```
**********************************************************
   79274 Fri Nov 29 20:18:34 2013
new/usr/src/cmd/mdb/common/mdb/mdb_cmds.c
4229 mdb hangs on exit when long umem cache names exist
Reviewed by: Robert Mustacchi <rm@joyent.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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright (c) 2013 Joyent, Inc. All rights reserved.
 * Copyright (c) 2013 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
#endif /* ! codereview */
 */

#include <sys/elf.h>
#include <sys/elf_SPARC.h>

#include <libproc.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <alloca.h>
#include <libctf.h>
#include <ctype.h>

#include <mdb/mdb_string.h>
#include <mdb/mdb_argvec.h>
#include <mdb/mdb_nv.h>
#include <mdb/mdb_fmt.h>
#include <mdb/mdb_target.h>
#include <mdb/mdb_err.h>
#include <mdb/mdb_debug.h>
#include <mdb/mdb_conf.h>
#include <mdb/mdb_module.h>
#include <mdb/mdb_modapi.h>
#include <mdb/mdb_stdlib.h>
#include <mdb/mdb_lex.h>
#include <mdb/mdb_io_impl.h>
#include <mdb/mdb_help.h>
#include <mdb/mdb_disasm.h>
```



```
#include <mdb/mdb_frame.h>
#include <mdb/mdb_evset.h>
#include <mdb/mdb_print.h>
#include <mdb/mdb_nm.h>
#include <mdb/mdb_set.h>
#include <mdb/mdb_demangle.h>
#include <mdb/mdb_ctf.h>
#include <mdb/mdb_whatis.h>
#include <mdb/mdb_whatis_impl.h>
#include <mdb/mdb_macalias.h>
#include <mdb/mdb_tab.h>
#include <mdb/mdb_typedef.h>
#ifdef _KMDB
#include <kmdb/kmdb_kdi.h>
#endif
#include <mdb/mdb.h>

#ifdef __sparc
#define SETHI_MASK      0xc1c00000
#define SETHI_VALUE     0x01000000

#define IS_SETHI(machcode)      (((machcode) & SETHI_MASK) == SETHI_VALUE)

#define OP(machcode)    ((machcode) >> 30)
#define OP3(machcode)   (((machcode) >> 19) & 0x3f)
#define RD(machcode)    (((machcode) >> 25) & 0x1f)
#define RS1(machcode)   (((machcode) >> 14) & 0x1f)
#define I(machcode)     (((machcode) >> 13) & 0x01)

#define IMM13(machcode) ((machcode) & 0x1fff)
#define IMM22(machcode) ((machcode) & 0x3fffff)

#define OP_ARITH_MEM_MASK       0x2
#define OP_ARITH                0x2
#define OP_MEM                  0x3

#define OP3_CC_MASK             0x10
#define OP3_COMPLEX_MASK        0x20

#define OP3_ADD                 0x00
#define OP3_OR                  0x02
#define OP3_XOR                 0x03

#ifndef R_O7
#define R_O7    0xf
#endif
#endif /* __sparc */

static mdb_tgt_addr_t
write_uint8(mdb_tgt_as_t as, mdb_tgt_addr_t addr, uint64_t ull, uint_t rdback)
{
        uint8_t o, n = (uint8_t)ull;

        if (rdback && mdb_tgt_aread(mdb.m_target, as, &o, sizeof (o),
            addr) == -1)
                return (addr);

        if (mdb_tgt_awrite(mdb.m_target, as, &n, sizeof (n), addr) == -1)
                return (addr);

        if (rdback) {
                if (mdb_tgt_aread(mdb.m_target, as, &n, sizeof (n), addr) == -1)
                        return (addr);

                mdb_iob_printf(mdb.m_out, "%-#*lla%16T%-#8x=%8T0x%x\n",
                    mdb_iob_getmargin(mdb.m_out), addr, o, n);
```

```
127         }
129         return (addr + sizeof (n));
130 }
132 static mdb_tgt_addr_t
133 write_uint16(mdb_tgt_as_t as, mdb_tgt_addr_t addr, uint64_t ull, uint_t rdback)
134 {
135         uint16_t o, n = (uint16_t)ull;
137         if (rdback && mdb_tgt_aread(mdb.m_target, as, &o, sizeof (o),
138             addr) == -1)
139                 return (addr);
141         if (mdb_tgt_awrite(mdb.m_target, as, &n, sizeof (n), addr) == -1)
142                 return (addr);
144         if (rdback) {
145                 if (mdb_tgt_aread(mdb.m_target, as, &n, sizeof (n), addr) == -1)
146                         return (addr);
148                 mdb_iob_printf(mdb.m_out, "%-#*lla%16T%-#8hx=%8T0x%hx\n",
149                     mdb_iob_getmargin(mdb.m_out), addr, o, n);
150         }
152         return (addr + sizeof (n));
153 }
155 static mdb_tgt_addr_t
156 write_uint32(mdb_tgt_as_t as, mdb_tgt_addr_t addr, uint64_t ull, uint_t rdback)
157 {
158         uint32_t o, n = (uint32_t)ull;
160         if (rdback && mdb_tgt_aread(mdb.m_target, as, &o, sizeof (o),
161             addr) == -1)
162                 return (addr);
164         if (mdb_tgt_awrite(mdb.m_target, as, &n, sizeof (n), addr) == -1)
165                 return (addr);
167         if (rdback) {
168                 if (mdb_tgt_aread(mdb.m_target, as, &n, sizeof (n), addr) == -1)
169                         return (addr);
171                 mdb_iob_printf(mdb.m_out, "%-#*lla%16T%-#16x=%8T0x%x\n",
172                     mdb_iob_getmargin(mdb.m_out), addr, o, n);
173         }
175         return (addr + sizeof (n));
176 }
178 static mdb_tgt_addr_t
179 write_uint64(mdb_tgt_as_t as, mdb_tgt_addr_t addr, uint64_t n, uint_t rdback)
180 {
181         uint64_t o;
183         if (rdback && mdb_tgt_aread(mdb.m_target, as, &o, sizeof (o),
184             addr) == -1)
185                 return (addr);
187         if (mdb_tgt_awrite(mdb.m_target, as, &n, sizeof (n), addr) == -1)
188                 return (addr);
190         if (rdback) {
191                 if (mdb_tgt_aread(mdb.m_target, as, &n, sizeof (n), addr) == -1)
192                         return (addr);
```



```
194                 mdb_iob_printf(mdb.m_out, "%-#*lla%16T%-#24llx=%8T0x%llx\n",
195                     mdb_iob_getmargin(mdb.m_out), addr, o, n);
196         }
198         return (addr + sizeof (n));
199 }
201 static int
202 write_arglist(mdb_tgt_as_t as, mdb_tgt_addr_t addr,
203     int argc, const mdb_arg_t *argv)
204 {
205         mdb_tgt_addr_t (*write_value)(mdb_tgt_as_t, mdb_tgt_addr_t,
206             uint64_t, uint_t);
207         mdb_tgt_addr_t naddr;
208         uintmax_t value;
209         int rdback = mdb.m_flags & MDB_FL_READBACK;
210         size_t i;
212         if (argc == 1) {
213                 mdb_warn("expected value to write following %c\n",
214                     argv->a_un.a_char);
215                 return (DCMD_ERR);
216         }
218         switch (argv->a_un.a_char) {
219         case 'v':
220                 write_value = write_uint8;
221                 break;
222         case 'w':
223                 write_value = write_uint16;
224                 break;
225         case 'W':
226                 write_value = write_uint32;
227                 break;
228         case 'Z':
229                 write_value = write_uint64;
230                 break;
231         }
233         for (argv++, i = 1; i < argc; i++, argv++) {
234                 if (argv->a_type == MDB_TYPE_CHAR) {
235                         mdb_warn("expected immediate value instead of '%c'\n",
236                             argv->a_un.a_char);
237                         return (DCMD_ERR);
238                 }
240                 if (argv->a_type == MDB_TYPE_STRING) {
241                         if (mdb_eval(argv->a_un.a_str) == -1) {
242                                 mdb_warn("failed to write \"%s\"",
243                                     argv->a_un.a_str);
244                                 return (DCMD_ERR);
245                         }
246                         value = mdb_nv_get_value(mdb.m_dot);
247                 } else
248                         value = argv->a_un.a_val;
250                 mdb_nv_set_value(mdb.m_dot, addr);
252                 if ((naddr = write_value(as, addr, value, rdback)) == addr) {
253                         mdb_warn("failed to write %llr at address 0x%llx",
254                             value, addr);
255                         mdb.m_incr = 0;
256                         break;
257                 }
```

```
259                 mdb.m_incr = naddr - addr;
260                 addr = naddr;
261         }

263         return (DCMD_OK);
264 }

266 static mdb_tgt_addr_t
267 match_uint16(mdb_tgt_as_t as, mdb_tgt_addr_t addr, uint64_t v64, uint64_t m64)
268 {
269         uint16_t x, val = (uint16_t)v64, mask = (uint16_t)m64;

271         for (; mdb_tgt_aread(mdb.m_target, as, &x,
272             sizeof (x), addr) == sizeof (x); addr += sizeof (x)) {

274                 if ((x & mask) == val) {
275                         mdb_iob_printf(mdb.m_out, "%lla\n", addr);
276                         break;
277                 }
278         }
279         return (addr);
280 }

282 static mdb_tgt_addr_t
283 match_uint32(mdb_tgt_as_t as, mdb_tgt_addr_t addr, uint64_t v64, uint64_t m64)
284 {
285         uint32_t x, val = (uint32_t)v64, mask = (uint32_t)m64;

287         for (; mdb_tgt_aread(mdb.m_target, as, &x,
288             sizeof (x), addr) == sizeof (x); addr += sizeof (x)) {

290                 if ((x & mask) == val) {
291                         mdb_iob_printf(mdb.m_out, "%lla\n", addr);
292                         break;
293                 }
294         }
295         return (addr);
296 }

298 static mdb_tgt_addr_t
299 match_uint64(mdb_tgt_as_t as, mdb_tgt_addr_t addr, uint64_t val, uint64_t mask)
300 {
301         uint64_t x;

303         for (; mdb_tgt_aread(mdb.m_target, as, &x,
304             sizeof (x), addr) == sizeof (x); addr += sizeof (x)) {

306                 if ((x & mask) == val) {
307                         mdb_iob_printf(mdb.m_out, "%lla\n", addr);
308                         break;
309                 }
310         }
311         return (addr);
312 }

314 static int
315 match_arglist(mdb_tgt_as_t as, uint_t flags, mdb_tgt_addr_t addr,
316     int argc, const mdb_arg_t *argv)
317 {
318         mdb_tgt_addr_t (*match_value)(mdb_tgt_as_t, mdb_tgt_addr_t,
319             uint64_t, uint64_t);

321         uint64_t args[2] = { 0, -1ULL }; /* [ value, mask ] */
322         size_t i;

324         if (argc < 2) {
```



```
325                 mdb_warn("expected value following %c\n", argv->a_un.a_char);
326                 return (DCMD_ERR);
327         }

329         if (argc > 3) {
330                 mdb_warn("only value and mask may follow %c\n",
331                     argv->a_un.a_char);
332                 return (DCMD_ERR);
333         }

335         switch (argv->a_un.a_char) {
336         case 'l':
337                 match_value = match_uint16;
338                 break;
339         case 'L':
340                 match_value = match_uint32;
341                 break;
342         case 'M':
343                 match_value = match_uint64;
344                 break;
345         }

347         for (argv++, i = 1; i < argc; i++, argv++) {
348                 if (argv->a_type == MDB_TYPE_CHAR) {
349                         mdb_warn("expected immediate value instead of '%c'\n",
350                             argv->a_un.a_char);
351                         return (DCMD_ERR);
352                 }

354                 if (argv->a_type == MDB_TYPE_STRING) {
355                         if (mdb_eval(argv->a_un.a_str) == -1) {
356                                 mdb_warn("failed to evaluate \"%s\"",
357                                     argv->a_un.a_str);
358                                 return (DCMD_ERR);
359                         }
360                         args[i - 1] = mdb_nv_get_value(mdb.m_dot);
361                 } else
362                         args[i - 1] = argv->a_un.a_val;
363         }

365         addr = match_value(as, addr, args[0], args[1]);
366         mdb_nv_set_value(mdb.m_dot, addr);

368         /*
369          * In adb(1), the match operators ignore any repeat count that has
370          * been applied to them.  We emulate this undocumented property
371          * by returning DCMD_ABORT if our input is not a pipeline.
372          */
373         return ((flags & DCMD_PIPE) ? DCMD_OK : DCMD_ABORT);
374 }

376 static int
377 argncmp(int argc, const mdb_arg_t *argv, const char *s)
378 {
379         for (; *s != '\0'; s++, argc--, argv++) {
380                 if (argc == 0 || argv->a_type != MDB_TYPE_CHAR)
381                         return (FALSE);
382                 if (argv->a_un.a_char != *s)
383                         return (FALSE);
384         }
385         return (TRUE);
386 }

388 static int
389 print_arglist(mdb_tgt_as_t as, mdb_tgt_addr_t addr, uint_t flags,
390     int argc, const mdb_arg_t *argv)
```

```
{
        char buf[MDB_TGT_SYM_NAMLEN];
        mdb_tgt_addr_t oaddr = addr;
        mdb_tgt_addr_t naddr;
        GElf_Sym sym;
        size_t i, n;

        if (DCMD_HDRSPEC(flags) && (flags & DCMD_PIPE_OUT) == 0) {
                const char *fmt;
                int is_dis;
                /*
                 * This is nasty, but necessary for precise adb compatibility.
                 * Detect disassembly format by looking for "ai" or "ia":
                 */
                if (argncmp(argc, argv, "ai")) {
                        fmt = "%-#*lla\n";
                        is_dis = TRUE;
                } else if (argncmp(argc, argv, "ia")) {
                        fmt = "%-#*lla";
                        is_dis = TRUE;
                } else {
                        fmt = "%-#*lla%16T";
                        is_dis = FALSE;
                }

                /*
                 * If symbolic decoding is on, disassembly is off, and the
                 * address exactly matches a symbol, print the symbol name:
                 */
                if ((mdb.m_flags & MDB_FL_PSYM) && !is_dis &&
                    (as == MDB_TGT_AS_VIRT || as == MDB_TGT_AS_FILE) &&
                    mdb_tgt_lookup_by_addr(mdb.m_target, (uintptr_t)addr,
                    MDB_TGT_SYM_EXACT, buf, sizeof (buf), &sym, NULL) == 0)
                        mdb_iob_printf(mdb.m_out, "%s:\n", buf);

                /*
                 * If this is a virtual address, cast it so that it reflects
                 * only the valid component of the address.
                 */
                if (as == MDB_TGT_AS_VIRT)
                        addr = (uintptr_t)addr;

                mdb_iob_printf(mdb.m_out, fmt,
                    (uint_t)mdb_iob_getmargin(mdb.m_out), addr);
        }

        if (argc == 0) {
                /*
                 * Yes, for you trivia buffs: if you use a format verb and give
                 * no format string, you get: X^"= "i ... note that in adb the
                 * the '=' verb once had 'z' as its default, but then 'z' was
                 * deleted (it was once an alias for 'i') and so =\n now calls
                 * scanform("z") and produces a 'bad modifier' message.
                 */
                static const mdb_arg_t def_argv[] = {
                        { MDB_TYPE_CHAR, MDB_INIT_CHAR('X') },
                        { MDB_TYPE_CHAR, MDB_INIT_CHAR('^') },
                        { MDB_TYPE_STRING, MDB_INIT_STRING("= ") },
                        { MDB_TYPE_CHAR, MDB_INIT_CHAR('i') }
                };

                argc = sizeof (def_argv) / sizeof (mdb_arg_t);
                argv = def_argv;
        }

        mdb_iob_setflags(mdb.m_out, MDB_IOB_INDENT);
```



```
        for (i = 0, n = 1; i < argc; i++, argv++) {
                switch (argv->a_type) {
                case MDB_TYPE_CHAR:
                        naddr = mdb_fmt_print(mdb.m_target, as, addr, n,
                            argv->a_un.a_char);
                        mdb.m_incr = naddr - addr;
                        addr = naddr;
                        n = 1;
                        break;

                case MDB_TYPE_IMMEDIATE:
                        n = argv->a_un.a_val;
                        break;

                case MDB_TYPE_STRING:
                        mdb_iob_puts(mdb.m_out, argv->a_un.a_str);
                        n = 1;
                        break;
                }
        }

        mdb.m_incr = addr - oaddr;
        mdb_iob_clrflags(mdb.m_out, MDB_IOB_INDENT);
        return (DCMD_OK);
}

static int
print_common(mdb_tgt_as_t as, uint_t flags, int argc, const mdb_arg_t *argv)
{
        mdb_tgt_addr_t addr = mdb_nv_get_value(mdb.m_dot);

        if (argc != 0 && argv->a_type == MDB_TYPE_CHAR) {
                if (strchr("vwWZ", argv->a_un.a_char))
                        return (write_arglist(as, addr, argc, argv));
                if (strchr("lLM", argv->a_un.a_char))
                        return (match_arglist(as, flags, addr, argc, argv));
        }

        return (print_arglist(as, addr, flags, argc, argv));
}

/*ARGSUSED*/
static int
cmd_print_core(uintptr_t x, uint_t flags, int argc, const mdb_arg_t *argv)
{
        return (print_common(MDB_TGT_AS_VIRT, flags, argc, argv));
}

#ifndef _KMDB
/*ARGSUSED*/
static int
cmd_print_object(uintptr_t x, uint_t flags, int argc, const mdb_arg_t *argv)
{
        return (print_common(MDB_TGT_AS_FILE, flags, argc, argv));
}
#endif

/*ARGSUSED*/
static int
cmd_print_phys(uintptr_t x, uint_t flags, int argc, const mdb_arg_t *argv)
{
        return (print_common(MDB_TGT_AS_PHYS, flags, argc, argv));
}

/*ARGSUSED*/
```

```
523 static int
524 cmd_print_value(uintptr_t addr, uint_t flags,
525         int argc, const mdb_arg_t *argv)
526 {
527         uintmax_t ndot, dot = mdb_get_dot();
528         const char *tgt_argv[1];
529         mdb_tgt_t *t;
530         size_t i, n;

532         if (argc == 0) {
533                 mdb_warn("expected one or more format characters "
534                     "following '='\n");
535                 return (DCMD_ERR);
536         }

538         tgt_argv[0] = (const char *)&dot;
539         t = mdb_tgt_create(mdb_value_tgt_create, 0, 1, tgt_argv);
540         mdb_iob_setflags(mdb.m_out, MDB_IOB_INDENT);

542         for (i = 0, n = 1; i < argc; i++, argv++) {
543                 switch (argv->a_type) {
544                 case MDB_TYPE_CHAR:
545                         ndot = mdb_fmt_print(t, MDB_TGT_AS_VIRT,
546                             dot, n, argv->a_un.a_char);
547                         if (argv->a_un.a_char == '+' ||
548                             argv->a_un.a_char == '-')
549                                 dot = ndot;
550                         n = 1;
551                         break;

553                 case MDB_TYPE_IMMEDIATE:
554                         n = argv->a_un.a_val;
555                         break;

557                 case MDB_TYPE_STRING:
558                         mdb_iob_puts(mdb.m_out, argv->a_un.a_str);
559                         n = 1;
560                         break;
561                 }
562         }

564         mdb_iob_clrflags(mdb.m_out, MDB_IOB_INDENT);
565         mdb_nv_set_value(mdb.m_dot, dot);
566         mdb.m_incr = 0;

568         mdb_tgt_destroy(t);
569         return (DCMD_OK);
570 }

572 /*ARGSUSED*/
573 static int
574 cmd_assign_variable(uintptr_t addr, uint_t flags,
575     int argc, const mdb_arg_t *argv)
576 {
577         uintmax_t dot = mdb_nv_get_value(mdb.m_dot);
578         const char *p;
579         mdb_var_t *v;

581         if (argc == 2) {
582                 if (argv->a_type != MDB_TYPE_CHAR) {
583                         mdb_warn("improper arguments following '>' operator\n");
584                         return (DCMD_ERR);
585                 }

587                 switch (argv->a_un.a_char) {
588                 case 'c':
```



```
589                         addr = *((uchar_t *)&addr);
590                         break;
591                 case 's':
592                         addr = *((ushort_t *)&addr);
593                         break;
594                 case 'i':
595                         addr = *((uint_t *)&addr);
596                         break;
597                 case 'l':
598                         addr = *((ulong_t *)&addr);
599                         break;
600                 default:
601                         mdb_warn("%c is not a valid // modifier\n",
602                             argv->a_un.a_char);
603                         return (DCMD_ERR);
604                 }

606                 dot = addr;
607                 argv++;
608                 argc--;
609         }

611         if (argc != 1 || argv->a_type != MDB_TYPE_STRING) {
612                 mdb_warn("expected single variable name following '>'\n");
613                 return (DCMD_ERR);
614         }

616         if (strlen(argv->a_un.a_str) >= (size_t)MDB_NV_NAMELEN) {
617                 mdb_warn("variable names may not exceed %d characters\n",
618                     MDB_NV_NAMELEN - 1);
619                 return (DCMD_ERR);
620         }

622         if ((p = strbadid(argv->a_un.a_str)) != NULL) {
623                 mdb_warn("'%c' may not be used in a variable name\n", *p);
624                 return (DCMD_ERR);
625         }

627         if ((v = mdb_nv_lookup(&mdb.m_nv, argv->a_un.a_str)) == NULL)
628                 (void) mdb_nv_insert(&mdb.m_nv, argv->a_un.a_str, NULL, dot, 0);
629         else
630                 mdb_nv_set_value(v, dot);

632         mdb.m_incr = 0;
633         return (DCMD_OK);
634 }

636 static int
637 print_soutype(const char *sou, uintptr_t addr, uint_t flags)
638 {
639         static const char *prefixes[] = { "struct ", "union " };
640         size_t namesz = 7 + strlen(sou) + 1;
641         char *name = mdb_alloc(namesz, UM_SLEEP | UM_GC);
642         mdb_ctf_id_t id;
643         int i;

645         for (i = 0; i < 2; i++) {
646                 (void) mdb_snprintf(name, namesz, "%s%s", prefixes[i], sou);

648                 if (mdb_ctf_lookup_by_name(name, &id) == 0) {
649                         mdb_arg_t v;
650                         int rv;

652                         v.a_type = MDB_TYPE_STRING;
653                         v.a_un.a_str = name;
```

```
655                         rv = mdb_call_dcmd("print", addr, flags, 1, &v);
656                         return (rv);
657                 }
658         }

660         return (DCMD_ERR);
661 }

663 static int
664 print_type(const char *name, uintptr_t addr, uint_t flags)
665 {
666         mdb_ctf_id_t id;
667         char *sname;
668         size_t snamesz;
669         int rv;

671         if (!(flags & DCMD_ADDRSPEC)) {
672                 addr = mdb_get_dot();
673                 flags |= DCMD_ADDRSPEC;
674         }

676         if ((rv = print_soutype(name, addr, flags)) != DCMD_ERR)
677                 return (rv);

679         snamesz = strlen(name) + 3;
680         sname = mdb_zalloc(snamesz, UM_SLEEP | UM_GC);
681         (void) mdb_snprintf(sname, snamesz, "%s_t", name);

683         if (mdb_ctf_lookup_by_name(sname, &id) == 0) {
684                 mdb_arg_t v;
685                 int rv;

687                 v.a_type = MDB_TYPE_STRING;
688                 v.a_un.a_str = sname;

690                 rv = mdb_call_dcmd("print", addr, flags, 1, &v);
691                 return (rv);
692         }

694         sname[snamesz - 2] = 's';
695         rv = print_soutype(sname, addr, flags);
696         return (rv);
697 }

699 static int
700 exec_alias(const char *fname, uintptr_t addr, uint_t flags)
701 {
702         const char *alias;
703         int rv;

705         if ((alias = mdb_macalias_lookup(fname)) == NULL)
706                 return (DCMD_ERR);

708         if (flags & DCMD_ADDRSPEC) {
709                 size_t sz = sizeof (uintptr_t) * 2 + strlen(alias) + 1;
710                 char *addralias = mdb_alloc(sz, UM_SLEEP | UM_GC);
711                 (void) mdb_snprintf(addralias, sz, "%p%s", addr, alias);
712                 rv = mdb_eval(addralias);
713         } else {
714                 rv = mdb_eval(alias);
715         }

717         return (rv == -1 ? DCMD_ABORT : DCMD_OK);
718 }

720 /*ARGSUSED*/
```



```
721 static int
722 cmd_src_file(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
723 {
724         const char *fname;
725         mdb_io_t *fio;
726         int rv;

728         if (argc != 1 || argv->a_type != MDB_TYPE_STRING)
729                 return (DCMD_USAGE);

731         fname = argv->a_un.a_str;

733         if (flags & DCMD_PIPE_OUT) {
734                 mdb_warn("macro files cannot be used as input to a pipeline\n");
735                 return (DCMD_ABORT);
736         }

738         if ((fio = mdb_fdio_create_path(mdb.m_ipath, fname,
739             O_RDONLY, 0)) != NULL) {
740                 mdb_frame_t *fp = mdb.m_frame;
741                 int err;

743                 mdb_iob_stack_push(&fp->f_istk, mdb.m_in, yylineno);
744                 mdb.m_in = mdb_iob_create(fio, MDB_IOB_RDONLY);
745                 err = mdb_run();

747                 ASSERT(fp == mdb.m_frame);
748                 mdb.m_in = mdb_iob_stack_pop(&fp->f_istk);
749                 yylineno = mdb_iob_lineno(mdb.m_in);

751                 if (err == MDB_ERR_PAGER && mdb.m_fmark != fp)
752                         longjmp(fp->f_pcb, err);

754                 if (err == MDB_ERR_QUIT || err == MDB_ERR_ABORT ||
755                     err == MDB_ERR_SIGINT || err == MDB_ERR_OUTPUT)
756                         longjmp(fp->f_pcb, err);

758                 return (DCMD_OK);
759         }

761         if ((rv = exec_alias(fname, addr, flags)) != DCMD_ERR ||
762             (rv = print_type(fname, addr, flags)) != DCMD_ERR)
763                 return (rv);

765         mdb_warn("failed to open %s (see ::help '$<')\n", fname);
766         return (DCMD_ABORT);
767 }

769 static int
770 cmd_exec_file(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
771 {
772         const char *fname;
773         mdb_io_t *fio;
774         int rv;

776         /*
777          * The syntax [expr[,count]]$< with no trailing macro file name is
778          * magic in that if count is zero, this command won't be called and
779          * the expression is thus a no-op.  If count is non-zero, we get
780          * invoked with argc == 0, and this means abort the current macro.
781          * If our debugger stack depth is greater than one, we may be using
782          * $< from within a previous $<<, so in that case we set m_in to
783          * NULL to force this entire frame to be popped.
784          */
785         if (argc == 0) {
786                 if (mdb_iob_stack_size(&mdb.m_frame->f_istk) != 0) {
```

```
787                         mdb_iob_destroy(mdb.m_in);
788                         mdb.m_in = mdb_iob_stack_pop(&mdb.m_frame->f_istk);
789                 } else if (mdb.m_depth > 1) {
790                         mdb_iob_destroy(mdb.m_in);
791                         mdb.m_in = NULL;
792                 } else
793                         mdb_warn("input stack is empty\n");
794                 return (DCMD_OK);
795         }

797         if ((flags & (DCMD_PIPE | DCMD_PIPE_OUT)) || mdb.m_depth == 1)
798                 return (cmd_src_file(addr, flags, argc, argv));

800         if (argc != 1 || argv->a_type != MDB_TYPE_STRING)
801                 return (DCMD_USAGE);

803         fname = argv->a_un.a_str;

805         if ((fio = mdb_fdio_create_path(mdb.m_ipath, fname,
806             O_RDONLY, 0)) != NULL) {
807                 mdb_iob_destroy(mdb.m_in);
808                 mdb.m_in = mdb_iob_create(fio, MDB_IOB_RDONLY);
809                 return (DCMD_OK);
810         }

812         if ((rv = exec_alias(fname, addr, flags)) != DCMD_ERR ||
813             (rv = print_type(fname, addr, flags)) != DCMD_ERR)
814                 return (rv);

816         mdb_warn("failed to open %s (see ::help '$<')\n", fname);
817         return (DCMD_ABORT);
818 }

820 #ifndef _KMDB
821 /*ARGSUSED*/
822 static int
823 cmd_cat(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
824 {
825         int status = DCMD_OK;
826         char buf[BUFSIZ];
827         mdb_iob_t *iob;
828         mdb_io_t *fio;

830         if (flags & DCMD_ADDRSPEC)
831                 return (DCMD_USAGE);

833         for (; argc-- != 0; argv++) {
834                 if (argv->a_type != MDB_TYPE_STRING) {
835                         mdb_warn("expected string argument\n");
836                         status = DCMD_ERR;
837                         continue;
838                 }

840                 if ((fio = mdb_fdio_create_path(NULL,
841                     argv->a_un.a_str, O_RDONLY, 0)) == NULL) {
842                         mdb_warn("failed to open %s", argv->a_un.a_str);
843                         status = DCMD_ERR;
844                         continue;
845                 }

847                 iob = mdb_iob_create(fio, MDB_IOB_RDONLY);

849                 while (!(mdb_iob_getflags(iob) & (MDB_IOB_EOF | MDB_IOB_ERR))) {
850                         ssize_t len = mdb_iob_read(iob, buf, sizeof (buf));
851                         if (len > 0) {
852                                 if (mdb_iob_write(mdb.m_out, buf, len) < 0) {
```



```
853                                         if (errno != EPIPE)
854                                                 mdb_warn("write failed");
855                                         status = DCMD_ERR;
856                                         break;
857                                 }
858                         }
859                 }

861                 if (mdb_iob_err(iob))
862                         mdb_warn("error while reading %s", mdb_iob_name(iob));

864                 mdb_iob_destroy(iob);
865         }

867         return (status);
868 }
869 #endif

871 /*ARGSUSED*/
872 static int
873 cmd_grep(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
874 {
875         if (argc != 1 || argv->a_type != MDB_TYPE_STRING)
876                 return (DCMD_USAGE);

878         if (mdb_eval(argv->a_un.a_str) == -1)
879                 return (DCMD_ABORT);

881         if (mdb_get_dot() != 0)
882                 mdb_printf("%lr\n", addr);

884         return (DCMD_OK);
885 }

887 /*ARGSUSED*/
888 static int
889 cmd_map(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
890 {
891         if (argc != 1 || argv->a_type != MDB_TYPE_STRING)
892                 return (DCMD_USAGE);

894         if (mdb_eval(argv->a_un.a_str) == -1)
895                 return (DCMD_ABORT);

897         mdb_printf("%llr\n", mdb_get_dot());
898         return (DCMD_OK);
899 }

901 /*ARGSUSED*/
902 static int
903 cmd_notsup(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
904 {
905         mdb_warn("command is not supported by current target\n");
906         return (DCMD_ERR);
907 }

909 /*ARGSUSED*/
910 static int
911 cmd_quit(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
912 {
913 #ifdef _KMDB
914         uint_t opt_u = FALSE;

916         if (mdb_getopts(argc, argv,
917             'u', MDB_OPT_SETBITS, TRUE, &opt_u, NULL) != argc)
918                 return (DCMD_USAGE);
```

```
920         if (opt_u) {
921                 if (mdb.m_flags & MDB_FL_NOUNLOAD) {
922                         warn("%s\n", mdb_strerror(EMDB_KNOUNLOAD));
923                         return (DCMD_ERR);
924                 }

926                 kmdb_kdi_set_unload_request();
927         }
928 #endif

930         longjmp(mdb.m_frame->f_pcb, MDB_ERR_QUIT);
931         /*NOTREACHED*/
932         return (DCMD_ERR);
933 }

935 #ifdef _KMDB
936 static void
937 quit_help(void)
938 {
939         mdb_printf(
940             "-u    unload the debugger (if not loaded at boot)\n");
941 }
942 #endif

944 /*ARGSUSED*/
945 static int
946 cmd_vars(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
947 {
948         uint_t opt_nz = FALSE, opt_tag = FALSE, opt_prt = FALSE;
949         mdb_var_t *v;

951         if (mdb_getopts(argc, argv,
952             'n', MDB_OPT_SETBITS, TRUE, &opt_nz,
953             'p', MDB_OPT_SETBITS, TRUE, &opt_prt,
954             't', MDB_OPT_SETBITS, TRUE, &opt_tag, NULL) != argc)
955                 return (DCMD_USAGE);

957         mdb_nv_rewind(&mdb.m_nv);

959         while ((v = mdb_nv_advance(&mdb.m_nv)) != NULL) {
960                 if ((opt_tag == FALSE || (v->v_flags & MDB_NV_TAGGED)) &&
961                     (opt_nz == FALSE || mdb_nv_get_value(v) != 0)) {
962                         if (opt_prt) {
963                                 mdb_printf("%#llr>%s\n",
964                                     mdb_nv_get_value(v), mdb_nv_get_name(v));
965                         } else {
966                                 mdb_printf("%s = %llr\n",
967                                     mdb_nv_get_name(v), mdb_nv_get_value(v));
968                         }
969                 }
970         }

972         return (DCMD_OK);
973 }

975 /*ARGSUSED*/
976 static int
977 cmd_nzvars(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
978 {
979         uintmax_t value;
980         mdb_var_t *v;

982         if (argc != 0)
983                 return (DCMD_USAGE);
```



```
985         mdb_nv_rewind(&mdb.m_nv);

987         while ((v = mdb_nv_advance(&mdb.m_nv)) != NULL) {
988                 if ((value = mdb_nv_get_value(v)) != 0)
989                         mdb_printf("%s = %llr\n", mdb_nv_get_name(v), value);
990         }

992         return (DCMD_OK);
993 }

995 /*ARGSUSED*/
996 static int
997 cmd_radix(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
998 {
999         if (argc != 0)
1000                 return (DCMD_USAGE);

1002        if (flags & DCMD_ADDRSPEC) {
1003                if (addr < 2 || addr > 16) {
1004                        mdb_warn("expected radix from 2 to 16\n");
1005                        return (DCMD_ERR);
1006                }
1007                mdb.m_radix = (int)addr;
1008        }

1010        mdb_iob_printf(mdb.m_out, "radix = %d base ten\n", mdb.m_radix);
1011        return (DCMD_OK);
1012 }

1014 /*ARGSUSED*/
1015 static int
1016 cmd_symdist(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1017 {
1018        if (argc != 0)
1019                return (DCMD_USAGE);

1021        if (flags & DCMD_ADDRSPEC)
1022                mdb.m_symdist = addr;

1024        mdb_printf("symbol matching distance = %lr (%s)\n",
1025            mdb.m_symdist, mdb.m_symdist ? "absolute mode" : "smart mode");

1027        return (DCMD_OK);
1028 }

1030 /*ARGSUSED*/
1031 static int
1032 cmd_pgwidth(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1033 {
1034        if (argc != 0)
1035                return (DCMD_USAGE);

1037        if (flags & DCMD_ADDRSPEC)
1038                mdb_iob_resize(mdb.m_out, mdb.m_out->iob_rows, addr);

1040        mdb_printf("output page width = %lu\n", mdb.m_out->iob_cols);
1041        return (DCMD_OK);
1042 }

1044 /*ARGSUSED*/
1045 static int
1046 cmd_reopen(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1047 {
1048        if (argc != 0)
1049                return (DCMD_USAGE);
```

```
1051         if (mdb_tgt_setflags(mdb.m_target, MDB_TGT_F_RDWR) == -1) {
1052                 mdb_warn("failed to re-open target for writing");
1053                 return (DCMD_ERR);
1054         }

1056         return (DCMD_OK);
1057 }

1059 /*ARGSUSED*/
1060 static int
1061 print_xdata(void *ignored, const char *name, const char *desc, size_t nbytes)
1062 {
1063         mdb_printf("%-24s - %s (%lu bytes)\n", name, desc, (ulong_t)nbytes);
1064         return (0);
1065 }

1067 /*ARGSUSED*/
1068 static int
1069 cmd_xdata(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1070 {
1071         if (argc != 0 || (flags & DCMD_ADDRSPEC))
1072                 return (DCMD_USAGE);

1074         (void) mdb_tgt_xdata_iter(mdb.m_target, print_xdata, NULL);
1075         return (DCMD_OK);
1076 }

1078 /*ARGSUSED*/
1079 static int
1080 cmd_unset(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1081 {
1082         mdb_var_t *v;
1083         size_t i;

1085         for (i = 0; i < argc; i++) {
1086                 if (argv[i].a_type != MDB_TYPE_STRING) {
1087                         mdb_warn("bad option: arg %lu is not a string\n",
1088                             (ulong_t)i + 1);
1089                         return (DCMD_USAGE);
1090                 }
1091         }

1093         for (i = 0; i < argc; i++, argv++) {
1094                 if ((v = mdb_nv_lookup(&mdb.m_nv, argv->a_un.a_str)) == NULL)
1095                         mdb_warn("variable '%s' not defined\n",
1096                             argv->a_un.a_str);
1097                 else
1098                         mdb_nv_remove(&mdb.m_nv, v);
1099         }

1101         return (DCMD_OK);
1102 }

1104 #ifndef _KMDB
1105 /*ARGSUSED*/
1106 static int
1107 cmd_log(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1108 {
1109         uint_t opt_e = FALSE, opt_d = FALSE;
1110         const char *filename = NULL;
1111         int i;

1113         i = mdb_getopts(argc, argv,
1114             'd', MDB_OPT_SETBITS, TRUE, &opt_d,
1115             'e', MDB_OPT_SETBITS, TRUE, &opt_e, NULL);
```



```
1117         if ((i != argc && i != argc - 1) || (opt_d && opt_e) ||
1118             (i != argc && argv[i].a_type != MDB_TYPE_STRING) ||
1119             (i != argc && opt_d == TRUE) || (flags & DCMD_ADDRSPEC))
1120                 return (DCMD_USAGE);

1122         if (mdb.m_depth != 1) {
1123                 mdb_warn("log may not be manipulated in this context\n");
1124                 return (DCMD_ABORT);
1125         }

1127         if (i != argc)
1128                 filename = argv[i].a_un.a_str;

1130         /*
1131          * If no arguments were specified, print the log file name (if any)
1132          * and report whether the log is enabled or disabled.
1133          */
1134         if (argc == 0) {
1135                 if (mdb.m_log) {
1136                         mdb_printf("%s: logging to \"%s\" is currently %s\n",
1137                             mdb.m_pname, IOP_NAME(mdb.m_log),
1138                             mdb.m_flags & MDB_FL_LOG ?  "enabled" : "disabled");
1139                 } else
1140                         mdb_printf("%s: no log is active\n", mdb.m_pname);
1141                 return (DCMD_OK);
1142         }

1144         /*
1145          * If the -d option was specified, pop the log i/o object off the
1146          * i/o stack of stdin, stdout, and stderr.
1147          */
1148         if (opt_d) {
1149                 if (mdb.m_flags & MDB_FL_LOG) {
1150                         (void) mdb_iob_pop_io(mdb.m_in);
1151                         (void) mdb_iob_pop_io(mdb.m_out);
1152                         (void) mdb_iob_pop_io(mdb.m_err);
1153                         mdb.m_flags &= ~MDB_FL_LOG;
1154                 } else
1155                         mdb_warn("logging is already disabled\n");
1156                 return (DCMD_OK);
1157         }

1159         /*
1160          * The -e option is the default: (re-)enable logging by pushing
1161          * the log i/o object on to stdin, stdout, and stderr.  If we have
1162          * a previous log file, we need to pop it and close it.  If we have
1163          * no new log file, push the previous one back on.
1164          */
1165         if (filename != NULL) {
1166                 if (mdb.m_log != NULL) {
1167                         if (mdb.m_flags & MDB_FL_LOG) {
1168                                 (void) mdb_iob_pop_io(mdb.m_in);
1169                                 (void) mdb_iob_pop_io(mdb.m_out);
1170                                 (void) mdb_iob_pop_io(mdb.m_err);
1171                                 mdb.m_flags &= ~MDB_FL_LOG;
1172                         }
1173                         mdb_io_rele(mdb.m_log);
1174                 }

1176                 mdb.m_log = mdb_fdio_create_path(NULL, filename,
1177                     O_CREAT | O_APPEND | O_WRONLY, 0666);

1179                 if (mdb.m_log == NULL) {
1180                         mdb_warn("failed to open %s", filename);
1181                         return (DCMD_ERR);
1182                 }
```

```
1183         }

1185         if (mdb.m_log != NULL) {
1186                 mdb_iob_push_io(mdb.m_in, mdb_logio_create(mdb.m_log));
1187                 mdb_iob_push_io(mdb.m_out, mdb_logio_create(mdb.m_log));
1188                 mdb_iob_push_io(mdb.m_err, mdb_logio_create(mdb.m_log));

1190                 mdb_printf("%s: logging to \"%s\"\n", mdb.m_pname, filename);
1191                 mdb.m_log = mdb_io_hold(mdb.m_log);
1192                 mdb.m_flags |= MDB_FL_LOG;

1194                 return (DCMD_OK);
1195         }

1197         mdb_warn("no log file has been selected\n");
1198         return (DCMD_ERR);
1199 }

1201 static int
1202 cmd_old_log(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1203 {
1204         if (argc == 0) {
1205                 mdb_arg_t arg = { MDB_TYPE_STRING, MDB_INIT_STRING("-d") };
1206                 return (cmd_log(addr, flags, 1, &arg));
1207         }

1209         return (cmd_log(addr, flags, argc, argv));
1210 }
1211 #endif

1213 /*ARGSUSED*/
1214 static int
1215 cmd_load(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1216 {
1217         int i, mode = MDB_MOD_LOCAL;

1219         i = mdb_getopts(argc, argv,
1220 #ifdef _KMDB
1221             'd', MDB_OPT_SETBITS, MDB_MOD_DEFER, &mode,
1222 #endif
1223             'f', MDB_OPT_SETBITS, MDB_MOD_FORCE, &mode,
1224             'g', MDB_OPT_SETBITS, MDB_MOD_GLOBAL, &mode,
1225             's', MDB_OPT_SETBITS, MDB_MOD_SILENT, &mode,
1226             NULL);

1228         argc -= i;
1229         argv += i;

1231         if ((flags & DCMD_ADDRSPEC) || argc != 1 ||
1232             argv->a_type != MDB_TYPE_STRING ||
1233             strchr("+-", argv->a_un.a_str[0]) != NULL)
1234                 return (DCMD_USAGE);

1236         if (mdb_module_load(argv->a_un.a_str, mode) < 0)
1237                 return (DCMD_ERR);

1239         return (DCMD_OK);
1240 }

1242 static void
1243 load_help(void)
1244 {
1245         mdb_printf(
1246 #ifdef _KMDB
1247             "-d    defer load until next continue\n"
1248 #endif
```



```
1249             "-s    load module silently\n");
1250 }

1252 /*ARGSUSED*/
1253 static int
1254 cmd_unload(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1255 {
1256         int mode = 0;
1257         int i;

1259         i = mdb_getopts(argc, argv,
1260 #ifdef _KMDB
1261             'd', MDB_OPT_SETBITS, MDB_MOD_DEFER, &mode,
1262 #endif
1263             NULL);

1265         argc -= i;
1266         argv += i;

1268         if (argc != 1 || argv->a_type != MDB_TYPE_STRING)
1269                 return (DCMD_USAGE);

1271         if (mdb_module_unload(argv->a_un.a_str, mode) == -1) {
1272                 mdb_warn("failed to unload %s", argv->a_un.a_str);
1273                 return (DCMD_ERR);
1274         }

1276         return (DCMD_OK);
1277 }

1279 #ifdef _KMDB
1280 static void
1281 unload_help(void)
1282 {
1283         mdb_printf(
1284             "-d    defer unload until next continue\n");
1285 }
1286 #endif

1288 static int
1289 cmd_dbmode(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1290 {
1291         if (argc > 1 || (argc != 0 && (flags & DCMD_ADDRSPEC)))
1292                 return (DCMD_USAGE);

1294         if (argc != 0) {
1295                 if (argv->a_type != MDB_TYPE_STRING)
1296                         return (DCMD_USAGE);
1297                 if ((addr = mdb_dstr2mode(argv->a_un.a_str)) != MDB_DBG_HELP)
1298                         mdb_dmode(addr);
1299         } else if (flags & DCMD_ADDRSPEC)
1300                 mdb_dmode(addr);

1302         mdb_printf("debugging mode = 0x%04x\n", mdb.m_debug);
1303         return (DCMD_OK);
1304 }

1306 /*ARGSUSED*/
1307 static int
1308 cmd_version(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1309 {
1310 #ifdef DEBUG
1311         mdb_printf("\r%s (DEBUG)\n", mdb_conf_version());
1312 #else
1313         mdb_printf("\r%s\n", mdb_conf_version());
1314 #endif
```

```
1315         return (DCMD_OK);
1316 }

1318 /*ARGSUSED*/
1319 static int
1320 cmd_algol(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1321 {
1322         if (mdb.m_flags & MDB_FL_ADB)
1323                 mdb_printf("No algol 68 here\n");
1324         else
1325                 mdb_printf("No adb here\n");
1326         return (DCMD_OK);
1327 }

1329 /*ARGSUSED*/
1330 static int
1331 cmd_obey(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1332 {
1333         if (mdb.m_flags & MDB_FL_ADB)
1334                 mdb_printf("CHAPTER 1\n");
1335         else
1336                 mdb_printf("No Language H here\n");
1337         return (DCMD_OK);
1338 }

1340 /*ARGSUSED*/
1341 static int
1342 print_global(void *data, const GElf_Sym *sym, const char *name,
1343     const mdb_syminfo_t *sip, const char *obj)
1344 {
1345         uintptr_t value;

1347         if (mdb_tgt_vread((mdb_tgt_t *)data, &value, sizeof (value),
1348             (uintptr_t)sym->st_value) == sizeof (value))
1349                 mdb_printf("%s(%llr):\t%lr\n", name, sym->st_value, value);
1350         else
1351                 mdb_printf("%s(%llr):\t?\n", name, sym->st_value);

1353         return (0);
1354 }

1356 /*ARGSUSED*/
1357 static int
1358 cmd_globals(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1359 {
1360         if (argc != 0)
1361                 return (DCMD_USAGE);

1363         (void) mdb_tgt_symbol_iter(mdb.m_target, MDB_TGT_OBJ_EVERY,
1364             MDB_TGT_SYMTAB, MDB_TGT_BIND_GLOBAL | MDB_TGT_TYPE_OBJECT |
1365             MDB_TGT_TYPE_FUNC, print_global, mdb.m_target);

1367         return (0);
1368 }

1370 /*ARGSUSED*/
1371 static int
1372 cmd_eval(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1373 {
1374         if (argc != 1 || argv->a_type != MDB_TYPE_STRING)
1375                 return (DCMD_USAGE);

1377         if (mdb_eval(argv->a_un.a_str) == -1)
1378                 return (DCMD_ABORT);

1380         return (DCMD_OK);
```



```
1381 }

1383 /*ARGSUSED*/
1384 static int
1385 print_file(void *data, const GElf_Sym *sym, const char *name,
1386     const mdb_syminfo_t *sip, const char *obj)
1387 {
1388         int i = *((int *)data);

1390         mdb_printf("%d\t%s\n", i++, name);
1391         *((int *)data) = i;
1392         return (0);
1393 }

1395 /*ARGSUSED*/
1396 static int
1397 cmd_files(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1398 {
1399         int i = 1;
1400         const char *obj = MDB_TGT_OBJ_EVERY;

1402         if ((flags & DCMD_ADDRSPEC) || argc > 1)
1403                 return (DCMD_USAGE);

1405         if (argc == 1) {
1406                 if (argv->a_type != MDB_TYPE_STRING)
1407                         return (DCMD_USAGE);

1409                 obj = argv->a_un.a_str;
1410         }

1412         (void) mdb_tgt_symbol_iter(mdb.m_target, obj, MDB_TGT_SYMTAB,
1413             MDB_TGT_BIND_ANY | MDB_TGT_TYPE_FILE, print_file, &i);

1415         return (DCMD_OK);
1416 }

1418 static const char *
1419 map_name(const mdb_map_t *map, const char *name)
1420 {
1421         if (map->map_flags & MDB_TGT_MAP_HEAP)
1422                 return ("[ heap ]");
1423         if (name != NULL && name[0] != 0)
1424                 return (name);

1426         if (map->map_flags & MDB_TGT_MAP_SHMEM)
1427                 return ("[ shmem ]");
1428         if (map->map_flags & MDB_TGT_MAP_STACK)
1429                 return ("[ stack ]");
1430         if (map->map_flags & MDB_TGT_MAP_ANON)
1431                 return ("[ anon ]");
1432         if (map->map_name != NULL)
1433                 return (map->map_name);
1434         return ("[ unknown ]");
1435 }

1437 /*ARGSUSED*/
1438 static int
1439 print_map(void *ignored, const mdb_map_t *map, const char *name)
1440 {
1441         name = map_name(map, name);

1443         mdb_printf("%?p %?p %?lx %s\n", map->map_base,
1444             map->map_base + map->map_size, map->map_size, name);
1445         return (0);
1446 }
```

```
1448 static int
1449 cmd_mappings(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1450 {
1451         const mdb_map_t *m;

1453         if (argc > 1 || (argc != 0 && (flags & DCMD_ADDRSPEC)))
1454                 return (DCMD_USAGE);

1456         mdb_printf("%<u>%?s %?s %?s %s%</u>\n",
1457             "BASE", "LIMIT", "SIZE", "NAME");

1459         if (flags & DCMD_ADDRSPEC) {
1460                 if ((m = mdb_tgt_addr_to_map(mdb.m_target, addr)) == NULL)
1461                         mdb_warn("failed to obtain mapping");
1462                 else
1463                         (void) print_map(NULL, m, NULL);

1465         } else if (argc != 0) {
1466                 if (argv->a_type == MDB_TYPE_STRING)
1467                         m = mdb_tgt_name_to_map(mdb.m_target, argv->a_un.a_str);
1468                 else
1469                         m = mdb_tgt_addr_to_map(mdb.m_target, argv->a_un.a_val);

1471                 if (m == NULL)
1472                         mdb_warn("failed to obtain mapping");
1473                 else
1474                         (void) print_map(NULL, m, NULL);

1476         } else if (mdb_tgt_mapping_iter(mdb.m_target, print_map, NULL) == -1)
1477                 mdb_warn("failed to iterate over mappings");

1479         return (DCMD_OK);
1480 }

1482 static int
1483 whatis_map_callback(void *wp, const mdb_map_t *map, const char *name)
1484 {
1485         mdb_whatis_t *w = wp;
1486         uintptr_t cur;

1488         name = map_name(map, name);

1490         while (mdb_whatis_match(w, map->map_base, map->map_size, &cur))
1491                 mdb_whatis_report_address(w, cur, "in %s [%p,%p)\n",
1492                     name, map->map_base, map->map_base + map->map_size);

1494         return (0);
1495 }

1497 /*ARGSUSED*/
1498 int
1499 whatis_run_mappings(mdb_whatis_t *w, void *ignored)
1500 {
1501         (void) mdb_tgt_mapping_iter(mdb.m_target, whatis_map_callback, w);
1502         return (0);
1503 }

1505 /*ARGSUSED*/
1506 static int
1507 objects_printversion(void *ignored, const mdb_map_t *map, const char *name)
1508 {
1509         ctf_file_t *ctfp;
1510         const char *version;

1512         ctfp = mdb_tgt_name_to_ctf(mdb.m_target, name);
```



```
1513         if (ctfp == NULL || (version = ctf_label_topmost(ctfp)) == NULL)
1514                 version = "Unknown";

1516         mdb_printf("%-28s %s\n", name, version);
1517         return (0);
1518 }

1520 /*ARGSUSED*/
1521 static int
1522 cmd_objects(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1523 {
1524         uint_t opt_v = FALSE;
1525         mdb_tgt_map_f *cb;

1527         if ((flags & DCMD_ADDRSPEC) || mdb_getopts(argc, argv,
1528             'v', MDB_OPT_SETBITS, TRUE, &opt_v, NULL) != argc)
1529                 return (DCMD_USAGE);

1531         if (opt_v) {
1532                 cb = objects_printversion;
1533                 mdb_printf("%<u>%-28s %s%</u>\n", "NAME", "VERSION");
1534         } else {
1535                 cb = print_map;
1536                 mdb_printf("%<u>%?s %?s %?s %s%</u>\n",
1537                     "BASE", "LIMIT", "SIZE", "NAME");
1538         }

1540         if (mdb_tgt_object_iter(mdb.m_target, cb, NULL) == -1) {
1541                 mdb_warn("failed to iterate over objects");
1542                 return (DCMD_ERR);
1543         }

1545         return (DCMD_OK);
1546 }

1548 /*ARGSUSED*/
1549 static int
1550 showrev_addversion(void *vers_nv, const mdb_map_t *ignored, const char *object)
1551 {
1552         ctf_file_t *ctfp;
1553         const char *version = NULL;
1554         char *objname;

1556         objname = mdb_alloc(strlen(object) + 1, UM_SLEEP | UM_GC);
1557         (void) strcpy(objname, object);

1559         if ((ctfp = mdb_tgt_name_to_ctf(mdb.m_target, objname)) != NULL)
1560                 version = ctf_label_topmost(ctfp);

1562         /*
1563          * Not all objects have CTF and label data, so set version to "Unknown".
1564          */
1565         if (version == NULL)
1566                 version = "Unknown";

  30         /*
  31          * The hash table implementation in OVERLOAD mode limits the version
  32          * name to 31 characters because we cannot specify an external name.
  33          * The full version name is available via the ::objects dcmd if needed.
  34          */
1568         (void) mdb_nv_insert(vers_nv, version, NULL, (uintptr_t)objname,
1569             MDB_NV_OVERLOAD);

1571         return (0);
1572 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   10234 Fri Nov 29 20:18:35 2013
new/usr/src/cmd/mdb/common/mdb/mdb_nv.c
4229 mdb hangs on exit when long umem cache names exist
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (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 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright (c) 2013 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <mdb/mdb_debug.h>
#include <mdb/mdb_string.h>
#include <mdb/mdb_modapi.h>
#include <mdb/mdb_err.h>
#include <mdb/mdb_nv.h>
#include <mdb/mdb.h>

#define NV_NAME(v) \
        (((v)->v_flags & MDB_NV_EXTNAME) ? (v)->v_ename : (v)->v_lname)

#define NV_SIZE(v) \
        (((v)->v_flags & MDB_NV_EXTNAME) ? sizeof (mdb_var_t) : \
        sizeof (mdb_var_t) + MDB_NV_NAMELEN - 1)

#define NV_HASHSZ       211

static size_t
nv_hashstring(const char *key)
{
        size_t g, h = 0;
        const char *p;

        ASSERT(key != NULL);

        for (p = key; *p != '\0'; p++) {
                h = (h << 4) + *p;

                if ((g = (h & 0xf0000000)) != 0) {
```



```
                        h ^= (g >> 24);
                        h ^= g;
                }
        }

        return (h);
}

static mdb_var_t *
nv_var_alloc(const char *name, const mdb_nv_disc_t *disc,
        uintmax_t value, uint_t flags, uint_t um_flags, mdb_var_t *next)
{
        size_t nbytes;
        mdb_var_t *v;
        size_t nbytes = (flags & MDB_NV_EXTNAME) ? sizeof (mdb_var_t) :
            (sizeof (mdb_var_t) + MDB_NV_NAMELEN - 1);

        if (flags & MDB_NV_EXTNAME)
                nbytes = sizeof (mdb_var_t);
        else
                nbytes = sizeof (mdb_var_t) + strlen(name);

        v = mdb_alloc(nbytes, um_flags);
        mdb_var_t *v = mdb_alloc(nbytes, um_flags);

        if (v == NULL)
                return (NULL);

        if (flags & MDB_NV_EXTNAME) {
                v->v_ename = name;
                v->v_lname[0] = '\0';
                v->v_lname[0] = 0;
        } else {
                /*
                 * We don't overflow here since the mdb_var_t itself has
                 * room for the trailing \0.
                 */
                (void) strcpy(v->v_lname, name);
                (void) strncpy(v->v_lname, name, MDB_NV_NAMELEN - 1);
                v->v_lname[MDB_NV_NAMELEN - 1] = '\0';
                v->v_ename = NULL;
        }

        v->v_uvalue = value;
        v->v_flags = flags & ~(MDB_NV_SILENT | MDB_NV_INTERPOS);
        v->v_disc = disc;
        v->v_next = next;

        return (v);
}
______unchanged_portion_omitted_
```

```
**********************************************************
    5313 Fri Nov 29 20:18:35 2013
new/usr/src/cmd/mdb/common/mdb/mdb_nv.h
4229 mdb hangs on exit when long umem cache names exist
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (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 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright (c) 2013 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
 */
#endif /* ! codereview */

#ifndef _MDB_NV_H
#define _MDB_NV_H

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/types.h>

#ifdef  __cplusplus
extern "C" {
#endif

#ifdef _MDB

/*
 * There used to be a cap (MDB_NV_NAMELEN bytes including null) on the
 * length of variable names stored in-line.  This cap is no longer there,
 * however parts of mdb use the constant to sanitize input.
 */
#endif /* ! codereview */
#define MDB_NV_NAMELEN  31      /* Max variable name length including null */

/*
 * These flags are stored inside each variable in v_flags:
 */
#define MDB_NV_PERSIST  0x01    /* Variable is persistent (cannot be unset) */
#define MDB_NV_RDONLY   0x02    /* Variable is read-only (cannot insert over) */
#define MDB_NV_EXTNAME  0x04    /* Variable name is stored externally */
#define MDB_NV_TAGGED   0x08    /* Variable is tagged (user-defined) */
#define MDB_NV_OVERLOAD 0x10    /* Variable can be overloaded (multiple defs) */
```



```
/*
 * These flags may be passed to mdb_nv_insert() but are not stored
 * inside the variable (and thus use bits outside of 0x00 - 0xff):
 */
#define MDB_NV_SILENT   0x100   /* Silence warnings about existing defs */
#define MDB_NV_INTERPOS 0x200   /* Interpose definition over previous defs */

struct mdb_var;                 /* Forward declaration */
struct mdb_walk_state;          /* Forward declaration */

/*
 * Each variable's behavior with respect to the get-value and set-value
 * operations can be changed using a discipline: a pointer to an ops
 * vector which can re-define these operations:
 */
typedef struct mdb_nv_disc {
        void (*disc_set)(struct mdb_var *, uintmax_t);
        uintmax_t (*disc_get)(const struct mdb_var *);
} mdb_nv_disc_t;

/*
 * Each variable is defined by the following variable-length structure.
 * The debugger uses name/value collections to hash almost everything, so
 * we make a few simple space optimizations:
 *
 * A variable's name can be a pointer to external storage (v_ename and
 * MDB_NV_EXTNAME set), or it can be stored locally (bytes of storage are
 * allocated immediately after v_lname[0]).
 * MDB_NV_EXTNAME set), or it can be stored locally (MDB_NV_NAMELEN - 1
 * bytes of storage are allocated immediately after v_lname[0]).
 *
 * A variable may have multiple definitions (v_ndef chain), but this feature
 * is mutually exclusive with MDB_NV_EXTNAME in order to save space.
 */
typedef struct mdb_var {
        uintmax_t v_uvalue;             /* Value as unsigned integral type */
        union {
                const char *v_ename;    /* Variable name if stored externally */
                struct mdb_var *v_ndef; /* Variable's next definition */
        } v_du;
        const mdb_nv_disc_t *v_disc;    /* Link to variable discipline */
        struct mdb_var *v_next;         /* Link to next var in hash chain */
        uchar_t v_flags;                /* Variable flags (see above) */
        char v_lname[1];                /* Variable name if stored locally */
} mdb_var_t;
______unchanged_portion_omitted_
```

```
**********************************************************
   14873 Fri Nov 29 20:18:35 2013
new/usr/src/cmd/mdb/common/mdb/mdb_tab.c
4229 mdb hangs on exit when long umem cache names exist
Reviewed by: Robert Mustacchi <rm@joyent.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) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 Joyent, Inc. All rights reserved.
 * Copyright (c) 2013 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
#endif /* ! codereview */
 */
/*
 * This file contains all of the interfaces for mdb's tab completion engine.
 * Currently some interfaces are private to mdb and its internal implementation,
 * those are in mdb_tab.h. Other pieces are public interfaces. Those are in
 * mdb_modapi.h.
 *
 * Memory allocations in tab completion context have to be done very carefully.
 * We need to think of ourselves as the same as any other command that is being
 * executed by the user, which means we must use UM_GC to handle being
 * interrupted.
 */

#include <mdb/mdb_modapi.h>
#include <mdb/mdb_ctf.h>
#include <mdb/mdb_ctf_impl.h>
#include <mdb/mdb_string.h>
#include <mdb/mdb_module.h>
#include <mdb/mdb_debug.h>
#include <mdb/mdb_print.h>
#include <mdb/mdb_nv.h>
#include <mdb/mdb_tab.h>
#include <mdb/mdb_target.h>
#include <mdb/mdb.h>

#include <ctype.h>

/*
 * There may be another way to do this, but this works well enough.
 */
#define COMMAND_SEPARATOR "::"

/*
 * find_command_start --
 *
```



```
 *      Given a buffer find the start of the last command.
 */
static char *
tab_find_command_start(char *buf)
{
        char *offset = strstr(buf, COMMAND_SEPARATOR);

        if (offset == NULL)
                return (NULL);

        for (;;) {
                char *next = strstr(offset + strlen(COMMAND_SEPARATOR),
                    COMMAND_SEPARATOR);

                if (next == NULL) {
                        return (offset);
                }

                offset = next;
        }
}

/*
 * get_dcmd --
 *
 *      Given a buffer containing a command and its argument return
 *      the name of the command and the offset in the buffer where
 *      the command arguments start.
 *
 *      Note: This will modify the buffer.
 */
char *
tab_get_dcmd(char *buf, char **args, uint_t *flags)
{
        char *start = buf + strlen(COMMAND_SEPARATOR);
        char *separator = start;
        const char *end = buf + strlen(buf);
        uint_t space = 0;

        while (separator < end && !isspace(*separator))
                separator++;

        if (separator == end) {
                *args = NULL;
        } else {
                if (isspace(*separator))
                        space = 1;

                *separator++ = '\0';
                *args = separator;
        }

        if (space)
                *flags |= DCMD_TAB_SPACE;

        return (start);
}

/*
 * count_args --
 *
 *      Given a buffer containing dmcd arguments return the total number
 *      of arguments.
 *
 *      While parsing arguments we need to keep track of whether or not the last
 *      arguments ends with a trailing space.
```

```
 */
static int
tab_count_args(const char *input, uint_t *flags)
{
        const char *index;
        int argc = 0;
        uint_t space = *flags & DCMD_TAB_SPACE;
        index = input;

        while (*index != '\0') {
                while (*index != '\0' && isspace(*index)) {
                        index++;
                        space = 1;
                }

                if (*index != '\0' && !isspace(*index)) {
                        argc++;
                        space = 0;
                        while (*index != '\0' && !isspace (*index)) {
                                index++;
                        }
                }
        }

        if (space)
                *flags |= DCMD_TAB_SPACE;
        else
                *flags &= ~DCMD_TAB_SPACE;

        return (argc);
}

/*
 * copy_args --
 *
 *      Given a buffer containing dcmd arguments and an array of mdb_arg_t's
 *      initialize the string value of each mdb_arg_t.
 *
 *      Note: This will modify the buffer.
 */
static int
tab_copy_args(char *input, int argc, mdb_arg_t *argv)
{
        int i = 0;
        char *index;

        index = input;

        while (*index) {
                while (*index && isspace(*index)) {
                        index++;
                }

                if (*index && !isspace(*index)) {
                        char *end = index;

                        while (*end && !isspace(*end)) {
                                end++;
                        }

                        if (*end) {
                                *end++ = '\0';
                        }

                        argv[i].a_type = MDB_TYPE_STRING;
                        argv[i].a_un.a_str = index;
```



```
                        index = end;
                        i++;
                }
        }

        if (i != argc)
                return (-1);

        return (0);
}

/*
 * parse-buf --
 *
 *      Parse the given buffer and return the specified dcmd, the number
 *      of arguments, and array of mdb_arg_t containing the argument
 *      values.
 *
 *      Note: this will modify the specified buffer. Caller is responisble
 *      for freeing argvp.
 */
static int
tab_parse_buf(char *buf, char **dcmdp, int *argcp, mdb_arg_t **argvp,
    uint_t *flags)
{
        char *data = tab_find_command_start(buf);
        char *args_data = NULL;
        char *dcmd = NULL;
        int argc = 0;
        mdb_arg_t *argv = NULL;

        if (data == NULL) {
                return (-1);
        }

        dcmd = tab_get_dcmd(data, &args_data, flags);

        if (dcmd == NULL) {
                return (-1);
        }

        if (args_data != NULL) {
                argc = tab_count_args(args_data, flags);

                if (argc != 0) {
                        argv = mdb_alloc(sizeof (mdb_arg_t) * argc,
                            UM_SLEEP | UM_GC);

                        if (tab_copy_args(args_data, argc, argv) == -1)
                                return (-1);
                }
        }

        *dcmdp = dcmd;
        *argcp = argc;
        *argvp = argv;

        return (0);
}

/*
 * tab_command --
 *
 *      This function is executed anytime a tab is entered. It checks
 *      the current buffer to determine if there is a valid dmcd,
```

```
 *      if that dcmd has a tab completion handler it will invoke it.
 *
 *      This function returns the string (if any) that should be added to the
 *      existing buffer to complete it.
 */
int
mdb_tab_command(mdb_tab_cookie_t *mcp, const char *buf)
{
        char *data;
        char *dcmd = NULL;
        int argc = 0;
        mdb_arg_t *argv = NULL;
        int ret = 0;
        mdb_idcmd_t *cp;
        uint_t flags = 0;

        /*
         * Parsing the command and arguments will modify the buffer
         * (replacing spaces with \0), so make a copy of the specified
         * buffer first.
         */
        data = mdb_alloc(strlen(buf) + 1, UM_SLEEP | UM_GC);
        (void) strcpy(data, buf);

        /*
         * Get the specified dcmd and arguments from the buffer.
         */
        ret = tab_parse_buf(data, &dcmd, &argc, &argv, &flags);

        /*
         * Match against global symbols if the input is not a dcmd
         */
        if (ret != 0) {
                (void) mdb_tab_complete_global(mcp, buf);
                goto out;
        }

        /*
         * Check to see if the buffer contains a valid dcmd
         */
        cp = mdb_dcmd_lookup(dcmd);

        /*
         * When argc is zero it indicates that we are trying to tab complete
         * a dcmd or a global symbol. Note, that if there isn't the start of
         * a dcmd, i.e. ::, then we will have already bailed in the call to
         * tab_parse_buf.
         */
        if (cp == NULL && argc != 0) {
                goto out;
        }

        /*
         * Invoke the command specific tab completion handler or the built in
         * dcmd one if there is no dcmd.
         */
        if (cp == NULL)
                (void) mdb_tab_complete_dcmd(mcp, dcmd);
        else
                mdb_call_tab(cp, mcp, flags, argc, argv);

out:
        return (mdb_tab_size(mcp));
}

static int
```



```
tab_complete_dcmd(mdb_var_t *v, void *arg)
{
        mdb_idcmd_t *idcp = mdb_nv_get_cookie(mdb_nv_get_cookie(v));
        mdb_tab_cookie_t *mcp = (mdb_tab_cookie_t *)arg;

        /*
         * The way that mdb is implemented, even commands like $C will show up
         * here. As such, we don't want to match anything that doesn't start
         * with an alpha or number. While nothing currently appears (via a
         * cursory search with mdb -k) to start with a capital letter or a
         * number, we'll support them anyways.
         */
        if (!isalnum(idcp->idc_name[0]))
                return (0);

        mdb_tab_insert(mcp, idcp->idc_name);
        return (0);
}

int
mdb_tab_complete_dcmd(mdb_tab_cookie_t *mcp, const char *dcmd)
{
        mdb_tab_setmbase(mcp, dcmd);
        mdb_nv_sort_iter(&mdb.m_dcmds, tab_complete_dcmd, mcp,
            UM_GC | UM_SLEEP);
        return (0);
}

static int
tab_complete_walker(mdb_var_t *v, void *arg)
{
        mdb_iwalker_t *iwp = mdb_nv_get_cookie(mdb_nv_get_cookie(v));
        mdb_tab_cookie_t *mcp = arg;

        mdb_tab_insert(mcp, iwp->iwlk_name);
        return (0);
}

int
mdb_tab_complete_walker(mdb_tab_cookie_t *mcp, const char *walker)
{
        if (walker != NULL)
                mdb_tab_setmbase(mcp, walker);
        mdb_nv_sort_iter(&mdb.m_walkers, tab_complete_walker, mcp,
            UM_GC | UM_SLEEP);

        return (0);
}

mdb_tab_cookie_t *
mdb_tab_init(void)
{
        mdb_tab_cookie_t *mcp;

        mcp = mdb_zalloc(sizeof (mdb_tab_cookie_t), UM_SLEEP | UM_GC);
        (void) mdb_nv_create(&mcp->mtc_nv, UM_SLEEP | UM_GC);

        return (mcp);
}

size_t
mdb_tab_size(mdb_tab_cookie_t *mcp)
{
        return (mdb_nv_size(&mcp->mtc_nv));
}
```

```
391 /*
392  * Determine whether the specified name is a valid tab completion for
393  * the given command. If the name is a valid tab completion then
394  * it will be saved in the mdb_tab_cookie_t.
395  */
396 void
397 mdb_tab_insert(mdb_tab_cookie_t *mcp, const char *name)
398 {
399         size_t matches, index;
 24         size_t len, matches, index;
 25         uint_t flags;
400         mdb_var_t *v;
 27         char *n;
 28         const char *nvn;

402         /*
403          * If we have a match set, then we want to verify that we actually match
404          * it.
405          */
406         if (mcp->mtc_base != NULL &&
407             strncmp(name, mcp->mtc_base, strlen(mcp->mtc_base)) != 0)
408                 return;

410         v = mdb_nv_lookup(&mcp->mtc_nv, name);
411         if (v != NULL)
412                 return;

414         (void) mdb_nv_insert(&mcp->mtc_nv, name, NULL, 0, MDB_NV_RDONLY);
 42         /*
 43          * Names that we get passed in may be longer than MDB_NV_NAMELEN which
 44          * is currently 31 including the null terminator. If that is the case,
 45          * then we're going to take care of allocating a string and holding it
 46          * for our caller. Note that we don't need to free it, because we're
 47          * allocating this with UM_GC.
 48          */
 49         flags = 0;
 50         len = strlen(name);
 51         if (len > MDB_NV_NAMELEN - 1) {
 52                 n = mdb_alloc(len + 1, UM_SLEEP | UM_GC);
 53                 (void) strcpy(n, name);
 54                 nvn = n;
 55                 flags |= MDB_NV_EXTNAME;
 56         } else {
 57                 nvn = name;
 58         }
 59         flags |= MDB_NV_RDONLY;

 61         (void) mdb_nv_insert(&mcp->mtc_nv, nvn, NULL, 0, flags);

416         matches = mdb_tab_size(mcp);
417         if (matches == 1) {
418                 (void) strlcpy(mcp->mtc_match, name, MDB_SYM_NAMLEN);
 65                 (void) strlcpy(mcp->mtc_match, nvn, MDB_SYM_NAMLEN);
419         } else {
420                 index = 0;
421                 while (mcp->mtc_match[index] &&
422                     mcp->mtc_match[index] == name[index])
 69                     mcp->mtc_match[index] == nvn[index])
423                         index++;

425                 mcp->mtc_match[index] = '\0';
426         }
427 }
```

_____***unchanged_portion_omitted_***