```
**********************************************************
   51372 Fri May  3 08:03:58 2019
new/usr/src/cmd/savecore/savecore.c
10654 savecore(1M) should be able to work on read-only dump devices
Reviewed by: Robert Mustacchi <rm@joyent.com>
Reviewed by: John Levon <john.levon@joyent.com>
Reviewed by: Andy_Stormont <astormont@racktopsystems.com>
Reviewed by: Gerg   Doma <domag02@gmail.com>
Reviewed by: Toomas Soome <tsoome@me.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1983, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2019 Joyent, Inc.
 * Copyright 2016 Joyent, Inc.
 */
/*
 * Copyright 2016 Nexenta Systems, Inc. All rights reserved.
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <deflt.h>
#include <time.h>
#include <syslog.h>
#include <stropts.h>
#include <pthread.h>
#include <limits.h>
#include <atomic.h>
#include <libnvpair.h>
#include <libintl.h>
#include <sys/mem.h>
#include <sys/statvfs.h>
#include <sys/dumphdr.h>
#include <sys/dumpadm.h>
#include <sys/compress.h>
#include <sys/panic.h>
#include <sys/sysmacros.h>
#include <sys/stat.h>
#include <sys/resource.h>
#include <bzip2/bzlib.h>
#include <sys/fm/util.h>
```



```
#include <fm/libfmevent.h>
#include <sys/int_fmtio.h>


/* fread/fwrite buffer size */
#define FBUFSIZE                (1ULL << 20)

/* minimum size for output buffering */
#define MINCOREBLKSIZE          (1ULL << 17)

/* create this file if metrics collection is enabled in the kernel */
#define METRICSFILE "METRICS.csv"

static char     progname[9] = "savecore";
static char     *savedir;               /* savecore directory */
static char     *dumpfile;              /* source of raw crash dump */
static long     bounds = -1;            /* numeric suffix */
static long     pagesize;               /* dump pagesize */
static int      dumpfd = -1;            /* dumpfile descriptor */
static boolean_t have_dumpfile = B_TRUE;        /* dumpfile existence */
static dumphdr_t corehdr, dumphdr;      /* initial and terminal dumphdrs */
static boolean_t dump_incomplete;       /* dumphdr indicates incomplete */
static boolean_t fm_panic;              /* dump is the result of fm_panic */
static offset_t endoff;                 /* offset of end-of-dump header */
static int      verbose;                /* chatty mode */
static int      disregard_valid_flag;   /* disregard valid flag */
static int      livedump;               /* dump the current running system */
static int      interactive;            /* user invoked; no syslog */
static int      csave;                  /* save dump compressed */
static int      filemode;               /* processing file, not dump device */
static int      percent_done;           /* progress indicator */
static int      sec_done;               /* progress last report time */
static hrtime_t startts;                /* timestamp at start */
static volatile uint64_t saved;         /* count of pages written */
static volatile uint64_t zpages;        /* count of zero pages not written */
static dumpdatahdr_t datahdr;           /* compression info */
static long     coreblksize;            /* preferred write size (st_blksize) */
static int      cflag;                  /* run as savecore -c */
static int      mflag;                  /* run as savecore -m */
static int      rflag;                  /* run as savecore -r */

/*
 * Payload information for the events we raise.  These are used
 * in raise_event to determine what payload to include.
 */
#define SC_PAYLOAD_SAVEDIR      0x0001  /* Include savedir in event */
#define SC_PAYLOAD_INSTANCE     0x0002  /* Include bounds instance number */
#define SC_PAYLOAD_IMAGEUUID    0x0004  /* Include dump OS instance uuid */
#define SC_PAYLOAD_CRASHTIME    0x0008  /* Include epoch crashtime */
#define SC_PAYLOAD_PANICSTR     0x0010  /* Include panic string */
#define SC_PAYLOAD_PANICSTACK   0x0020  /* Include panic string */
#define SC_PAYLOAD_FAILREASON   0x0040  /* Include failure reason */
#define SC_PAYLOAD_DUMPCOMPLETE 0x0080  /* Include completeness indicator */
#define SC_PAYLOAD_ISCOMPRESSED 0x0100  /* Dump is in vmdump.N form */
#define SC_PAYLOAD_DUMPADM_EN   0x0200  /* Is dumpadm enabled or not? */
#define SC_PAYLOAD_FM_PANIC     0x0400  /* Panic initiated by FMA */
#define SC_PAYLOAD_JUSTCHECKING 0x0800  /* Run with -c flag? */

enum sc_event_type {
        SC_EVENT_DUMP_PENDING,
        SC_EVENT_SAVECORE_FAILURE,
        SC_EVENT_DUMP_AVAILABLE
};
```

*unchanged_portion_omitted_*

```
static void raise_event(enum sc_event_type, char *);
```

```
164 static void
165 usage(void)
166 {
167         (void) fprintf(stderr,
168             "usage: %s [-L | -r] [-vd] [-f dumpfile] [dirname]\n", progname);
167             "usage: %s [-Lvd] [-f dumpfile] [dirname]\n", progname);
169         exit(1);
170 }

172 #define SC_SL_NONE      0x0001  /* no syslog */
173 #define SC_SL_ERR       0x0002  /* syslog if !interactive, LOG_ERR */
174 #define SC_SL_WARN      0x0004  /* syslog if !interactive, LOG_WARNING */
175 #define SC_IF_VERBOSE   0x0008  /* message only if -v */
176 #define SC_IF_ISATTY    0x0010  /* message only if interactive */
177 #define SC_EXIT_OK      0x0020  /* exit(0) */
178 #define SC_EXIT_ERR     0x0040  /* exit(1) */
179 #define SC_EXIT_PEND    0x0080  /* exit(2) */
180 #define SC_EXIT_FM      0x0100  /* exit(3) */

182 #define _SC_ALLEXIT     (SC_EXIT_OK | SC_EXIT_ERR | SC_EXIT_PEND | SC_EXIT_FM)

184 static void
185 logprint(uint32_t flags, char *message, ...)
186 {
187         va_list args;
188         char buf[1024];
189         int do_always = ((flags & (SC_IF_VERBOSE | SC_IF_ISATTY)) == 0);
190         int do_ifverb = (flags & SC_IF_VERBOSE) && verbose;
191         int do_ifisatty = (flags & SC_IF_ISATTY) && interactive;
192         int code;
193         static int logprint_raised = 0;

195         if (do_always || do_ifverb || do_ifisatty) {
196                 va_start(args, message);
197                 /*LINTED: E_SEC_PRINTF_VAR_FMT*/
198                 (void) vsnprintf(buf, sizeof (buf), message, args);
199                 (void) fprintf(stderr, "%s: %s\n", progname, buf);
200                 if (!interactive) {
201                         switch (flags & (SC_SL_NONE | SC_SL_ERR | SC_SL_WARN)) {
202                         case SC_SL_ERR:
203                                 /*LINTED: E_SEC_PRINTF_VAR_FMT*/
204                                 syslog(LOG_ERR, buf);
205                                 break;

207                         case SC_SL_WARN:
208                                 /*LINTED: E_SEC_PRINTF_VAR_FMT*/
209                                 syslog(LOG_WARNING, buf);
210                                 break;

212                         default:
213                                 break;
214                         }
215                 }
216                 va_end(args);
217         }

219         switch (flags & _SC_ALLEXIT) {
220         case 0:
221                 return;

223         case SC_EXIT_OK:
224                 code = 0;
225                 break;

227         case SC_EXIT_PEND:
```



```
228                 /*
229                  * Raise an ireport saying why we are exiting.  Do not
230                  * raise if run as savecore -m.  If something in the
231                  * raise_event codepath calls logprint avoid recursion.
232                  */
233                 if (!mflag && !rflag && logprint_raised++ == 0)
232                 if (!mflag && logprint_raised++ == 0)
234                         raise_event(SC_EVENT_SAVECORE_FAILURE, buf);
235                 code = 2;
236                 break;

238         case SC_EXIT_FM:
239                 code = 3;
240                 break;

242         case SC_EXIT_ERR:
243         default:
244                 if (!mflag && !rflag && logprint_raised++ == 0 && have_dumpfile)
243                 if (!mflag && logprint_raised++ == 0 && have_dumpfile)
245                         raise_event(SC_EVENT_SAVECORE_FAILURE, buf);
246                 code = 1;
247                 break;
248         }

250         exit(code);
251 }
_____unchanged_portion_omitted_

356 static void
357 read_dumphdr(void)
358 {
359         if (filemode || rflag)
358         if (filemode)
360                 dumpfd = Open(dumpfile, O_RDONLY, 0644);
361         else
362                 dumpfd = Open(dumpfile, O_RDWR | O_DSYNC, 0644);
363         endoff = llseek(dumpfd, -DUMP_OFFSET, SEEK_END) & -DUMP_OFFSET;
364         Pread(dumpfd, &dumphdr, sizeof (dumphdr), endoff);
365         Pread(dumpfd, &datahdr, sizeof (datahdr), endoff + sizeof (dumphdr));

367         pagesize = dumphdr.dump_pagesize;

369         if (dumphdr.dump_magic != DUMP_MAGIC)
370                 logprint(SC_SL_NONE | SC_EXIT_PEND, "bad magic number %x",
371                     dumphdr.dump_magic);

373         if ((dumphdr.dump_flags & DF_VALID) == 0 && !disregard_valid_flag)
374                 logprint(SC_SL_NONE | SC_IF_VERBOSE | SC_EXIT_OK,
375                     "dump already processed");

377         if (dumphdr.dump_version != DUMP_VERSION)
378                 logprint(SC_SL_NONE | SC_IF_VERBOSE | SC_EXIT_PEND,
379                     "dump version (%d) != %s version (%d)",
380                     dumphdr.dump_version, progname, DUMP_VERSION);

382         if (dumphdr.dump_wordsize != DUMP_WORDSIZE)
383                 logprint(SC_SL_NONE | SC_EXIT_PEND,
384                     "dump is from %u-bit kernel - cannot save on %u-bit kernel",
385                     dumphdr.dump_wordsize, DUMP_WORDSIZE);

387         if (datahdr.dump_datahdr_magic == DUMP_DATAHDR_MAGIC) {
388                 if (datahdr.dump_datahdr_version != DUMP_DATAHDR_VERSION)
389                         logprint(SC_SL_NONE | SC_IF_VERBOSE | SC_EXIT_PEND,
390                             "dump data version (%d) != %s data version (%d)",
391                             datahdr.dump_datahdr_version, progname,
392                             DUMP_DATAHDR_VERSION);
```

```
393         } else {
394                 (void) memset(&datahdr, 0, sizeof (datahdr));
395                 datahdr.dump_maxcsize = pagesize;
396         }

398         /*
399          * Read the initial header, clear the valid bits, and compare headers.
400          * The main header may have been overwritten by swapping if we're
401          * using a swap partition as the dump device, in which case we bail.
402          */
403         Pread(dumpfd, &corehdr, sizeof (dumphdr_t), dumphdr.dump_start);

405         corehdr.dump_flags &= ~DF_VALID;
406         dumphdr.dump_flags &= ~DF_VALID;

408         if (memcmp(&corehdr, &dumphdr, sizeof (dumphdr_t)) != 0) {
409                 /*
410                  * Clear valid bit so we don't complain on every invocation.
411                  */
412                 if (!filemode && !rflag)
411                 if (!filemode)
413                         Pwrite(dumpfd, &dumphdr, sizeof (dumphdr), endoff);
414                 logprint(SC_SL_ERR | SC_EXIT_ERR,
415                     "initial dump header corrupt");
416         }
417 }
```

*unchanged_portion_omitted_*

```
565 /*
566  * Concatenate dump contents into a new file.
567  * Update corehdr with new offsets.
568  */
569 static void
570 copy_crashfile(const char *corefile)
571 {
572         int corefd = Open(corefile, O_WRONLY | O_CREAT | O_TRUNC, 0644);
573         size_t bufsz = FBUFSIZE;
574         char *inbuf = Zalloc(bufsz);
575         offset_t coreoff;
576         size_t nb;

578         logprint(SC_SL_ERR | SC_IF_VERBOSE,
579             "Copying %s to %s/%s\n", dumpfile, savedir, corefile);

581         /*
582          * This dump file is still compressed
583          */
584         corehdr.dump_flags |= DF_COMPRESSED | DF_VALID;

586         /*
587          * Leave room for corehdr, it is updated and written last
588          */
589         corehdr.dump_start = 0;
590         coreoff = sizeof (corehdr);

592         /*
593          * Read in the compressed symbol table, copy it to corefile.
594          */
595         coreoff = roundup(coreoff, pagesize);
596         corehdr.dump_ksyms = coreoff;
597         Copy(dumphdr.dump_ksyms, dumphdr.dump_ksyms_csize, &coreoff, corefd,
598             inbuf, bufsz);

600         /*
601          * Save the pfn table.
602          */
```



```
603         coreoff = roundup(coreoff, pagesize);
604         corehdr.dump_pfn = coreoff;
605         Copy(dumphdr.dump_pfn, dumphdr.dump_npages * sizeof (pfn_t), &coreoff,
606             corefd, inbuf, bufsz);

608         /*
609          * Save the dump map.
610          */
611         coreoff = roundup(coreoff, pagesize);
612         corehdr.dump_map = coreoff;
613         Copy(dumphdr.dump_map, dumphdr.dump_nvtop * sizeof (mem_vtop_t),
614             &coreoff, corefd, inbuf, bufsz);

616         /*
617          * Save the data pages.
618          */
619         coreoff = roundup(coreoff, pagesize);
620         corehdr.dump_data = coreoff;
621         if (datahdr.dump_data_csize != 0)
622                 Copy(dumphdr.dump_data, datahdr.dump_data_csize, &coreoff,
623                     corefd, inbuf, bufsz);
624         else
625                 CopyPages(&coreoff, corefd, inbuf, bufsz);

627         /*
628          * Now write the modified dump header to front and end of the copy.
629          * Make it look like a valid dump device.
630          *
631          * From dumphdr.h: Two headers are written out: one at the
632          * beginning of the dump, and the other at the very end of the
633          * dump device. The terminal header is at a known location
634          * (end of device) so we can always find it.
635          *
636          * Pad with zeros to each DUMP_OFFSET boundary.
637          */
638         (void) memset(inbuf, 0, DUMP_OFFSET);

640         nb = DUMP_OFFSET - (coreoff & (DUMP_OFFSET - 1));
641         if (nb > 0) {
642                 Pwrite(corefd, inbuf, nb, coreoff);
643                 coreoff += nb;
644         }

646         Pwrite(corefd, &corehdr, sizeof (corehdr), coreoff);
647         coreoff += sizeof (corehdr);

649         Pwrite(corefd, &datahdr, sizeof (datahdr), coreoff);
650         coreoff += sizeof (datahdr);

652         nb = DUMP_OFFSET - (coreoff & (DUMP_OFFSET - 1));
653         if (nb > 0) {
654                 Pwrite(corefd, inbuf, nb, coreoff);
655         }

657         free(inbuf);
658         Pwrite(corefd, &corehdr, sizeof (corehdr), corehdr.dump_start);

660         /*
661          * Write out the modified dump header to the dump device.
662          * The dump device has been processed, so DF_VALID is clear.
663          */
664         if (!filemode && !rflag)
663         if (!filemode)
665                 Pwrite(dumpfd, &dumphdr, sizeof (dumphdr), endoff);

667         (void) close(corefd);
```

```
 668 }
______unchanged_portion_omitted_

1335 static void
1336 build_corefile(const char *namelist, const char *corefile)
1337 {
1338         size_t pfn_table_size = dumphdr.dump_npages * sizeof (pfn_t);
1339         size_t ksyms_size = dumphdr.dump_ksyms_size;
1340         size_t ksyms_csize = dumphdr.dump_ksyms_csize;
1341         pfn_t *pfn_table;
1342         char *ksyms_base = Zalloc(ksyms_size);
1343         char *ksyms_cbase = Zalloc(ksyms_csize);
1344         size_t ksyms_dsize;
1345         Stat_t st;
1346         int corefd = Open(corefile, O_WRONLY | O_CREAT | O_TRUNC, 0644);
1347         int namefd = Open(namelist, O_WRONLY | O_CREAT | O_TRUNC, 0644);

1349         (void) printf("Constructing namelist %s/%s\n", savedir, namelist);

1351         /*
1352          * Determine the optimum write size for the core file
1353          */
1354         Fstat(corefd, &st, corefile);

1356         if (verbose > 1)
1357                 (void) printf("%s: %ld block size\n", corefile,
1358                     (long)st.st_blksize);
1359         coreblksize = st.st_blksize;
1360         if (coreblksize < MINCOREBLKSIZE || !ISP2(coreblksize))
1361                 coreblksize = MINCOREBLKSIZE;

1363         hist = Zalloc((sizeof (uint64_t) * BTOP(coreblksize)) + 1);

1365         /*
1366          * This dump file is now uncompressed
1367          */
1368         corehdr.dump_flags &= ~DF_COMPRESSED;

1370         /*
1371          * Read in the compressed symbol table, copy it to corefile,
1372          * decompress it, and write the result to namelist.
1373          */
1374         corehdr.dump_ksyms = pagesize;
1375         Pread(dumpfd, ksyms_cbase, ksyms_csize, dumphdr.dump_ksyms);
1376         Pwrite(corefd, ksyms_cbase, ksyms_csize, corehdr.dump_ksyms);

1378         ksyms_dsize = decompress(ksyms_cbase, ksyms_base, ksyms_csize,
1379             ksyms_size);
1380         if (ksyms_dsize != ksyms_size)
1381                 logprint(SC_SL_WARN,
1382                     "bad data in symbol table, %lu of %lu bytes saved",
1383                     ksyms_dsize, ksyms_size);

1385         Pwrite(namefd, ksyms_base, ksyms_size, 0);
1386         (void) close(namefd);
1387         free(ksyms_cbase);
1388         free(ksyms_base);

1390         (void) printf("Constructing corefile %s/%s\n", savedir, corefile);

1392         /*
1393          * Read in and write out the pfn table.
1394          */
1395         pfn_table = Zalloc(pfn_table_size);
1396         corehdr.dump_pfn = corehdr.dump_ksyms + roundup(ksyms_size, pagesize);
1397         Pread(dumpfd, pfn_table, pfn_table_size, dumphdr.dump_pfn);
```



```
1398         Pwrite(corefd, pfn_table, pfn_table_size, corehdr.dump_pfn);

1400         /*
1401          * Convert the raw translation data into a hashed dump map.
1402          */
1403         corehdr.dump_map = corehdr.dump_pfn + roundup(pfn_table_size, pagesize);
1404         build_dump_map(corefd, pfn_table);
1405         free(pfn_table);

1407         /*
1408          * Decompress the pages
1409          */
1410         decompress_pages(corefd);
1411         (void) printf(": %ld of %ld pages saved\n", (pgcnt_t)saved,
1412             dumphdr.dump_npages);

1414         if (verbose)
1415                 (void) printf("%ld (%ld%%) zero pages were not written\n",
1416                     (pgcnt_t)zpages, (pgcnt_t)zpages * 100 /
1417                     dumphdr.dump_npages);

1419         if (saved != dumphdr.dump_npages)
1420                 logprint(SC_SL_WARN, "bad data after page %ld", saved);

1422         /*
1423          * Write out the modified dump headers.
1424          */
1425         Pwrite(corefd, &corehdr, sizeof (corehdr), 0);
1426         if (!filemode && !rflag)
1425         if (!filemode)
1427                 Pwrite(dumpfd, &dumphdr, sizeof (dumphdr), endoff);

1429         (void) close(corefd);
1430 }
______unchanged_portion_omitted_

1527 static void
1528 stack_retrieve(char *stack)
1529 {
1530         summary_dump_t sd;
1531         offset_t dumpoff = -(DUMP_OFFSET + DUMP_LOGSIZE +
1532             DUMP_ERPTSIZE);
1533         dumpoff -= DUMP_SUMMARYSIZE;

1535         if (rflag)
1536                 dumpfd = Open(dumpfile, O_RDONLY, 0644);
1537         else
1538                 dumpfd = Open(dumpfile, O_RDWR | O_DSYNC, 0644);
1539         dumpoff = llseek(dumpfd, dumpoff, SEEK_END) & -DUMP_OFFSET;

1541         Pread(dumpfd, &sd, sizeof (summary_dump_t), dumpoff);
1542         dumpoff += sizeof (summary_dump_t);

1544         if (sd.sd_magic == 0) {
1545                 *stack = '\0';
1546                 return;
1547         }

1549         if (sd.sd_magic != SUMMARY_MAGIC) {
1550                 *stack = '\0';
1551                 logprint(SC_SL_NONE | SC_IF_VERBOSE,
1552                     "bad summary magic %x", sd.sd_magic);
1553                 return;
1554         }
1555         Pread(dumpfd, stack, STACK_BUF_SIZE, dumpoff);
1556         if (sd.sd_ssum != checksum32(stack, STACK_BUF_SIZE))
```

```
1557                 logprint(SC_SL_NONE | SC_IF_VERBOSE, "bad stack checksum");
1558 }
_____unchanged_portion_omitted_


1652 int
1653 main(int argc, char *argv[])
1654 {
1655         int i, c, bfd;
1656         Stat_t st;
1657         struct rlimit rl;
1658         long filebounds = -1;
1659         char namelist[30], corefile[30], boundstr[30];
1660         dumpfile = NULL;

1662         startts = gethrtime();

1664         (void) getrlimit(RLIMIT_NOFILE, &rl);
1665         rl.rlim_cur = rl.rlim_max;
1666         (void) setrlimit(RLIMIT_NOFILE, &rl);

1668         openlog(progname, LOG_ODELAY, LOG_AUTH);

1670         (void) defopen("/etc/dumpadm.conf");
1671         savedir = defread("DUMPADM_SAVDIR=");
1672         if (savedir != NULL)
1673                 savedir = strdup(savedir);

1675         while ((c = getopt(argc, argv, "Lvcdmf:r")) != EOF) {
1671         while ((c = getopt(argc, argv, "Lvcdmf:")) != EOF) {
1676                 switch (c) {
1677                 case 'L':
1678                         livedump++;
1679                         break;
1680                 case 'v':
1681                         verbose++;
1682                         break;
1683                 case 'c':
1684                         cflag++;
1685                         break;
1686                 case 'd':
1687                         disregard_valid_flag++;
1688                         break;
1689                 case 'm':
1690                         mflag++;
1691                         break;
1692                 case 'r':
1693                         rflag++;
1694                         break;
1695                 case 'f':
1696                         dumpfile = optarg;
1697                         filebounds = getbounds(dumpfile);
1698                         break;
1699                 case '?':
1700                         usage();
1701                 }
1702         }

1704         /*
1705          * If doing something other than extracting an existing dump (i.e.
1706          * dumpfile has been provided as an option), the user must be root.
1707          */
1708         if (geteuid() != 0 && dumpfile == NULL) {
1709                 (void) fprintf(stderr, "%s: %s %s\n", progname,
1710                     gettext("you must be root to use"), progname);
1711                 exit(1);
```



```
1712         }

1714         interactive = isatty(STDOUT_FILENO);

1716         if (cflag && livedump)
1717                 usage();

1719         if (rflag && (cflag || mflag || livedump))
1720                 usage();

1722         if (dumpfile == NULL || livedump)
1723                 dumpfd = Open("/dev/dump", O_RDONLY, 0444);

1725         if (dumpfile == NULL) {
1726                 dumpfile = Zalloc(MAXPATHLEN);
1727                 if (ioctl(dumpfd, DIOCGETDEV, dumpfile) == -1) {
1728                         have_dumpfile = B_FALSE;
1729                         logprint(SC_SL_NONE | SC_IF_ISATTY | SC_EXIT_ERR,
1730                             "no dump device configured");
1731                 }
1732         }

1734         if (mflag)
1735                 return (message_save());

1737         if (optind == argc - 1)
1738                 savedir = argv[optind];

1740         if (savedir == NULL || optind < argc - 1)
1741                 usage();

1743         if (livedump && ioctl(dumpfd, DIOCDUMP, NULL) == -1)
1744                 logprint(SC_SL_NONE | SC_EXIT_ERR,
1745                     "dedicated dump device required");

1747         (void) close(dumpfd);
1748         dumpfd = -1;

1750         Stat(dumpfile, &st);

1752         filemode = S_ISREG(st.st_mode);

1754         if (!filemode && defread("DUMPADM_CSAVE=off") == NULL)
1755                 csave = 1;

1757         read_dumphdr();

1759         /*
1760          * We want this message to go to the log file, but not the console.
1761          * There's no good way to do that with the existing syslog facility.
1762          * We could extend it to handle this, but there doesn't seem to be
1763          * a general need for it, so we isolate the complexity here instead.
1764          */
1765         if (dumphdr.dump_panicstring[0] != '\0' && !rflag) {
1755         if (dumphdr.dump_panicstring[0] != '\0') {
1766                 int logfd = Open("/dev/conslog", O_WRONLY, 0644);
1767                 log_ctl_t lc;
1768                 struct strbuf ctl, dat;
1769                 char msg[DUMP_PANICSIZE + 100];
1770                 char fmt[] = "reboot after panic: %s";
1771                 uint32_t msgid;

1773                 STRLOG_MAKE_MSGID(fmt, msgid);

1775                 /* LINTED: E_SEC_SPRINTF_UNBOUNDED_COPY */
1776                 (void) sprintf(msg, "%s: [ID %u FACILITY_AND_PRIORITY] ",
```

```
1777                    progname, msgid);
1778                /* LINTED: E_SEC_PRINTF_VAR_FMT */
1779                (void) sprintf(msg + strlen(msg), fmt,
1780                    dumphdr.dump_panicstring);

1782                lc.pri = LOG_AUTH | LOG_ERR;
1783                lc.flags = SL_CONSOLE | SL_LOGONLY;
1784                lc.level = 0;

1786                ctl.buf = (void *)&lc;
1787                ctl.len = sizeof (log_ctl_t);

1789                dat.buf = (void *)msg;
1790                dat.len = strlen(msg) + 1;

1792                (void) putmsg(logfd, &ctl, &dat, 0);
1793                (void) close(logfd);
1794        }

1796        if ((dumphdr.dump_flags & DF_COMPLETE) == 0) {
1797                logprint(SC_SL_WARN, "incomplete dump on dump device");
1798                dump_incomplete = B_TRUE;
1799        }

1801        if (dumphdr.dump_fm_panic)
1802                fm_panic = B_TRUE;

1804        /*
1805         * We have a valid dump on a dump device and know as much about
1806         * it as we're going to at this stage.  Raise an event for
1807         * logging and so that FMA can open a case for this panic.
1808         * Avoid this step for FMA-initiated panics - FMA will replay
1809         * ereports off the dump device independently of savecore and
1810         * will make a diagnosis, so we don't want to open two cases
1811         * for the same event.  Also avoid raising an event for a
1812         * livedump, or when we inflating a compressed dump.
1813         */
1814        if (!fm_panic && !livedump && !filemode && !rflag)
1804        if (!fm_panic && !livedump && !filemode)
1815                raise_event(SC_EVENT_DUMP_PENDING, NULL);

1817        logprint(SC_SL_WARN, "System dump time: %s",
1818            ctime(&dumphdr.dump_crashtime));

1820        /*
1821         * Option -c is designed for use from svc-dumpadm where we know
1822         * that dumpadm -n is in effect but run savecore -c just to
1823         * get the above dump_pending_on_device event raised.  If it is run
1824         * interactively then just print further panic details.
1825         */
1826        if (cflag) {
1827                char *disabled = defread("DUMPADM_ENABLE=no");
1828                int lvl = interactive ? SC_SL_WARN : SC_SL_ERR;
1829                int ec = fm_panic ? SC_EXIT_FM : SC_EXIT_PEND;

1831                logprint(lvl | ec,
1832                    "Panic crashdump pending on dump device%s "
1833                    "run savecore(1M) manually to extract. "
1834                    "Image UUID %s%s.",
1835                    disabled ? " but dumpadm -n in effect;" : ";",
1836                    corehdr.dump_uuid,
1837                    fm_panic ?  "(fault-management initiated)" : "");
1838                /*NOTREACHED*/
1839        }

1841        if (chdir(savedir) == -1)
```



```
1842                logprint(SC_SL_ERR | SC_EXIT_ERR, "chdir(\"%s\"): %s",
1843                    savedir, strerror(errno));

1845        check_space(csave);

1847        if (filebounds < 0)
1848                bounds = read_number_from_file("bounds", 0);
1849        else
1850                bounds = filebounds;

1852        if (csave) {
1853                size_t metrics_size = datahdr.dump_metrics;

1855                (void) sprintf(corefile, "vmdump.%ld", bounds);

1857                datahdr.dump_metrics = 0;

1859                logprint(SC_SL_ERR,
1860                    "Saving compressed system crash dump in %s/%s",
1861                    savedir, corefile);

1863                copy_crashfile(corefile);

1865                /*
1866                 * Raise a fault management event that indicates the system
1867                 * has panicked. We know a reasonable amount about the
1868                 * condition at this time, but the dump is still compressed.
1869                 */
1870                if (!livedump && !fm_panic && !rflag)
1860                if (!livedump && !fm_panic)
1871                        raise_event(SC_EVENT_DUMP_AVAILABLE, NULL);

1873                if (metrics_size > 0) {
1874                        int sec = (gethrtime() - startts) / 1000 / 1000 / 1000;
1875                        FILE *mfile = fopen(METRICSFILE, "a");
1876                        char *metrics = Zalloc(metrics_size + 1);

1878                        Pread(dumpfd, metrics, metrics_size, endoff +
1879                            sizeof (dumphdr) + sizeof (datahdr));

1881                        if (sec < 1)
1882                                sec = 1;

1884                        if (mfile == NULL) {
1885                                logprint(SC_SL_WARN,
1886                                    "Can't create %s:\n%s",
1887                                    METRICSFILE, metrics);
1888                        } else {
1889                                (void) fprintf(mfile, "[[[[,,,");
1890                                for (i = 0; i < argc; i++)
1891                                        (void) fprintf(mfile, "%s ", argv[i]);
1892                                (void) fprintf(mfile, "\n");
1893                                (void) fprintf(mfile, ",,,%s %s %s %s %s\n",
1894                                    dumphdr.dump_utsname.sysname,
1895                                    dumphdr.dump_utsname.nodename,
1896                                    dumphdr.dump_utsname.release,
1897                                    dumphdr.dump_utsname.version,
1898                                    dumphdr.dump_utsname.machine);
1899                                (void) fprintf(mfile, ",,,%s dump time %s\n",
1900                                    dumphdr.dump_flags & DF_LIVE ? "Live" :
1901                                    "Crash", ctime(&dumphdr.dump_crashtime));
1902                                (void) fprintf(mfile, ",,,%s/%s\n", savedir,
1903                                    corefile);
1904                                (void) fprintf(mfile, "Metrics:\n%s\n",
1905                                    metrics);
1906                                (void) fprintf(mfile, "Copy pages,%ld\n",
```

```
1907                                    dumphdr.  dump_npages);
1908                                (void) fprintf(mfile, "Copy time,%d\n", sec);
1909                                (void) fprintf(mfile, "Copy pages/sec,%ld\n",
1910                                    dumphdr.dump_npages / sec);
1911                                (void) fprintf(mfile, "]]]]\n");
1912                                (void) fclose(mfile);
1913                        }
1914                        free(metrics);
1915                }

1917                logprint(SC_SL_ERR,
1918                    "Decompress the crash dump with "
1919                    "\n'savecore -vf %s/%s'",
1920                    savedir, corefile);

1922        } else {
1923                (void) sprintf(namelist, "unix.%ld", bounds);
1924                (void) sprintf(corefile, "vmcore.%ld", bounds);

1926                if (interactive && filebounds >= 0 && access(corefile, F_OK)
1927                    == 0)
1928                        logprint(SC_SL_NONE | SC_EXIT_ERR,
1929                            "%s already exists: remove with "
1930                            "'rm -f %s/{unix,vmcore}.%ld'",
1931                            corefile, savedir, bounds);

1933                logprint(SC_SL_ERR,
1934                    "saving system crash dump in %s/{unix,vmcore}.%ld",
1935                    savedir, bounds);

1937                build_corefile(namelist, corefile);

1939                if (!livedump && !filemode && !fm_panic && !rflag)
1929                /* (!livedump && !filemode && !fm_panic) */
1940                        raise_event(SC_EVENT_DUMP_AVAILABLE, NULL);

1942                if (access(METRICSFILE, F_OK) == 0) {
1943                        int sec = (gethrtime() - startts) / 1000 / 1000 / 1000;
1944                        FILE *mfile = fopen(METRICSFILE, "a");

1946                        if (sec < 1)
1947                                sec = 1;

1949                        if (mfile == NULL) {
1950                                logprint(SC_SL_WARN,
1951                                    "Can't create %s: %s",
1952                                    METRICSFILE, strerror(errno));
1953                        } else {
1954                                (void) fprintf(mfile, "[[[[,,,");
1955                                for (i = 0; i < argc; i++)
1956                                        (void) fprintf(mfile, "%s ", argv[i]);
1957                                (void) fprintf(mfile, "\n");
1958                                (void) fprintf(mfile, ",,,%s/%s\n", savedir,
1959                                    corefile);
1960                                (void) fprintf(mfile, ",,,%s %s %s %s %s\n",
1961                                    dumphdr.dump_utsname.sysname,
1962                                    dumphdr.dump_utsname.nodename,
1963                                    dumphdr.dump_utsname.release,
1964                                    dumphdr.dump_utsname.version,
1965                                    dumphdr.dump_utsname.machine);
1966                                (void) fprintf(mfile,
1967                                    "Uncompress pages,%"PRIu64"\n", saved);
1968                                (void) fprintf(mfile, "Uncompress time,%d\n",
1969                                    sec);
1970                                (void) fprintf(mfile, "Uncompress pages/sec,%"
1971                                    PRIu64"\n", saved / sec);
```



```
1972                                (void) fprintf(mfile, "]]]]\n");
1973                                (void) fclose(mfile);
1974                        }
1975                }
1976        }

1978        if (filebounds < 0) {
1979                (void) sprintf(boundstr, "%ld\n", bounds + 1);
1980                bfd = Open("bounds", O_WRONLY | O_CREAT | O_TRUNC, 0644);
1981                Pwrite(bfd, boundstr, strlen(boundstr), 0);
1982                (void) close(bfd);
1983        }

1985        if (verbose) {
1986                int sec = (gethrtime() - startts) / 1000 / 1000 / 1000;

1988                (void) printf("%d:%02d dump %s is done\n",
1989                    sec / 60, sec % 60,
1990                    csave ? "copy" : "decompress");
1991        }

1993        if (verbose > 1 && hist != NULL) {
1994                int i, nw;

1996                for (i = 1, nw = 0; i <= BTOP(coreblksize); ++i)
1997                        nw += hist[i] * i;
1998                (void) printf("pages count     %%\n");
1999                for (i = 0; i <= BTOP(coreblksize); ++i) {
2000                        if (hist[i] == 0)
2001                                continue;
2002                        (void) printf("%3d   %5u  %6.2f\n",
2003                            i, hist[i], 100.0 * hist[i] * i / nw);
2004                }
2005        }

2007        (void) close(dumpfd);
2008        dumpfd = -1;

2010        return (0);
2011 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    5328 Fri May  3 08:03:58 2019
new/usr/src/man/man1m/savecore.1m
10654 savecore(1M) should be able to work on read-only dump devices
Reviewed by: Robert Mustacchi <rm@joyent.com>
Reviewed by: John Levon <john.levon@joyent.com>
Reviewed by: Andy_Stormont <astormont@racktopsystems.com>
Reviewed by: Gerg   Doma <domag02@gmail.com>
Reviewed by: Toomas Soome <tsoome@me.com>
**********************************************************
   1 '\" te
   2 .\" Copyright (c) 2004, Sun Microsystems, Inc. All Rights Reserved.
   3 .\" Copyright (c) 1983 Regents of the University of California. All rights reser
   4 .\" Copyright 2013 Nexenta Systems, Inc.  All Rights Reserved.
   5 .\" Copyright 2019 Joyent, Inc.
   6 .TH SAVECORE 1M "February 22, 2019"
   5 .TH SAVECORE 1M "Jan 30, 2013"
   7 .SH NAME
   8 savecore \- save a crash dump of the operating system
   9 .SH SYNOPSIS
  10 .LP
  11 .nf
  12 \fB/usr/bin/savecore\fR [\fB-L\fR | \fB-r\fR] [\fB-vd\fR] [\fB-f\fR \fIdumpfile\
  11 \fB/usr/bin/savecore\fR [\fB-Lvd\fR] [\fB-f\fR \fIdumpfile\fR] [\fIdirectory\fR]
  13 .fi

  15 .SH DESCRIPTION
  15 .sp
  16 .LP
  17 The \fBsavecore\fR utility saves a crash dump of the kernel (assuming that one
  18 was made) and writes a reboot message in the shutdown log. By default, it is
  19 invoked by the \fBdumpadm\fR service each time the system boots.
  20 .sp
  21 .LP
  22 Depending on the \fBdumpadm\fR(1M) configuration \fBsavecore\fR saves either
  23 the compressed or uncompressed crash dump. The compressed crash dump is saved in
  24 the file \fIdirectory\fR\fB/vmdump.\fR\fIn\fR.
  25 \fBsavecore\fR saves the uncompressed crash dump data in the file
  26 \fIdirectory\fR\fB/vmcore.\fR\fIn\fR and the kernel's namelist in
  27 \fIdirectory\fR\fB/unix.\fR\fIn.\fR The trailing \fIn\fR in the
  28 pathnames is replaced by a number which grows every time \fBsavecore\fR is run
  29 in that directory.
  30 .sp
  31 .LP
  32 Before writing out a crash dump, \fBsavecore\fR reads a number from the file
  33 \fIdirectory\fR\fB/minfree\fR. This is the minimum number of kilobytes that
  34 must remain free on the file system containing \fIdirectory\fR. If after saving
  35 the crash dump the file system containing \fIdirectory\fR would have less free
  36 space the number of kilobytes specified in \fBminfree\fR, the crash dump is not
  37 saved. if the \fBminfree\fR file does not exist, \fBsavecore\fR assumes a
  38 \fBminfree\fR value of 1 megabyte.
  39 .sp
  40 .LP
  41 The \fBsavecore\fR utility also logs a reboot message using facility
  42 \fBLOG_AUTH\fR (see \fBsyslog\fR(3C)). If the system crashed as a result of a
  43 panic, \fBsavecore\fR logs the panic string too.
  44 .SH OPTIONS
  45 .sp
  45 .LP
  46 The following options are supported:
  47 .sp
  48 .ne 2
  49 .na
  50 \fB\fB-d\fR\fR
  51 .ad
  52 .RS 15n
```



```
  53 Disregard dump header valid flag. Force \fBsavecore\fR to attempt to save a
  54 crash dump even if the header information stored on the dump device indicates
  55 the dump has already been saved.
  56 .RE

  58 .sp
  59 .ne 2
  60 .na
  61 \fB\fB-f\fR \fIdumpfile\fR\fR
  62 .ad
  63 .RS 15n
  64 Attempt to save a crash dump from the specified file instead of from the
  65 system's current dump device. This option may be useful if the information
  66 stored on the dump device has been copied to an on-disk file by means of the
  67 \fBdd\fR(1M) command.
  68 .RE

  70 .sp
  71 .ne 2
  72 .na
  73 \fB\fB-L\fR\fR
  74 .ad
  75 .RS 15n
  76 Save a crash dump of the live running Solaris system, without actually
  77 rebooting or altering the system in any way. This option forces \fBsavecore\fR
  78 to save a live snapshot of the system to the dump device, and then immediately
  79 to retrieve the data and to write it out to a new set of crash dump files in
  80 the specified directory. Live system crash dumps can only be performed if you
  81 have configured your system to have a dedicated dump device using
  82 \fBdumpadm\fR(1M).
  83 .sp
  84 \fBsavecore\fR \fB-L\fR does not suspend the system, so the contents of memory
  85 continue to change while the dump is saved. This means that live crash dumps
  86 are not fully self-consistent.
  87 .RE

  89 .sp
  90 .ne 2
  91 .na
  92 \fB\fB-r\fR\fR
  93 .ad
  94 .RS 15n
  95 Open the dump device or file as read-only, and don't update the dump header
  96 or do anything else that might modify the crash dump. This option can be used
  97 to recover a crash dump from a read-only device. This flag cannot be used in
  98 conjunction with \fB\fB-L\fR\fR.
  99 .RE

 101 .sp
 102 .ne 2
 103 .na
 104 \fB\fB-v\fR\fR
 105 .ad
 106 .RS 15n
 107 Verbose. Enables verbose error messages from \fBsavecore\fR.
 108 .RE

 110 .SH OPERANDS
 100 .sp
 111 .LP
 112 The following operands are supported:
 113 .sp
 114 .ne 2
 115 .na
 116 \fB\fIdirectory\fR\fR
 117 .ad
```

```
118 .RS 13n
119 Save the crash dump files to the specified directory. If \fIdirectory\fR is not
120 specified, \fBsavecore\fR saves the crash dump files to the default
121 \fBsavecore\fR \fIdirectory\fR, configured by \fBdumpadm\fR(1M).
122 .RE

124 .SH FILES
115 .sp
125 .ne 2
126 .na
127 \fB\fIdirectory\fR\fB/vmdump.\fR\fIn\fR\fR
128 .ad
129 .RS 29n

131 .RE

133 .sp
134 .ne 2
135 .na
136 \fB\fIdirectory\fR\fB/vmcore.\fR\fIn\fR\fR
137 .ad
138 .RS 29n

140 .RE

142 .sp
143 .ne 2
144 .na
145 \fB\fIdirectory\fR\fB/unix.\fR\fIn\fR\fR
146 .ad
147 .RS 29n

149 .RE

151 .sp
152 .ne 2
153 .na
154 \fB\fIdirectory\fR\fB/bounds\fR\fR
155 .ad
156 .RS 29n

158 .RE

160 .sp
161 .ne 2
162 .na
163 \fB\fIdirectory\fR\fB/minfree\fR\fR
164 .ad
165 .RS 29n

167 .RE

169 .sp
170 .ne 2
171 .na
172 \fB\fB/var/crash/\&‘uname \fR\fB-n\fR\fB\&‘\fR\fR
173 .ad
174 .RS 29n
175 default crash dump directory
176 .RE

178 .SH SEE ALSO
170 .sp
179 .LP
180 \fBadb\fR(1), \fBmdb\fR(1), \fBsvcs\fR(1), \fBdd\fR(1M), \fBdumpadm\fR(1M),
181 \fBsvcadm\fR(1M), \fBsyslog\fR(3C), \fBattributes\fR(5), \fBsmf\fR(5)
```



```
182 .SH NOTES
175 .sp
183 .LP
184 The system crash dump service is managed by the service management facility,
185 \fBsmf\fR(5), under the service identifier:
186 .sp
187 .in +2
188 .nf
189 svc:/system/dumpadm:default
190 .fi
191 .in -2
192 .sp

194 .sp
195 .LP
196 Administrative actions on this service, such as enabling, disabling, or
197 requesting restart, can be performed using \fBsvcadm\fR(1M). The service’s
198 status can be queried using the \fBsvcs\fR(1) command.
199 .sp
200 .LP
201 If the dump device is also being used as a swap device, you must run
202 \fBsavecore\fR very soon after booting, before the swap space containing the
203 crash dump is overwritten by programs currently running.
```