```
**********************************************************
   21424 Mon Aug  5 12:25:08 2013
new/usr/src/cmd/mdb/common/modules/genunix/findstack.c
3992 mdb ::stacks segv
**********************************************************
```

```
/*
 * 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) 1999, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013, Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
#endif /* ! codereview */
 */

#include <mdb/mdb_modapi.h>
#include <mdb/mdb_ctf.h>

#include <sys/types.h>
#include <sys/regset.h>
#include <sys/stack.h>
#include <sys/thread.h>
#include <sys/modctl.h>
#include <assert.h>

#include "findstack.h"
#include "thread.h"
#include "sobj.h"

int findstack_debug_on = 0;

/*
 * "sp" is a kernel VA.
 */
static int
print_stack(uintptr_t sp, uintptr_t pc, uintptr_t addr,
    int argc, const mdb_arg_t *argv, int free_state)
{
        int showargs = 0, count, err;

        count = mdb_getopts(argc, argv,
            'v', MDB_OPT_SETBITS, TRUE, &showargs, NULL);
        argc -= count;
        argv += count;

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

        mdb_printf("stack pointer for thread %p%s: %p\n",
```



```
            addr, (free_state ? " (TS_FREE)" : ""), sp);
        if (pc != 0)
                mdb_printf("[ %0?lr %a() ]\n", sp, pc);

        mdb_inc_indent(2);
        mdb_set_dot(sp);

        if (argc == 1)
                err = mdb_eval(argv->a_un.a_str);
        else if (showargs)
                err = mdb_eval("<.$C");
        else
                err = mdb_eval("<.$C0");

        mdb_dec_indent(2);

        return ((err == -1) ? DCMD_ABORT : DCMD_OK);
}

int
findstack(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
{
        findstack_info_t fsi;
        int retval;

        if (!(flags & DCMD_ADDRSPEC))
                return (DCMD_USAGE);

        bzero(&fsi, sizeof (fsi));

        if ((retval = stacks_findstack(addr, &fsi, 1)) != DCMD_OK ||
            fsi.fsi_failed)
                return (retval);

        return (print_stack(fsi.fsi_sp, fsi.fsi_pc, addr,
            argc, argv, fsi.fsi_tstate == TS_FREE));
}

/*ARGSUSED*/
int
findstack_debug(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *av)
{
        findstack_debug_on ^= 1;

        mdb_printf("findstack: debugging is now %s\n",
            findstack_debug_on ? "on" : "off");

        return (DCMD_OK);
}

static void
uppercase(char *p)
{
        for (; *p != '\0'; p++) {
                if (*p >= 'a' && *p <= 'z')
                        *p += 'A' - 'a';
        }
}

static void
sobj_to_text(uintptr_t addr, char *out, size_t out_sz)
{
        sobj_ops_to_text(addr, out, out_sz);
        uppercase(out);
}
```

```
#define SOBJ_ALL        1

static int
text_to_sobj(const char *text, uintptr_t *out)
{
        if (strcasecmp(text, "ALL") == 0) {
                *out = SOBJ_ALL;
                return (0);
        }

        return (sobj_text_to_ops(text, out));
}

#define TSTATE_PANIC    -2U
static int
text_to_tstate(const char *text, uint_t *out)
{
        if (strcasecmp(text, "panic") == 0)
                *out = TSTATE_PANIC;
        else if (thread_text_to_state(text, out) != 0) {
                mdb_warn("tstate \"%s\" not recognized\n", text);
                return (-1);
        }
        return (0);
}

static void
tstate_to_text(uint_t tstate, uint_t paniced, char *out, size_t out_sz)
{
        if (paniced)
                mdb_snprintf(out, out_sz, "panic");
        else
                thread_state_to_text(tstate, out, out_sz);
        uppercase(out);
}

typedef struct stacks_entry {
        struct stacks_entry     *se_next;
        struct stacks_entry     *se_dup;        /* dups of this stack */
        uintptr_t               se_thread;
        uintptr_t               se_sp;
        uintptr_t               se_sobj_ops;
        uint32_t                se_tstate;
        uint32_t                se_count;       /* # threads w/ this stack */
        uint8_t                 se_overflow;
        uint8_t                 se_depth;
        uint8_t                 se_failed;      /* failure reason; FSI_FAIL_* */
        uint8_t                 se_panic;
        uintptr_t               se_stack[1];
} stacks_entry_t;
#define STACKS_ENTRY_SIZE(x) OFFSETOF(stacks_entry_t, se_stack[(x)])

#define STACKS_HSIZE 127

/* Maximum stack depth reported in stacks */
#define STACKS_MAX_DEPTH        254

typedef struct stacks_info {
        size_t          si_count;       /* total stacks_entry_ts (incl dups) */
        size_t          si_entries;     /* # entries in hash table */
        stacks_entry_t  **si_hash;      /* hash table */
        findstack_info_t si_fsi;        /* transient callback state */
} stacks_info_t;

/* global state cached between invocations */
#define STACKS_STATE_CLEAN      0
```



```
#define STACKS_STATE_DIRTY      1
#define STACKS_STATE_DONE       2
static uint_t stacks_state = STACKS_STATE_CLEAN;
static stacks_entry_t **stacks_hash;
static stacks_entry_t **stacks_array;
static size_t stacks_array_size;

size_t
stacks_hash_entry(stacks_entry_t *sep)
{
        size_t depth = sep->se_depth;
        uintptr_t *stack = sep->se_stack;

        uint64_t total = depth;

        while (depth > 0) {
                total += *stack;
                stack++; depth--;
        }

        return (total % STACKS_HSIZE);
}

/*
 * This is used to both compare stacks for equality and to sort the final
 * list of unique stacks.  forsort specifies the latter behavior, which
 * additionally:
 *      compares se_count, and
 *      sorts the stacks by text function name.
 *
 * The equality test is independent of se_count, and doesn't care about
 * relative ordering, so we don't do the extra work of looking up symbols
 * for the stack addresses.
 */
int
stacks_entry_comp_impl(stacks_entry_t *l, stacks_entry_t *r,
    uint_t forsort)
{
        int idx;

        int depth = MIN(l->se_depth, r->se_depth);

        /* no matter what, panic stacks come last. */
        if (l->se_panic > r->se_panic)
                return (1);
        if (l->se_panic < r->se_panic)
                return (-1);

        if (forsort) {
                /* put large counts earlier */
                if (l->se_count > r->se_count)
                        return (-1);
                if (l->se_count < r->se_count)
                        return (1);
        }

        if (l->se_tstate > r->se_tstate)
                return (1);
        if (l->se_tstate < r->se_tstate)
                return (-1);

        if (l->se_failed > r->se_failed)
                return (1);
        if (l->se_failed < r->se_failed)
                return (-1);
```

```
260         for (idx = 0; idx < depth; idx++) {
261                 char lbuf[MDB_SYM_NAMLEN];
262                 char rbuf[MDB_SYM_NAMLEN];

264                 int rval;
265                 uintptr_t laddr = l->se_stack[idx];
266                 uintptr_t raddr = r->se_stack[idx];

268                 if (laddr == raddr)
269                         continue;

271                 if (forsort &&
272                     mdb_lookup_by_addr(laddr, MDB_SYM_FUZZY,
273                     lbuf, sizeof (lbuf), NULL) != -1 &&
274                     mdb_lookup_by_addr(raddr, MDB_SYM_FUZZY,
275                     rbuf, sizeof (rbuf), NULL) != -1 &&
276                     (rval = strcmp(lbuf, rbuf)) != 0)
277                         return (rval);

279                 if (laddr > raddr)
280                         return (1);
281                 return (-1);
282         }

284         if (l->se_overflow > r->se_overflow)
285                 return (-1);
286         if (l->se_overflow < r->se_overflow)
287                 return (1);

289         if (l->se_depth > r->se_depth)
290                 return (1);
291         if (l->se_depth < r->se_depth)
292                 return (-1);

294         if (l->se_sobj_ops > r->se_sobj_ops)
295                 return (1);
296         if (l->se_sobj_ops < r->se_sobj_ops)
297                 return (-1);

299         return (0);
300 }

302 int
303 stacks_entry_comp(const void *l_arg, const void *r_arg)
304 {
305         stacks_entry_t * const *lp = l_arg;
306         stacks_entry_t * const *rp = r_arg;

308         return (stacks_entry_comp_impl(*lp, *rp, 1));
309 }

311 void
312 stacks_cleanup(int force)
313 {
314         int idx = 0;
315         stacks_entry_t *cur, *next;

317         if (stacks_state == STACKS_STATE_CLEAN)
318                 return;

320         if (!force && stacks_state == STACKS_STATE_DONE)
321                 return;

323         /*
324          * Until the array is sorted and stable, stacks_hash will be non-NULL.
325          * This way, we can get at all of the data, even if qsort() was
```



```
326          * interrupted while mucking with the array.
327          */
328         if (stacks_hash != NULL) {
329                 for (idx = 0; idx < STACKS_HSIZE; idx++) {
330                         while ((cur = stacks_hash[idx]) != NULL) {
331                                 while ((next = cur->se_dup) != NULL) {
332                                         cur->se_dup = next->se_dup;
333                                         mdb_free(next,
334                                             STACKS_ENTRY_SIZE(next->se_depth));
335                                 }
336                                 next = cur->se_next;
337                                 stacks_hash[idx] = next;
338                                 mdb_free(cur, STACKS_ENTRY_SIZE(cur->se_depth));
339                         }
340                 }
341                 if (stacks_array != NULL)
342                         mdb_free(stacks_array,
343                             stacks_array_size * sizeof (*stacks_array));

345                 mdb_free(stacks_hash, STACKS_HSIZE * sizeof (*stacks_hash));

347 #endif /* ! codereview */
348         } else if (stacks_array != NULL) {
349                 for (idx = 0; idx < stacks_array_size; idx++) {
350                         if ((cur = stacks_array[idx]) != NULL) {
351                                 while ((next = cur->se_dup) != NULL) {
352                                         cur->se_dup = next->se_dup;
353                                         mdb_free(next,
354                                             STACKS_ENTRY_SIZE(next->se_depth));
355                                 }
356                                 stacks_array[idx] = NULL;
357                                 mdb_free(cur, STACKS_ENTRY_SIZE(cur->se_depth));
358                         }
359                 }
360                 mdb_free(stacks_array,
361                     stacks_array_size * sizeof (*stacks_array));
362         }

364         stacks_findstack_cleanup();

366         stacks_array_size = 0;
367         stacks_state = STACKS_STATE_CLEAN;
368         stacks_hash = NULL;
369         stacks_array = NULL;
370 #endif /* ! codereview */
371 }

373 /*ARGSUSED*/
374 int
375 stacks_thread_cb(uintptr_t addr, const void *ignored, void *cbarg)
376 {
377         stacks_info_t *sip = cbarg;
378         findstack_info_t *fsip = &sip->si_fsi;

380         stacks_entry_t **sepp, *nsep, *sep;
381         int idx;
382         size_t depth;

384         if (stacks_findstack(addr, fsip, 0) != DCMD_OK &&
385             fsip->fsi_failed == FSI_FAIL_BADTHREAD) {
386                 mdb_warn("couldn't read thread at %p\n", addr);
387                 return (WALK_NEXT);
388         }

390         sip->si_count++;
```

```
392         depth = fsip->fsi_depth;
393         nsep = mdb_zalloc(STACKS_ENTRY_SIZE(depth), UM_SLEEP);
394         nsep->se_thread = addr;
395         nsep->se_sp = fsip->fsi_sp;
396         nsep->se_sobj_ops = fsip->fsi_sobj_ops;
397         nsep->se_tstate = fsip->fsi_tstate;
398         nsep->se_count = 1;
399         nsep->se_overflow = fsip->fsi_overflow;
400         nsep->se_depth = depth;
401         nsep->se_failed = fsip->fsi_failed;
402         nsep->se_panic = fsip->fsi_panic;

404         for (idx = 0; idx < depth; idx++)
405                 nsep->se_stack[idx] = fsip->fsi_stack[idx];

407         for (sepp = &sip->si_hash[stacks_hash_entry(nsep)];
408             (sep = *sepp) != NULL;
409             sepp = &sep->se_next) {

411                 if (stacks_entry_comp_impl(sep, nsep, 0) != 0)
412                         continue;

414                 nsep->se_dup = sep->se_dup;
415                 sep->se_dup = nsep;
416                 sep->se_count++;
417                 return (WALK_NEXT);
418         }

420         nsep->se_next = NULL;
421         *sepp = nsep;
422         sip->si_entries++;

424         return (WALK_NEXT);
425 }

427 int
428 stacks_run_tlist(mdb_pipe_t *tlist, stacks_info_t *si)
429 {
430         size_t idx;
431         size_t found = 0;
432         int ret;

434         for (idx = 0; idx < tlist->pipe_len; idx++) {
435                 uintptr_t addr = tlist->pipe_data[idx];

437                 found++;

439                 ret = stacks_thread_cb(addr, NULL, si);
440                 if (ret == WALK_DONE)
441                         break;
442                 if (ret != WALK_NEXT)
443                         return (-1);
444         }

446         if (found)
447                 return (0);
448         return (-1);
449 }

451 int
452 stacks_run(int verbose, mdb_pipe_t *tlist)
453 {
454         stacks_info_t si;
455         findstack_info_t *fsip = &si.si_fsi;
456         size_t idx;
457         stacks_entry_t **cur;
```



```
459         bzero(&si, sizeof (si));

461         stacks_state = STACKS_STATE_DIRTY;

463         stacks_hash = si.si_hash =
464             mdb_zalloc(STACKS_HSIZE * sizeof (*si.si_hash), UM_SLEEP);
465         si.si_entries = 0;
466         si.si_count = 0;

468         fsip->fsi_max_depth = STACKS_MAX_DEPTH;
469         fsip->fsi_stack =
470             mdb_alloc(fsip->fsi_max_depth * sizeof (*fsip->fsi_stack),
471             UM_SLEEP | UM_GC);

473         if (verbose)
474                 mdb_warn("stacks: processing kernel threads\n");

476         if (tlist != NULL) {
477                 if (stacks_run_tlist(tlist, &si))
478                         return (DCMD_ERR);
479         } else {
480                 if (mdb_walk("thread", stacks_thread_cb, &si) != 0) {
481                         mdb_warn("cannot walk \"thread\"");
482                         return (DCMD_ERR);
483                 }
484         }

486         if (verbose)
487                 mdb_warn("stacks: %d unique stacks / %d threads\n",
488                     si.si_entries, si.si_count);

490         stacks_array_size = si.si_entries;
491         stacks_array =
492             mdb_zalloc(si.si_entries * sizeof (*stacks_array), UM_SLEEP);
493         cur = stacks_array;
494         for (idx = 0; idx < STACKS_HSIZE; idx++) {
495                 stacks_entry_t *sep;
496                 for (sep = si.si_hash[idx]; sep != NULL; sep = sep->se_next)
497                         *(cur++) = sep;
498         }

500         if (cur != stacks_array + si.si_entries) {
501                 mdb_warn("stacks: miscounted array size (%d != size: %d)\n",
502                     (cur - stacks_array), stacks_array_size);
503                 return (DCMD_ERR);
504         }
505         qsort(stacks_array, si.si_entries, sizeof (*stacks_array),
506             stacks_entry_comp);

508         /* Now that we're done, free the hash table */
509         stacks_hash = NULL;
510         mdb_free(si.si_hash, STACKS_HSIZE * sizeof (*si.si_hash));

512         if (tlist == NULL)
513                 stacks_state = STACKS_STATE_DONE;

515         if (verbose)
516                 mdb_warn("stacks: done\n");

518         return (DCMD_OK);
519 }

521 static int
522 stacks_has_caller(stacks_entry_t *sep, uintptr_t addr)
523 {
```

```
524         uintptr_t laddr = addr;
525         uintptr_t haddr = addr + 1;
526         int idx;
527         char c[MDB_SYM_NAMLEN];
528         GElf_Sym sym;

530         if (mdb_lookup_by_addr(addr, MDB_SYM_FUZZY,
531             c, sizeof (c), &sym) != -1 &&
532             addr == (uintptr_t)sym.st_value) {
533                 laddr = (uintptr_t)sym.st_value;
534                 haddr = (uintptr_t)sym.st_value + sym.st_size;
535         }

537         for (idx = 0; idx < sep->se_depth; idx++)
538                 if (sep->se_stack[idx] >= laddr && sep->se_stack[idx] < haddr)
539                         return (1);

541         return (0);
542 }

544 static int
545 stacks_has_module(stacks_entry_t *sep, stacks_module_t *mp)
546 {
547         int idx;

549         for (idx = 0; idx < sep->se_depth; idx++) {
550                 if (sep->se_stack[idx] >= mp->sm_text &&
551                     sep->se_stack[idx] < mp->sm_text + mp->sm_size)
552                         return (1);
553         }

555         return (0);
556 }

558 static int
559 stacks_module_find(const char *name, stacks_module_t *mp)
560 {
561         (void) strncpy(mp->sm_name, name, sizeof (mp->sm_name));

563         if (stacks_module(mp) != 0)
564                 return (-1);

566         if (mp->sm_size == 0) {
567                 mdb_warn("stacks: module \"%s\" is unknown\n", name);
568                 return (-1);
569         }

571         return (0);
572 }

574 static int
575 uintptrcomp(const void *lp, const void *rp)
576 {
577         uintptr_t lhs = *(const uintptr_t *)lp;
578         uintptr_t rhs = *(const uintptr_t *)rp;
579         if (lhs > rhs)
580                 return (1);
581         if (lhs < rhs)
582                 return (-1);
583         return (0);
584 }

586 /*ARGSUSED*/
587 int
588 stacks(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
589 {
```



```
590         size_t idx;

592         char *seen = NULL;

594         const char *caller_str = NULL;
595         const char *excl_caller_str = NULL;
596         uintptr_t caller = 0, excl_caller = 0;
597         const char *module_str = NULL;
598         const char *excl_module_str = NULL;
599         stacks_module_t module, excl_module;
600         const char *sobj = NULL;
601         const char *excl_sobj = NULL;
602         uintptr_t sobj_ops = 0, excl_sobj_ops = 0;
603         const char *tstate_str = NULL;
604         const char *excl_tstate_str = NULL;
605         uint_t tstate = -1U;
606         uint_t excl_tstate = -1U;
607         uint_t printed = 0;

609         uint_t all = 0;
610         uint_t force = 0;
611         uint_t interesting = 0;
612         uint_t verbose = 0;

614         /*
615          * We have a slight behavior difference between having piped
616          * input and 'addr::stacks'.  Without a pipe, we assume the
617          * thread pointer given is a representative thread, and so
618          * we include all similar threads in the system in our output.
619          *
620          * With a pipe, we filter down to just the threads in our
621          * input.
622          */
623         uint_t addrspec = (flags & DCMD_ADDRSPEC);
624         uint_t only_matching = addrspec && (flags & DCMD_PIPE);

626         mdb_pipe_t p;

628         bzero(&module, sizeof (module));
629         bzero(&excl_module, sizeof (excl_module));

631         if (mdb_getopts(argc, argv,
632             'a', MDB_OPT_SETBITS, TRUE, &all,
633             'f', MDB_OPT_SETBITS, TRUE, &force,
634             'i', MDB_OPT_SETBITS, TRUE, &interesting,
635             'v', MDB_OPT_SETBITS, TRUE, &verbose,
636             'c', MDB_OPT_STR, &caller_str,
637             'C', MDB_OPT_STR, &excl_caller_str,
638             'm', MDB_OPT_STR, &module_str,
639             'M', MDB_OPT_STR, &excl_module_str,
640             's', MDB_OPT_STR, &sobj,
641             'S', MDB_OPT_STR, &excl_sobj,
642             't', MDB_OPT_STR, &tstate_str,
643             'T', MDB_OPT_STR, &excl_tstate_str,
644             NULL) != argc)
645                 return (DCMD_USAGE);

647         if (interesting) {
648                 if (sobj != NULL || excl_sobj != NULL ||
649                     tstate_str != NULL || excl_tstate_str != NULL) {
650                         mdb_warn(
651                             "stacks: -i is incompatible with -[sStT]\n");
652                         return (DCMD_USAGE);
653                 }
654                 excl_sobj = "CV";
655                 excl_tstate_str = "FREE";
```

```
656         }
658         if (caller_str != NULL) {
659                 mdb_set_dot(0);
660                 if (mdb_eval(caller_str) != 0) {
661                         mdb_warn("stacks: evaluation of \"%s\" failed",
662                             caller_str);
663                         return (DCMD_ABORT);
664                 }
665                 caller = mdb_get_dot();
666         }
668         if (excl_caller_str != NULL) {
669                 mdb_set_dot(0);
670                 if (mdb_eval(excl_caller_str) != 0) {
671                         mdb_warn("stacks: evaluation of \"%s\" failed",
672                             excl_caller_str);
673                         return (DCMD_ABORT);
674                 }
675                 excl_caller = mdb_get_dot();
676         }
677         mdb_set_dot(addr);
679         if (module_str != NULL && stacks_module_find(module_str, &module) != 0)
680                 return (DCMD_ABORT);
682         if (excl_module_str != NULL &&
683             stacks_module_find(excl_module_str, &excl_module) != 0)
684                 return (DCMD_ABORT);
686         if (sobj != NULL && text_to_sobj(sobj, &sobj_ops) != 0)
687                 return (DCMD_USAGE);
689         if (excl_sobj != NULL && text_to_sobj(excl_sobj, &excl_sobj_ops) != 0)
690                 return (DCMD_USAGE);
692         if (sobj_ops != 0 && excl_sobj_ops != 0) {
693                 mdb_warn("stacks: only one of -s and -S can be specified\n");
694                 return (DCMD_USAGE);
695         }
697         if (tstate_str != NULL && text_to_tstate(tstate_str, &tstate) != 0)
698                 return (DCMD_USAGE);
700         if (excl_tstate_str != NULL &&
701             text_to_tstate(excl_tstate_str, &excl_tstate) != 0)
702                 return (DCMD_USAGE);
704         if (tstate != -1U && excl_tstate != -1U) {
705                 mdb_warn("stacks: only one of -t and -T can be specified\n");
706                 return (DCMD_USAGE);
707         }
709         /*
710          * If there's an address specified, we're going to further filter
711          * to only entries which have an address in the input.  To reduce
712          * overhead (and make the sorted output come out right), we
713          * use mdb_get_pipe() to grab the entire pipeline of input, then
714          * use qsort() and bsearch() to speed up the search.
715          */
716         if (addrspec) {
717                 mdb_get_pipe(&p);
718                 if (p.pipe_data == NULL || p.pipe_len == 0) {
719                         p.pipe_data = &addr;
720                         p.pipe_len = 1;
721                 }
```



```
722                 qsort(p.pipe_data, p.pipe_len, sizeof (uintptr_t),
723                     uintptrcomp);
725                 /* remove any duplicates in the data */
726                 idx = 0;
727                 while (idx < p.pipe_len - 1) {
728                         uintptr_t *data = &p.pipe_data[idx];
729                         size_t len = p.pipe_len - idx;
731                         if (data[0] == data[1]) {
732                                 memmove(data, data + 1,
733                                     (len - 1) * sizeof (*data));
734                                 p.pipe_len--;
735                                 continue; /* repeat without incrementing idx */
736                         }
737                         idx++;
738                 }
740                 seen = mdb_zalloc(p.pipe_len, UM_SLEEP | UM_GC);
741         }
743         /*
744          * Force a cleanup if we're connected to a live system. Never
745          * do a cleanup after the first invocation around the loop.
746          */
747         force |= (mdb_get_state() == MDB_STATE_RUNNING);
748         if (force && (flags & (DCMD_LOOPFIRST|DCMD_LOOP)) == DCMD_LOOP)
749                 force = 0;
751         stacks_cleanup(force);
753         if (stacks_state == STACKS_STATE_CLEAN) {
754                 int res = stacks_run(verbose, addrspec ? &p : NULL);
755                 if (res != DCMD_OK)
756                         return (res);
757         }
759         for (idx = 0; idx < stacks_array_size; idx++) {
760                 stacks_entry_t *sep = stacks_array[idx];
761                 stacks_entry_t *cur = sep;
762                 int frame;
763                 size_t count = sep->se_count;
765                 if (addrspec) {
766                         stacks_entry_t *head = NULL, *tail = NULL, *sp;
767                         size_t foundcount = 0;
768                         /*
769                          * We use the now-unused hash chain field se_next to
770                          * link together the dups which match our list.
771                          */
772                         for (sp = sep; sp != NULL; sp = sp->se_dup) {
773                                 uintptr_t *entry = bsearch(&sp->se_thread,
774                                     p.pipe_data, p.pipe_len, sizeof (uintptr_t),
775                                     uintptrcomp);
776                                 if (entry != NULL) {
777                                         foundcount++;
778                                         seen[entry - p.pipe_data]++;
779                                         if (head == NULL)
780                                                 head = sp;
781                                         else
782                                                 tail->se_next = sp;
783                                         tail = sp;
784                                         sp->se_next = NULL;
785                                 }
786                         }
787                         if (head == NULL)
```

```
788                                 continue;       /* no match, skip entry */

790                         if (only_matching) {
791                                 cur = sep = head;
792                                 count = foundcount;
793                         }
794                 }

796                 if (caller != 0 && !stacks_has_caller(sep, caller))
797                         continue;

799                 if (excl_caller != 0 && stacks_has_caller(sep, excl_caller))
800                         continue;

802                 if (module.sm_size != 0 && !stacks_has_module(sep, &module))
803                         continue;

805                 if (excl_module.sm_size != 0 &&
806                     stacks_has_module(sep, &excl_module))
807                         continue;

809                 if (tstate != -1U) {
810                         if (tstate == TSTATE_PANIC) {
811                                 if (!sep->se_panic)
812                                         continue;
813                         } else if (sep->se_panic || sep->se_tstate != tstate)
814                                 continue;
815                 }
816                 if (excl_tstate != -1U) {
817                         if (excl_tstate == TSTATE_PANIC) {
818                                 if (sep->se_panic)
819                                         continue;
820                         } else if (!sep->se_panic &&
821                             sep->se_tstate == excl_tstate)
822                                 continue;
823                 }

825                 if (sobj_ops == SOBJ_ALL) {
826                         if (sep->se_sobj_ops == 0)
827                                 continue;
828                 } else if (sobj_ops != 0) {
829                         if (sobj_ops != sep->se_sobj_ops)
830                                 continue;
831                 }

833                 if (!(interesting && sep->se_panic)) {
834                         if (excl_sobj_ops == SOBJ_ALL) {
835                                 if (sep->se_sobj_ops != 0)
836                                         continue;
837                         } else if (excl_sobj_ops != 0) {
838                                 if (excl_sobj_ops == sep->se_sobj_ops)
839                                         continue;
840                         }
841                 }

843                 if (flags & DCMD_PIPE_OUT) {
844                         while (sep != NULL) {
845                                 mdb_printf("%lr\n", sep->se_thread);
846                                 sep = only_matching ?
847                                     sep->se_next : sep->se_dup;
848                         }
849                         continue;
850                 }

852                 if (all || !printed) {
853                         mdb_printf("%<u>%-?s %-8s %-?s %8s%</u>\n",
```



```
854                                 "THREAD", "STATE", "SOBJ", "COUNT");
855                         printed = 1;
856                 }

858                 do {
859                         char state[20];
860                         char sobj[100];

862                         tstate_to_text(cur->se_tstate, cur->se_panic,
863                             state, sizeof (state));
864                         sobj_to_text(cur->se_sobj_ops,
865                             sobj, sizeof (sobj));

867                         if (cur == sep)
868                                 mdb_printf("%-?p %-8s %-?s %8d\n",
869                                     cur->se_thread, state, sobj, count);
870                         else
871                                 mdb_printf("%-?p %-8s %-?s %8s\n",
872                                     cur->se_thread, state, sobj, "-");

874                         cur = only_matching ? cur->se_next : cur->se_dup;
875                 } while (all && cur != NULL);

877                 if (sep->se_failed != 0) {
878                         char *reason;
879                         switch (sep->se_failed) {
880                         case FSI_FAIL_NOTINMEMORY:
881                                 reason = "thread not in memory";
882                                 break;
883                         case FSI_FAIL_THREADCORRUPT:
884                                 reason = "thread structure stack info corrupt";
885                                 break;
886                         case FSI_FAIL_STACKNOTFOUND:
887                                 reason = "no consistent stack found";
888                                 break;
889                         default:
890                                 reason = "unknown failure";
891                                 break;
892                         }
893                         mdb_printf("%?s <%s>\n", "", reason);
894                 }

896                 for (frame = 0; frame < sep->se_depth; frame++)
897                         mdb_printf("%?s %a\n", "", sep->se_stack[frame]);
898                 if (sep->se_overflow)
899                         mdb_printf("%?s ... truncated ...\n", "");
900                 mdb_printf("\n");
901         }

903         if (flags & DCMD_ADDRSPEC) {
904                 for (idx = 0; idx < p.pipe_len; idx++)
905                         if (seen[idx] == 0)
906                                 mdb_warn("stacks: %p not in thread list\n",
907                                     p.pipe_data[idx]);
908         }
909         return (DCMD_OK);
910 }
```