTYPE(pp));
3006                         ASSERT(pp->p_szc == szc);
3007                         if (szc == 0) {
3008                                 page_sub(&PAGE_FREELISTS(mnode,
3009                                     szc, bin, mtype), pp);
3010                         } else {
3011                                 page_vpsub(&PAGE_FREELISTS(mnode,
3012                                     szc, bin, mtype), pp);
3013                                 CHK_LPG(pp, szc);
3014                         }
3015                         page_ctr_sub(mnode, mtype, pp, PG_FREE_LIST);

3017                         if ((PP_ISFREE(pp) == 0) || (PP_ISAGED(pp) == 0))
3018                                 panic("free page is not. pp %p", (void *)pp);
3019                         mutex_exit(pcm);

3021 #if defined(__sparc)
3022                         ASSERT(!kcage_on || PP_ISNORELOC(pp) ||
3023                             (flags & PG_NORELOC) == 0);

3025                         if (PP_ISNORELOC(pp))
3026                                 kcage_freemem_sub(page_get_pagecnt(szc));
3027 #endif
3028                         VM_STAT_ADD(vmm_vmstats.pgmf_allocok[szc]);
3029                         return (pp);

3031 bin_empty_0:
3032                         mutex_exit(pcm);
3033 bin_empty_1:
3034                         if (plw_initialized == 0) {
3035                                 page_list_walk_init(szc, flags, bin, 1, 1,
3036                                     &plw);
3037                                 plw_initialized = 1;
3038                                 ASSERT(plw.plw_colors <=
3039                                     PAGE_GET_PAGECOLORS(szc));
```

```
3040                                ASSERT(plw.plw_colors > 0);
3041                                ASSERT((plw.plw_colors &
3042                                    (plw.plw_colors - 1)) == 0);
3043                                ASSERT(bin < plw.plw_colors);
3044                                ASSERT(plw.plw_ceq_mask[szc] < plw.plw_colors);
3045                        }
3046                        /* calculate the next bin with equivalent color */
3047                        bin = ADD_MASKED(bin, plw.plw_bin_step,
3048                            plw.plw_ceq_mask[szc], plw.plw_color_mask);
3049                } while (sbin != bin);

3051                /*
3052                 * color bins are all empty if color match. Try and
3053                 * satisfy the request by breaking up or coalescing
3054                 * pages from a different size freelist of the correct
3055                 * color that satisfies the ORIGINAL color requested.
3056                 * If that fails then try pages of the same size but
3057                 * different colors assuming we are not called with
3058                 * PG_MATCH_COLOR.
3059                 */
3060                if (plw.plw_do_split &&
3061                    (pp = page_freelist_split(szc, bin, mnode,
3062                    mtype, PFNNULL, PFNNULL, &plw)) != NULL)
3063                        return (pp);

3065                if (szc > 0 && (pp = page_freelist_coalesce(mnode, szc,
3066                    bin, plw.plw_ceq_mask[szc], mtype, PFNNULL)) !=  NULL)
3067                        return (pp);

3069                if (plw.plw_ceq_dif > 1)
3070                        bin = page_list_walk_next_bin(szc, bin, &plw);
3071        }

3073        /* if allowed, cycle through additional mtypes */
3074        MTYPE_NEXT(mnode, mtype, flags);
3075        if (mtype >= 0)
3076                goto try_again;

3078        VM_STAT_ADD(vmm_vmstats.pgmf_allocfailed[szc]);

3080        return (NULL);
3081 }
```

*_____unchanged_portion_omitted_*

```
3980 page_t *
3981 page_get_mnode_cachelist(uint_t bin, uint_t flags, int mnode, int mtype)
3982 {
3983        kmutex_t                *pcm;
3984        page_t                  *pp, *first_pp;
3985        uint_t                  sbin;
3986        int                     plw_initialized;
3987        page_list_walker_t      plw;

3989        VM_STAT_ADD(vmm_vmstats.pgmc_alloc);

3991        /* LINTED */
3992        MTYPE_START(mnode, mtype, flags);
3993        if (mtype < 0) {        /* mnode does not have memory in mtype range */
3994                VM_STAT_ADD(vmm_vmstats.pgmc_allocempty);
3995                return (NULL);
3996        }

3998 try_again:

4000        plw_initialized = 0;
4001        plw.plw_ceq_dif = 1;
```



```
4003        /*
4004         * Only hold one cachelist lock at a time, that way we
4005         * can start anywhere and not have to worry about lock
4006         * ordering.
4007         */

4009        for (plw.plw_count = 0;
4010            plw.plw_count < plw.plw_ceq_dif; plw.plw_count++) {
4011                sbin = bin;
4012                do {

4014                        if (!PAGE_CACHELISTS(mnode, bin, mtype))
4015                                goto bin_empty_1;
4016                        pcm = PC_BIN_MUTEX(mnode, bin, PG_CACHE_LIST);
4017                        mutex_enter(pcm);
4018                        pp = PAGE_CACHELISTS(mnode, bin, mtype);
4019                        if (pp == NULL)
4020                                goto bin_empty_0;

4022                        first_pp = pp;
4023                        ASSERT(pp->p_vnode);
4024                        ASSERT(PP_ISAGED(pp) == 0);
4025                        ASSERT(pp->p_szc == 0);
4026                        ASSERT(PFN_2_MEM_NODE(pp->p_pagenum) == mnode);
4027                        while (!page_trylock(pp, SE_EXCL)) {
4028                        while (IS_DUMP_PAGE(pp) || !page_trylock(pp, SE_EXCL)) {
4028                                pp = pp->p_next;
4029                                ASSERT(pp->p_szc == 0);
4030                                if (pp == first_pp) {
4031                                        /*
4032                                         * We have searched the complete list!
4033                                         * And all of them (might only be one)
4034                                         * are locked. This can happen since
4035                                         * these pages can also be found via
4036                                         * the hash list. When found via the
4037                                         * hash list, they are locked first,
4038                                         * then removed. We give up to let the
4039                                         * other thread run.
4040                                         */
4041                                        pp = NULL;
4042                                        break;
4043                                }
4044                                ASSERT(pp->p_vnode);
4045                                ASSERT(PP_ISFREE(pp));
4046                                ASSERT(PP_ISAGED(pp) == 0);
4047                                ASSERT(PFN_2_MEM_NODE(pp->p_pagenum) ==
4048                                    mnode);
4049                        }

4051                        if (pp) {
4052                                page_t  **ppp;
4053                                /*
4054                                 * Found and locked a page.
4055                                 * Pull it off the list.
4056                                 */
4057                                ASSERT(mtype == PP_2_MTYPE(pp));
4058                                ppp = &PAGE_CACHELISTS(mnode, bin, mtype);
4059                                page_sub(ppp, pp);
4060                                /*
4061                                 * Subtract counters before releasing pcm mutex
4062                                 * to avoid a race with page_freelist_coalesce
4063                                 * and page_freelist_split.
4064                                 */
4065                                page_ctr_sub(mnode, mtype, pp, PG_CACHE_LIST);
4066                                mutex_exit(pcm);
```

```
4067                                    ASSERT(pp->p_vnode);
4068                                    ASSERT(PP_ISAGED(pp) == 0);
4069 #if defined(__sparc)
4070                                    ASSERT(!kcage_on ||
4071                                        (flags & PG_NORELOC) == 0 ||
4072                                        PP_ISNORELOC(pp));
4073                                    if (PP_ISNORELOC(pp)) {
4074                                            kcage_freemem_sub(1);
4075                                    }
4076 #endif
4077                                    VM_STAT_ADD(vmm_vmstats. pgmc_allocok);
4078                                    return (pp);
4079                            }
4080 bin_empty_0:
4081                            mutex_exit(pcm);
4082 bin_empty_1:
4083                            if (plw_initialized == 0) {
4084                                    page_list_walk_init(0, flags, bin, 0, 1, &plw);
4085                                    plw_initialized = 1;
4086                            }
4087                            /* calculate the next bin with equivalent color */
4088                            bin = ADD_MASKED(bin, plw.plw_bin_step,
4089                                plw.plw_ceq_mask[0], plw.plw_color_mask);
4090                    } while (sbin != bin);

4092                    if (plw.plw_ceq_dif > 1)
4093                            bin = page_list_walk_next_bin(0, bin, &plw);
4094            }

4096            MTYPE_NEXT(mnode, mtype, flags);
4097            if (mtype >= 0)
4098                    goto try_again;

4100            VM_STAT_ADD(vmm_vmstats.pgmc_allocfailed);
4101            return (NULL);
4102 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   34361 Fri Oct 26 17:54:28 2012
new/usr/src/uts/i86pc/os/machdep.c
[mq]: core-v2
**********************************************************
______unchanged_portion_omitted_

 888 /*ARGSUSED*/
 889 void
 890 lwp_stk_fini(klwp_t *lwp)
 891 {}

 893 /*
 894  * If we're not the panic CPU, we wait in panic_idle for reboot.
 895  */
 896 void
 897 panic_idle(void)
 898 {
 899         splx(ipltospl(CLOCK_LEVEL));
 900         (void) setjmp(&curthread->t_pcb);

 902         dumpsys_helper();

 902 #ifndef __xpv
 903         for (;;)
 904                 i86_halt();
 905 #else
 906         for (;;)
 907                 ;
 908 #endif
 909 }
______unchanged_portion_omitted_
```

```
**********************************************************
   97263 Fri Oct 26 17:54:28 2012
new/usr/src/uts/i86pc/vm/vm_machdep.c
[mq]: core-v2
**********************************************************
```

_____*unchanged_portion_omitted_*

```
 978 #if !defined(__xpv)
 979 /*
 980  * is_contigpage_free:
 981  *      returns a page list of contiguous pages. It minimally has to return
 982  *      minctg pages. Caller determines minctg based on the scatter-gather
 983  *      list length.
 984  *
 985  *      pfnp is set to the next page frame to search on return.
 986  */
 987 static page_t *
 988 is_contigpage_free(
 989         pfn_t *pfnp,
 990         pgcnt_t *pgcnt,
 991         pgcnt_t minctg,
 992         uint64_t pfnseg,
 993         int iolock)
 994 {
 995         int     i = 0;
 996         pfn_t   pfn = *pfnp;
 997         page_t  *pp;
 998         page_t  *plist = NULL;

1000         /*
1001          * fail if pfn + minctg crosses a segment boundary.
1002          * Adjust for next starting pfn to begin at segment boundary.
1003          */

1005         if (((*pfnp + minctg - 1) & pfnseg) < (*pfnp & pfnseg)) {
1006                 *pfnp = roundup(*pfnp, pfnseg + 1);
1007                 return (NULL);
1008         }

1010         do {
1011 retry:
1012                 pp = page_numtopp_nolock(pfn + i);
1013                 if ((pp == NULL) || (page_trylock(pp, SE_EXCL) == 0)) {
1013                 if ((pp == NULL) || IS_DUMP_PAGE(pp) ||
1014                     (page_trylock(pp, SE_EXCL) == 0)) {
1014                         (*pfnp)++;
1015                         break;
1016                 }
1017                 if (page_pptonum(pp) != pfn + i) {
1018                         page_unlock(pp);
1019                         goto retry;
1020                 }

1022                 if (!(PP_ISFREE(pp))) {
1023                         page_unlock(pp);
1024                         (*pfnp)++;
1025                         break;
1026                 }

1028                 if (!PP_ISAGED(pp)) {
1029                         page_list_sub(pp, PG_CACHE_LIST);
1030                         page_hashout(pp, (kmutex_t *)NULL);
1031                 } else {
1032                         page_list_sub(pp, PG_FREE_LIST);
1033                 }
```



```
1035                 if (iolock)
1036                         page_io_lock(pp);
1037                 page_list_concat(&plist, &pp);

1039                 /*
1040                  * exit loop when pgcnt satisfied or segment boundary reached.
1041                  */

1043         } while ((++i < *pgcnt) && ((pfn + i) & pfnseg));

1045         *pfnp += i;             /* set to next pfn to search */

1047         if (i >= minctg) {
1048                 *pgcnt -= i;
1049                 return (plist);
1050         }

1052         /*
1053          * failure: minctg not satisfied.
1054          *
1055          * if next request crosses segment boundary, set next pfn
1056          * to search from the segment boundary.
1057          */
1058         if (((*pfnp + minctg - 1) & pfnseg) < (*pfnp & pfnseg))
1059                 *pfnp = roundup(*pfnp, pfnseg + 1);

1061         /* clean up any pages already allocated */

1063         while (plist) {
1064                 pp = plist;
1065                 page_sub(&plist, pp);
1066                 page_list_add(pp, PG_FREE_LIST | PG_LIST_TAIL);
1067                 if (iolock)
1068                         page_io_unlock(pp);
1069                 page_unlock(pp);
1070         }

1072         return (NULL);
1073 }
```

_____*unchanged_portion_omitted_*

```
3062 #else /* !__xpv */

3064 /*
3065  * get a page from any list with the given mnode
3066  */
3067 static page_t *
3068 page_get_mnode_anylist(ulong_t origbin, uchar_t szc, uint_t flags,
3069     int mnode, int mtype, ddi_dma_attr_t *dma_attr)
3070 {
3071         kmutex_t                *pcm;
3072         int                     i;
3073         page_t                  *pp;
3074         page_t                  *first_pp;
3075         uint64_t                pgaddr;
3076         ulong_t                 bin;
3077         int                     mtypestart;
3078         int                     plw_initialized;
3079         page_list_walker_t      plw;

3081         VM_STAT_ADD(pga_vmstats.pgma_alloc);

3083         ASSERT((flags & PG_MATCH_COLOR) == 0);
3084         ASSERT(szc == 0);
3085         ASSERT(dma_attr != NULL);
```

```
3087         MTYPE_START(mnode, mtype, flags);
3088         if (mtype < 0) {
3089                 VM_STAT_ADD(pga_vmstats.pgma_allocempty);
3090                 return (NULL);
3091         }

3093         mtypestart = mtype;

3095         bin = origbin;

3097         /*
3098          * check up to page_colors + 1 bins - origbin may be checked twice
3099          * because of BIN_STEP skip
3100          */
3101         do {
3102                 plw_initialized = 0;

3104                 for (plw.plw_count = 0;
3105                     plw.plw_count < page_colors; plw.plw_count++) {

3107                         if (PAGE_FREELISTS(mnode, szc, bin, mtype) == NULL)
3108                                 goto nextfreebin;

3110                         pcm = PC_BIN_MUTEX(mnode, bin, PG_FREE_LIST);
3111                         mutex_enter(pcm);
3112                         pp = PAGE_FREELISTS(mnode, szc, bin, mtype);
3113                         first_pp = pp;
3114                         while (pp != NULL) {
3115                                 if (page_trylock(pp, SE_EXCL) == 0) {
3116                                 if (IS_DUMP_PAGE(pp) || page_trylock(pp,
3117                                     SE_EXCL) == 0) {
3116                                         pp = pp->p_next;
3117                                         if (pp == first_pp) {
3118                                                 pp = NULL;
3119                                         }
3120                                         continue;
3121                                 }

3123                                 ASSERT(PP_ISFREE(pp));
3124                                 ASSERT(PP_ISAGED(pp));
3125                                 ASSERT(pp->p_vnode == NULL);
3126                                 ASSERT(pp->p_hash == NULL);
3127                                 ASSERT(pp->p_offset == (u_offset_t)-1);
3128                                 ASSERT(pp->p_szc == szc);
3129                                 ASSERT(PFN_2_MEM_NODE(pp->p_pagenum) == mnode);
3130                                 /* check if page within DMA attributes */
3131                                 pgaddr = pa_to_ma(pfn_to_pa(pp->p_pagenum));
3132                                 if ((pgaddr >= dma_attr->dma_attr_addr_lo) &&
3133                                     (pgaddr + MMU_PAGESIZE - 1 <=
3134                                     dma_attr->dma_attr_addr_hi)) {
3135                                         break;
3136                                 }

3138                                 /* continue looking */
3139                                 page_unlock(pp);
3140                                 pp = pp->p_next;
3141                                 if (pp == first_pp)
3142                                         pp = NULL;

3144                         }
3145                         if (pp != NULL) {
3146                                 ASSERT(mtype == PP_2_MTYPE(pp));
3147                                 ASSERT(pp->p_szc == 0);

3149                                 /* found a page with specified DMA attributes */
3150                                 page_sub(&PAGE_FREELISTS(mnode, szc, bin,
```



```
3151                                     mtype), pp);
3152                                 page_ctr_sub(mnode, mtype, pp, PG_FREE_LIST);

3154                                 if ((PP_ISFREE(pp) == 0) ||
3155                                     (PP_ISAGED(pp) == 0)) {
3156                                         cmn_err(CE_PANIC, "page %p is not free",
3157                                             (void *)pp);
3158                                 }

3160                                 mutex_exit(pcm);
3161                                 check_dma(dma_attr, pp, 1);
3162                                 VM_STAT_ADD(pga_vmstats.pgma_allocok);
3163                                 return (pp);
3164                         }
3165                         mutex_exit(pcm);
3166 nextfreebin:
3167                         if (plw_initialized == 0) {
3168                                 page_list_walk_init(szc, 0, bin, 1, 0, &plw);
3169                                 ASSERT(plw.plw_ceq_dif == page_colors);
3170                                 plw_initialized = 1;
3171                         }

3173                         if (plw.plw_do_split) {
3174                                 pp = page_freelist_split(szc, bin, mnode,
3175                                     mtype,
3176                                     mmu_btop(dma_attr->dma_attr_addr_lo),
3177                                     mmu_btop(dma_attr->dma_attr_addr_hi + 1),
3178                                     &plw);
3179                                 if (pp != NULL) {
3180                                         check_dma(dma_attr, pp, 1);
3181                                         return (pp);
3182                                 }
3183                         }

3185                         bin = page_list_walk_next_bin(szc, bin, &plw);
3186                 }

3188                 MTYPE_NEXT(mnode, mtype, flags);
3189         } while (mtype >= 0);

3191         /* failed to find a page in the freelist; try it in the cachelist */

3193         /* reset mtype start for cachelist search */
3194         mtype = mtypestart;
3195         ASSERT(mtype >= 0);

3197         /* start with the bin of matching color */
3198         bin = origbin;

3200         do {
3201                 for (i = 0; i <= page_colors; i++) {
3202                         if (PAGE_CACHELISTS(mnode, bin, mtype) == NULL)
3203                                 goto nextcachebin;
3204                         pcm = PC_BIN_MUTEX(mnode, bin, PG_CACHE_LIST);
3205                         mutex_enter(pcm);
3206                         pp = PAGE_CACHELISTS(mnode, bin, mtype);
3207                         first_pp = pp;
3208                         while (pp != NULL) {
3209                                 if (page_trylock(pp, SE_EXCL) == 0) {
3211                                 if (IS_DUMP_PAGE(pp) || page_trylock(pp,
3212                                     SE_EXCL) == 0) {
3210                                         pp = pp->p_next;
3211                                         if (pp == first_pp)
3212                                                 pp = NULL;
3213                                         continue;
3214                                 }
```

```
3215                                    ASSERT(pp->p_vnode);
3216                                    ASSERT(PP_ISAGED(pp) == 0);
3217                                    ASSERT(pp->p_szc == 0);
3218                                    ASSERT(PFN_2_MEM_NODE(pp->p_pagenum) == mnode);

3220                                    /* check if page within DMA attributes */

3222                                    pgaddr = pa_to_ma(pfn_to_pa(pp->p_pagenum));
3223                                    if ((pgaddr >= dma_attr->dma_attr_addr_lo) &&
3224                                        (pgaddr + MMU_PAGESIZE - 1 <=
3225                                        dma_attr->dma_attr_addr_hi)) {
3226                                            break;
3227                                    }

3229                                    /* continue looking */
3230                                    page_unlock(pp);
3231                                    pp = pp->p_next;
3232                                    if (pp == first_pp)
3233                                            pp = NULL;
3234                            }

3236                            if (pp != NULL) {
3237                                    ASSERT(mtype == PP_2_MTYPE(pp));
3238                                    ASSERT(pp->p_szc == 0);

3240                                    /* found a page with specified DMA attributes */
3241                                    page_sub(&PAGE_CACHELISTS(mnode, bin,
3242                                        mtype), pp);
3243                                    page_ctr_sub(mnode, mtype, pp, PG_CACHE_LIST);

3245                                    mutex_exit(pcm);
3246                                    ASSERT(pp->p_vnode);
3247                                    ASSERT(PP_ISAGED(pp) == 0);
3248                                    check_dma(dma_attr, pp, 1);
3249                                    VM_STAT_ADD(pga_vmstats.pgma_allocok);
3250                                    return (pp);
3251                            }
3252                            mutex_exit(pcm);
3253 nextcachebin:
3254                            bin += (i == 0) ? BIN_STEP : 1;
3255                            bin &= page_colors_mask;
3256                    }
3257                    MTYPE_NEXT(mnode, mtype, flags);
3258            } while (mtype >= 0);

3260            VM_STAT_ADD(pga_vmstats.pgma_allocfailed);
3261            return (NULL);
3262 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    6652 Fri Oct 26 17:54:29 2012
new/usr/src/uts/sun4/os/mp_states.c
[mq]: core-v2
**********************************************************
______unchanged_portion_omitted_

  58 static void
  59 cpu_idle_self(void)
  60 {
  61         uint_t s;
  62         label_t save;

  64         s = spl8();
  65         debug_flush_windows();

  67         CPU->cpu_m.in_prom = 1;
  68         membar_stld();

  70         save = curthread->t_pcb;
  71         (void) setjmp(&curthread->t_pcb);

  73         kern_idle[CPU->cpu_id] = 1;
  74         while (kern_idle[CPU->cpu_id])
  75                 /* SPIN */;
  75                 dumpsys_helper_nw();

  77         CPU->cpu_m.in_prom = 0;
  78         membar_stld();

  80         curthread->t_pcb = save;
  81         splx(s);
  82 }
______unchanged_portion_omitted_
```

```
**********************************************************
   13909 Fri Oct 26 17:54:30 2012
new/usr/src/uts/sun4u/os/mach_cpu_states.c
[mq]: core-v2
**********************************************************
______unchanged_portion_omitted_

 219 /*
 220  * We use the x-trap mechanism and idle_stop_xcall() to stop the other CPUs.
 221  * Once in panic_idle() they raise spl, record their location, and spin.
 222  */
 223 static void
 224 panic_idle(void)
 225 {
 226         cpu_async_panic_callb(); /* check for async errors */

 228         (void) spl7();

 230         debug_flush_windows();
 231         (void) setjmp(&curthread->t_pcb);

 233         CPU->cpu_m.in_prom = 1;
 234         membar_stld();

 236         dumpsys_helper();

 236         for (;;)
 237                 continue;
 238 }
______unchanged_portion_omitted_
```

****
38921 Fri Oct 26 17:54:30 2012
new/usr/src/uts/sun4v/os/mach_cpu_states.c
[mq]: core-v2
****
_____***unchanged_portion_omitted_***

```
 349 /*
 350  * We use the x-trap mechanism and idle_stop_xcall() to stop the other CPUs.
 351  * Once in panic_idle() they raise spl, record their location, and spin.
 352  */
 353 static void
 354 panic_idle(void)
 355 {
 356         (void) spl7();

 358         debug_flush_windows();
 359         (void) setjmp(&curthread->t_pcb);

 361         CPU->cpu_m.in_prom = 1;
 362         membar_stld();

 364         dumpsys_helper();

 364         for (;;)
 365                 ;
 366 }
```

_____***unchanged_portion_omitted_***