```
**********************************************************
   66811 Tue Jan 14 16:47:26 2014
new/usr/src/cmd/mdb/common/modules/dtrace/dtrace.c
4469 DTrace helper tracing should be dynamic
**********************************************************
```

```
/*
 * 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) 2003, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
#endif /* ! codereview */
 */

/*
 * explicitly define DTRACE_ERRDEBUG to pull in definition of dtrace_errhash_t
 * explicitly define _STDARG_H to avoid stdarg.h/varargs.h u/k defn conflict
 */
#define DTRACE_ERRDEBUG
#define _STDARG_H

#include <mdb/mdb_param.h>
#include <mdb/mdb_modapi.h>
#include <mdb/mdb_ks.h>
#include <sys/dtrace_impl.h>
#include <sys/vmem_impl.h>
#include <sys/ddi_impldefs.h>
#include <sys/sysmacros.h>
#include <sys/kobj.h>
#include <dtrace.h>
#include <alloca.h>
#include <ctype.h>
#include <errno.h>
#include <math.h>
#include <stdio.h>
#include <unistd.h>

/*ARGSUSED*/
int
id2probe(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
{
        uintptr_t probe = NULL;
        uintptr_t probes;

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



```
        if (addr == DTRACE_IDNONE || addr > UINT32_MAX)
                goto out;

        if (mdb_readvar(&probes, "dtrace_probes") == -1) {
                mdb_warn("failed to read 'dtrace_probes'");
                return (DCMD_ERR);
        }

        probes += (addr - 1) * sizeof (dtrace_probe_t *);

        if (mdb_vread(&probe, sizeof (uintptr_t), probes) == -1) {
                mdb_warn("failed to read dtrace_probes[%d]", addr - 1);
                return (DCMD_ERR);
        }

out:
        mdb_printf("%p\n", probe);
        return (DCMD_OK);
}

void
dtrace_help(void)
{

        mdb_printf("Given a dtrace_state_t structure that represents a "
            "DTrace consumer, prints\n"
            "dtrace(1M)-like output for in-kernel DTrace data.  (The "
            "dtrace_state_t\n"
            "structures for all DTrace consumers may be obtained by running "
            "the \n"
            "::dtrace_state dcmd.)   When data is present on multiple CPUs, "
            "data are\n"
            "presented in CPU order, with records within each CPU ordered "
            "oldest to \n"
            "youngest.  Options:\n\n"
            "-c cpu     Only provide output for specified CPU.\n");
}

static int
dtracemdb_eprobe(dtrace_state_t *state, dtrace_eprobedesc_t *epd)
{
        dtrace_epid_t epid = epd->dtepd_epid;
        dtrace_probe_t probe;
        dtrace_ecb_t ecb;
        uintptr_t addr, paddr, ap;
        dtrace_action_t act;
        int nactions, nrecs;

        addr = (uintptr_t)state->dts_ecbs +
            (epid - 1) * sizeof (dtrace_ecb_t *);

        if (mdb_vread(&addr, sizeof (addr), addr) == -1) {
                mdb_warn("failed to read ecb for epid %d", epid);
                return (-1);
        }

        if (addr == NULL) {
                mdb_warn("epid %d doesn't match an ecb\n", epid);
                return (-1);
        }

        if (mdb_vread(&ecb, sizeof (ecb), addr) == -1) {
                mdb_warn("failed to read ecb at %p", addr);
                return (-1);
        }
```

```
128         paddr = (uintptr_t)ecb.dte_probe;
129
130         if (mdb_vread(&probe, sizeof (probe), paddr) == -1) {
131                 mdb_warn("failed to read probe for ecb %p", addr);
132                 return (-1);
133         }
134
135         /*
136          * This is a little painful:  in order to find the number of actions,
137          * we need to first walk through them.
138          */
139         for (ap = (uintptr_t)ecb.dte_action, nactions = 0; ap != NULL; ) {
140                 if (mdb_vread(&act, sizeof (act), ap) == -1) {
141                         mdb_warn("failed to read action %p on ecb %p",
142                             ap, addr);
143                         return (-1);
144                 }
145
146                 if (!DTRACEACT_ISAGG(act.dta_kind) && !act.dta_intuple)
147                         nactions++;
148
149                 ap = (uintptr_t)act.dta_next;
150         }
151
152         nrecs = epd->dtepd_nrecs;
153         epd->dtepd_nrecs = nactions;
154         epd->dtepd_probeid = probe.dtpr_id;
155         epd->dtepd_uarg = ecb.dte_uarg;
156         epd->dtepd_size = ecb.dte_size;
157
158         for (ap = (uintptr_t)ecb.dte_action, nactions = 0; ap != NULL; ) {
159                 if (mdb_vread(&act, sizeof (act), ap) == -1) {
160                         mdb_warn("failed to read action %p on ecb %p",
161                             ap, addr);
162                         return (-1);
163                 }
164
165                 if (!DTRACEACT_ISAGG(act.dta_kind) && !act.dta_intuple) {
166                         if (nrecs-- == 0)
167                                 break;
168
169                         epd->dtepd_rec[nactions++] = act.dta_rec;
170                 }
171
172                 ap = (uintptr_t)act.dta_next;
173         }
174
175         return (0);
176 }
177
178 /*ARGSUSED*/
179 static int
180 dtracemdb_probe(dtrace_state_t *state, dtrace_probedesc_t *pd)
181 {
182         uintptr_t base, addr, paddr, praddr;
183         int nprobes, i;
184         dtrace_probe_t probe;
185         dtrace_provider_t prov;
186
187         if (pd->dtpd_id == DTRACE_IDNONE)
188                 pd->dtpd_id++;
189
190         if (mdb_readvar(&base, "dtrace_probes") == -1) {
191                 mdb_warn("failed to read 'dtrace_probes'");
192                 return (-1);
193         }
```



```
194
195         if (mdb_readvar(&nprobes, "dtrace_nprobes") == -1) {
196                 mdb_warn("failed to read 'dtrace_nprobes'");
197                 return (-1);
198         }
199
200         for (i = pd->dtpd_id; i <= nprobes; i++) {
201                 addr = base + (i - 1) * sizeof (dtrace_probe_t *);
202
203                 if (mdb_vread(&paddr, sizeof (paddr), addr) == -1) {
204                         mdb_warn("couldn't read probe pointer at %p", addr);
205                         return (-1);
206                 }
207
208                 if (paddr != NULL)
209                         break;
210         }
211
212         if (paddr == NULL) {
213                 errno = ESRCH;
214                 return (-1);
215         }
216
217         if (mdb_vread(&probe, sizeof (probe), paddr) == -1) {
218                 mdb_warn("couldn't read probe at %p", paddr);
219                 return (-1);
220         }
221
222         pd->dtpd_id = probe.dtpr_id;
223
224         if (mdb_vread(pd->dtpd_name, DTRACE_NAMELEN,
225             (uintptr_t)probe.dtpr_name) == -1) {
226                 mdb_warn("failed to read probe name for probe %p", paddr);
227                 return (-1);
228         }
229
230         if (mdb_vread(pd->dtpd_func, DTRACE_FUNCNAMELEN,
231             (uintptr_t)probe.dtpr_func) == -1) {
232                 mdb_warn("failed to read function name for probe %p", paddr);
233                 return (-1);
234         }
235
236         if (mdb_vread(pd->dtpd_mod, DTRACE_MODNAMELEN,
237             (uintptr_t)probe.dtpr_mod) == -1) {
238                 mdb_warn("failed to read module name for probe %p", paddr);
239                 return (-1);
240         }
241
242         praddr = (uintptr_t)probe.dtpr_provider;
243
244         if (mdb_vread(&prov, sizeof (prov), praddr) == -1) {
245                 mdb_warn("failed to read provider for probe %p", paddr);
246                 return (-1);
247         }
248
249         if (mdb_vread(pd->dtpd_provider, DTRACE_PROVNAMELEN,
250             (uintptr_t)prov.dtpv_name) == -1) {
251                 mdb_warn("failed to read provider name for probe %p", paddr);
252                 return (-1);
253         }
254
255         return (0);
256 }
257
258 /*ARGSUSED*/
259 static int
```

```
dtracemdb_aggdesc(dtrace_state_t *state, dtrace_aggdesc_t *agd)
{
        dtrace_aggid_t aggid = agd->dtagd_id;
        dtrace_aggregation_t agg;
        dtrace_ecb_t ecb;
        uintptr_t addr, eaddr, ap, last;
        dtrace_action_t act;
        dtrace_recdesc_t *lrec;
        int nactions, nrecs;

        addr = (uintptr_t)state->dts_aggregations +
            (aggid - 1) * sizeof (dtrace_aggregation_t *);

        if (mdb_vread(&addr, sizeof (addr), addr) == -1) {
                mdb_warn("failed to read aggregation for aggid %d", aggid);
                return (-1);
        }

        if (addr == NULL) {
                mdb_warn("aggid %d doesn't match an aggregation\n", aggid);
                return (-1);
        }

        if (mdb_vread(&agg, sizeof (agg), addr) == -1) {
                mdb_warn("failed to read aggregation at %p", addr);
                return (-1);
        }

        eaddr = (uintptr_t)agg.dtag_ecb;

        if (mdb_vread(&ecb, sizeof (ecb), eaddr) == -1) {
                mdb_warn("failed to read ecb for aggregation %p", addr);
                return (-1);
        }

        last = (uintptr_t)addr + offsetof(dtrace_aggregation_t, dtag_action);

        /*
         * This is a little painful:  in order to find the number of actions,
         * we need to first walk through them.
         */
        ap = (uintptr_t)agg.dtag_first;
        nactions = 0;

        for (;;) {
                if (mdb_vread(&act, sizeof (act), ap) == -1) {
                        mdb_warn("failed to read action %p on aggregation %p",
                            ap, addr);
                        return (-1);
                }

                nactions++;

                if (ap == last)
                        break;

                ap = (uintptr_t)act.dta_next;
        }

        lrec = &act.dta_rec;
        agd->dtagd_size = lrec->dtrd_offset + lrec->dtrd_size - agg.dtag_base;

        nrecs = agd->dtagd_nrecs;
        agd->dtagd_nrecs = nactions;
        agd->dtagd_epid = ecb.dte_epid;
```



```
        ap = (uintptr_t)agg.dtag_first;
        nactions = 0;

        for (;;) {
                dtrace_recdesc_t rec;

                if (mdb_vread(&act, sizeof (act), ap) == -1) {
                        mdb_warn("failed to read action %p on aggregation %p",
                            ap, addr);
                        return (-1);
                }

                if (nrecs-- == 0)
                        break;

                rec = act.dta_rec;
                rec.dtrd_offset -= agg.dtag_base;
                rec.dtrd_uarg = 0;
                agd->dtagd_rec[nactions++] = rec;

                if (ap == last)
                        break;

                ap = (uintptr_t)act.dta_next;
        }

        return (0);
}

static int
dtracemdb_bufsnap(dtrace_buffer_t *which, dtrace_bufdesc_t *desc)
{
        uintptr_t addr;
        size_t bufsize;
        dtrace_buffer_t buf;
        caddr_t data = desc->dtbd_data;
        processorid_t max_cpuid, cpu = desc->dtbd_cpu;

        if (mdb_readvar(&max_cpuid, "max_cpuid") == -1) {
                mdb_warn("failed to read 'max_cpuid'");
                errno = EIO;
                return (-1);
        }

        if (cpu < 0 || cpu > max_cpuid) {
                errno = EINVAL;
                return (-1);
        }

        addr = (uintptr_t)which + cpu * sizeof (dtrace_buffer_t);

        if (mdb_vread(&buf, sizeof (buf), addr) == -1) {
                mdb_warn("failed to read buffer description at %p", addr);
                errno = EIO;
                return (-1);
        }

        if (buf.dtb_tomax == NULL) {
                errno = ENOENT;
                return (-1);
        }

        if (buf.dtb_flags & DTRACEBUF_WRAPPED) {
                bufsize = buf.dtb_size;
        } else {
                bufsize = buf.dtb_offset;
```

```
392         }
394         if (mdb_vread(data, bufsize, (uintptr_t)buf.dtb_tomax) == -1) {
395                 mdb_warn("couldn't read buffer for CPU %d", cpu);
396                 errno = EIO;
397                 return (-1);
398         }
400         if (buf.dtb_offset > buf.dtb_size) {
401                 mdb_warn("buffer for CPU %d has corrupt offset\n", cpu);
402                 errno = EIO;
403                 return (-1);
404         }
406         if (buf.dtb_flags & DTRACEBUF_WRAPPED) {
407                 if (buf.dtb_xamot_offset > buf.dtb_size) {
408                         mdb_warn("ringbuffer for CPU %d has corrupt "
409                             "wrapped offset\n", cpu);
410                         errno = EIO;
411                         return (-1);
412                 }
414                 /*
415                  * If the ring buffer has wrapped, it needs to be polished.
416                  * See the comment in dtrace_buffer_polish() for details.
417                  */
418                 if (buf.dtb_offset < buf.dtb_xamot_offset) {
419                         bzero(data + buf.dtb_offset,
420                             buf.dtb_xamot_offset - buf.dtb_offset);
421                 }
423                 if (buf.dtb_offset > buf.dtb_xamot_offset) {
424                         bzero(data + buf.dtb_offset,
425                             buf.dtb_size - buf.dtb_offset);
426                         bzero(data, buf.dtb_xamot_offset);
427                 }
429                 desc->dtbd_oldest = buf.dtb_xamot_offset;
430         } else {
431                 desc->dtbd_oldest = 0;
432         }
434         desc->dtbd_size = bufsize;
435         desc->dtbd_drops = buf.dtb_drops;
436         desc->dtbd_errors = buf.dtb_errors;
438         return (0);
439 }
441 /*
442  * This is essentially identical to its cousin in the kernel -- with the
443  * notable exception that we automatically set DTRACEOPT_GRABANON if this
444  * state is an anonymous enabling.
445  */
446 static dof_hdr_t *
447 dtracemdb_dof_create(dtrace_state_t *state, int isanon)
448 {
449         dof_hdr_t *dof;
450         dof_sec_t *sec;
451         dof_optdesc_t *opt;
452         int i, len = sizeof (dof_hdr_t) +
453             roundup(sizeof (dof_sec_t), sizeof (uint64_t)) +
454             sizeof (dof_optdesc_t) * DTRACEOPT_MAX;
456         dof = mdb_zalloc(len, UM_SLEEP);
457         dof->dofh_ident[DOF_ID_MAG0] = DOF_MAG_MAG0;
```



```
458         dof->dofh_ident[DOF_ID_MAG1] = DOF_MAG_MAG1;
459         dof->dofh_ident[DOF_ID_MAG2] = DOF_MAG_MAG2;
460         dof->dofh_ident[DOF_ID_MAG3] = DOF_MAG_MAG3;
462         dof->dofh_ident[DOF_ID_MODEL] = DOF_MODEL_NATIVE;
463         dof->dofh_ident[DOF_ID_ENCODING] = DOF_ENCODE_NATIVE;
464         dof->dofh_ident[DOF_ID_VERSION] = DOF_VERSION;
465         dof->dofh_ident[DOF_ID_DIFVERS] = DIF_VERSION;
466         dof->dofh_ident[DOF_ID_DIFIREG] = DIF_DIR_NREGS;
467         dof->dofh_ident[DOF_ID_DIFTREG] = DIF_DTR_NREGS;
469         dof->dofh_flags = 0;
470         dof->dofh_hdrsize = sizeof (dof_hdr_t);
471         dof->dofh_secsize = sizeof (dof_sec_t);
472         dof->dofh_secnum = 1;   /* only DOF_SECT_OPTDESC */
473         dof->dofh_secoff = sizeof (dof_hdr_t);
474         dof->dofh_loadsz = len;
475         dof->dofh_filesz = len;
476         dof->dofh_pad = 0;
478         /*
479          * Fill in the option section header...
480          */
481         sec = (dof_sec_t *)((uintptr_t)dof + sizeof (dof_hdr_t));
482         sec->dofs_type = DOF_SECT_OPTDESC;
483         sec->dofs_align = sizeof (uint64_t);
484         sec->dofs_flags = DOF_SECF_LOAD;
485         sec->dofs_entsize = sizeof (dof_optdesc_t);
487         opt = (dof_optdesc_t *)((uintptr_t)sec +
488             roundup(sizeof (dof_sec_t), sizeof (uint64_t)));
490         sec->dofs_offset = (uintptr_t)opt - (uintptr_t)dof;
491         sec->dofs_size = sizeof (dof_optdesc_t) * DTRACEOPT_MAX;
493         for (i = 0; i < DTRACEOPT_MAX; i++) {
494                 opt[i].dofo_option = i;
495                 opt[i].dofo_strtab = DOF_SECIDX_NONE;
496                 opt[i].dofo_value = state->dts_options[i];
497         }
499         if (isanon)
500                 opt[DTRACEOPT_GRABANON].dofo_value = 1;
502         return (dof);
503 }
505 static int
506 dtracemdb_format(dtrace_state_t *state, dtrace_fmtdesc_t *desc)
507 {
508         uintptr_t addr, faddr;
509         char c;
510         int len = 0;
512         if (desc->dtfd_format == 0 || desc->dtfd_format > state->dts_nformats) {
513                 errno = EINVAL;
514                 return (-1);
515         }
517         faddr = (uintptr_t)state->dts_formats +
518             (desc->dtfd_format - 1) * sizeof (char *);
520         if (mdb_vread(&addr, sizeof (addr), faddr) == -1) {
521                 mdb_warn("failed to read format string pointer at %p", faddr);
522                 return (-1);
523         }
```

```
525         do {
526                 if (mdb_vread(&c, sizeof (c), addr + len++) == -1) {
527                         mdb_warn("failed to read format string at %p", addr);
528                         return (-1);
529                 }
530         } while (c != '\0');
531
532         if (len > desc->dtfd_length) {
533                 desc->dtfd_length = len;
534                 return (0);
535         }
536
537         if (mdb_vread(desc->dtfd_string, len, addr) == -1) {
538                 mdb_warn("failed to reread format string at %p", addr);
539                 return (-1);
540         }
541
542         return (0);
543 }
544
545 static int
546 dtracemdb_status(dtrace_state_t *state, dtrace_status_t *status)
547 {
548         dtrace_dstate_t *dstate;
549         int i, j;
550         uint64_t nerrs;
551         uintptr_t addr;
552         int ncpu;
553
554         if (mdb_readvar(&ncpu, "_ncpu") == -1) {
555                 mdb_warn("failed to read '_ncpu'");
556                 return (DCMD_ERR);
557         }
558
559         bzero(status, sizeof (dtrace_status_t));
560
561         if (state->dts_activity == DTRACE_ACTIVITY_INACTIVE) {
562                 errno = ENOENT;
563                 return (-1);
564         }
565
566         /*
567          * For the MDB backend, we never set dtst_exiting or dtst_filled.  This
568          * is by design:  we don't want the library to try to stop tracing,
569          * because it doesn't particularly mean anything.
570          */
571         nerrs = state->dts_errors;
572         dstate = &state->dts_vstate.dtvs_dynvars;
573
574         for (i = 0; i < ncpu; i++) {
575                 dtrace_dstate_percpu_t dcpu;
576                 dtrace_buffer_t buf;
577
578                 addr = (uintptr_t)&dstate->dtds_percpu[i];
579
580                 if (mdb_vread(&dcpu, sizeof (dcpu), addr) == -1) {
581                         mdb_warn("failed to read per-CPU dstate at %p", addr);
582                         return (-1);
583                 }
584
585                 status->dtst_dyndrops += dcpu.dtdsc_drops;
586                 status->dtst_dyndrops_dirty += dcpu.dtdsc_dirty_drops;
587                 status->dtst_dyndrops_rinsing += dcpu.dtdsc_rinsing_drops;
588
589                 addr = (uintptr_t)&state->dts_buffer[i];
```



```
590
591                 if (mdb_vread(&buf, sizeof (buf), addr) == -1) {
592                         mdb_warn("failed to read per-CPU buffer at %p", addr);
593                         return (-1);
594                 }
595
596                 nerrs += buf.dtb_errors;
597
598                 for (j = 0; j < state->dts_nspeculations; j++) {
599                         dtrace_speculation_t spec;
600
601                         addr = (uintptr_t)&state->dts_speculations[j];
602
603                         if (mdb_vread(&spec, sizeof (spec), addr) == -1) {
604                                 mdb_warn("failed to read "
605                                     "speculation at %p", addr);
606                                 return (-1);
607                         }
608
609                         addr = (uintptr_t)&spec.dtsp_buffer[i];
610
611                         if (mdb_vread(&buf, sizeof (buf), addr) == -1) {
612                                 mdb_warn("failed to read "
613                                     "speculative buffer at %p", addr);
614                                 return (-1);
615                         }
616
617                         status->dtst_specdrops += buf.dtb_xamot_drops;
618                 }
619         }
620
621         status->dtst_specdrops_busy = state->dts_speculations_busy;
622         status->dtst_specdrops_unavail = state->dts_speculations_unavail;
623         status->dtst_errors = nerrs;
624
625         return (0);
626 }
627
628 typedef struct dtracemdb_data {
629         dtrace_state_t *dtmd_state;
630         char *dtmd_symstr;
631         char *dtmd_modstr;
632         uintptr_t dtmd_addr;
633         int dtmd_isanon;
634 } dtracemdb_data_t;
635
636 static int
637 dtracemdb_ioctl(void *varg, int cmd, void *arg)
638 {
639         dtracemdb_data_t *data = varg;
640         dtrace_state_t *state = data->dtmd_state;
641
642         switch (cmd) {
643         case DTRACEIOC_CONF: {
644                 dtrace_conf_t *conf = arg;
645
646                 bzero(conf, sizeof (conf));
647                 conf->dtc_difversion = DIF_VERSION;
648                 conf->dtc_difintregs = DIF_DIR_NREGS;
649                 conf->dtc_diftupregs = DIF_DTR_NREGS;
650                 conf->dtc_ctfmodel = CTF_MODEL_NATIVE;
651
652                 return (0);
653         }
654
655         case DTRACEIOC_DOFGET: {
```

```
656                 dof_hdr_t *hdr = arg, *dof;

658                 dof = dtracemdb_dof_create(state, data->dtmd_isanon);
659                 bcopy(dof, hdr, MIN(hdr->dofh_loadsz, dof->dofh_loadsz));
660                 mdb_free(dof, dof->dofh_loadsz);

662                 return (0);
663         }

665         case DTRACEIOC_BUFSNAP:
666                 return (dtracemdb_bufsnap(state->dts_buffer, arg));

668         case DTRACEIOC_AGGSNAP:
669                 return (dtracemdb_bufsnap(state->dts_aggbuffer, arg));

671         case DTRACEIOC_AGGDESC:
672                 return (dtracemdb_aggdesc(state, arg));

674         case DTRACEIOC_EPROBE:
675                 return (dtracemdb_eprobe(state, arg));

677         case DTRACEIOC_PROBES:
678                 return (dtracemdb_probe(state, arg));

680         case DTRACEIOC_FORMAT:
681                 return (dtracemdb_format(state, arg));

683         case DTRACEIOC_STATUS:
684                 return (dtracemdb_status(state, arg));

686         case DTRACEIOC_GO:
687                 *(processorid_t *)arg = -1;
688                 return (0);

690         case DTRACEIOC_ENABLE:
691                 errno = ENOTTY; /* see dt_open.c:dtrace_go() */
692                 return (-1);

694         case DTRACEIOC_PROVIDER:
695         case DTRACEIOC_PROBEMATCH:
696                 errno = ESRCH;
697                 return (-1);

699         default:
700                 mdb_warn("unexpected ioctl 0x%x (%s)\n", cmd,
701                     cmd == DTRACEIOC_PROVIDER   ? "DTRACEIOC_PROVIDER" :
702                     cmd == DTRACEIOC_PROBES     ? "DTRACEIOC_PROBES" :
703                     cmd == DTRACEIOC_BUFSNAP    ? "DTRACEIOC_BUFSNAP" :
704                     cmd == DTRACEIOC_PROBEMATCH ? "DTRACEIOC_PROBEMATCH" :
705                     cmd == DTRACEIOC_ENABLE     ? "DTRACEIOC_ENABLE" :
706                     cmd == DTRACEIOC_AGGSNAP    ? "DTRACEIOC_AGGSNAP" :
707                     cmd == DTRACEIOC_EPROBE     ? "DTRACEIOC_EPROBE" :
708                     cmd == DTRACEIOC_PROBEARG   ? "DTRACEIOC_PROBEARG" :
709                     cmd == DTRACEIOC_CONF       ? "DTRACEIOC_CONF" :
710                     cmd == DTRACEIOC_STATUS     ? "DTRACEIOC_STATUS" :
711                     cmd == DTRACEIOC_GO         ? "DTRACEIOC_GO" :
712                     cmd == DTRACEIOC_STOP       ? "DTRACEIOC_STOP" :
713                     cmd == DTRACEIOC_AGGDESC    ? "DTRACEIOC_AGGDESC" :
714                     cmd == DTRACEIOC_FORMAT     ? "DTRACEIOC_FORMAT" :
715                     cmd == DTRACEIOC_DOFGET     ? "DTRACEIOC_DOFGET" :
716                     cmd == DTRACEIOC_REPLICATE  ? "DTRACEIOC_REPLICATE" :
717                     "???");
718                 errno = ENXIO;
719                 return (-1);
720         }
721 }
```



```
723 static int
724 dtracemdb_modctl(uintptr_t addr, const struct modctl *m, dtracemdb_data_t *data)
725 {
726         struct module mod;

728         if (m->mod_mp == NULL)
729                 return (WALK_NEXT);

731         if (mdb_vread(&mod, sizeof (mod), (uintptr_t)m->mod_mp) == -1) {
732                 mdb_warn("couldn't read modctl %p's module", addr);
733                 return (WALK_NEXT);
734         }

736         if ((uintptr_t)mod.text > data->dtmd_addr)
737                 return (WALK_NEXT);

739         if ((uintptr_t)mod.text + mod.text_size <= data->dtmd_addr)
740                 return (WALK_NEXT);

742         if (mdb_readstr(data->dtmd_modstr, MDB_SYM_NAMLEN,
743             (uintptr_t)m->mod_modname) == -1)
744                 return (WALK_ERR);

746         return (WALK_DONE);
747 }

749 static int
750 dtracemdb_lookup_by_addr(void *varg, GElf_Addr addr, GElf_Sym *symp,
751     dtrace_syminfo_t *sip)
752 {
753         dtracemdb_data_t *data = varg;

755         if (data->dtmd_symstr == NULL) {
756                 data->dtmd_symstr = mdb_zalloc(MDB_SYM_NAMLEN,
757                     UM_SLEEP | UM_GC);
758         }

760         if (data->dtmd_modstr == NULL) {
761                 data->dtmd_modstr = mdb_zalloc(MDB_SYM_NAMLEN,
762                     UM_SLEEP | UM_GC);
763         }

765         if (symp != NULL) {
766                 if (mdb_lookup_by_addr(addr, MDB_SYM_FUZZY, data->dtmd_symstr,
767                     MDB_SYM_NAMLEN, symp) == -1)
768                         return (-1);
769         }

771         if (sip != NULL) {
772                 data->dtmd_addr = addr;

774                 (void) strcpy(data->dtmd_modstr, "???");

776                 if (mdb_walk("modctl",
777                     (mdb_walk_cb_t)dtracemdb_modctl, varg) == -1) {
778                         mdb_warn("couldn't walk 'modctl'");
779                         return (-1);
780                 }

782                 sip->dts_object = data->dtmd_modstr;
783                 sip->dts_id = 0;
784                 sip->dts_name = symp != NULL ? data->dtmd_symstr : NULL;
785         }

787         return (0);
```

```
788 }

790 /*ARGSUSED*/
791 static int
792 dtracemdb_stat(void *varg, processorid_t cpu)
793 {
794         GElf_Sym sym;
795         cpu_t c;
796         uintptr_t caddr, addr;

798         if (mdb_lookup_by_name("cpu", &sym) == -1) {
799                 mdb_warn("failed to find symbol for 'cpu'");
800                 return (-1);
801         }

803         if (cpu * sizeof (uintptr_t) > sym.st_size)
804                 return (-1);

806         addr = (uintptr_t)sym.st_value + cpu * sizeof (uintptr_t);

808         if (mdb_vread(&caddr, sizeof (caddr), addr) == -1) {
809                 mdb_warn("failed to read cpu[%d]", cpu);
810                 return (-1);
811         }

813         if (caddr == NULL)
814                 return (-1);

816         if (mdb_vread(&c, sizeof (c), caddr) == -1) {
817                 mdb_warn("failed to read cpu at %p", caddr);
818                 return (-1);
819         }

821         if (c.cpu_flags & CPU_POWEROFF) {
822                 return (P_POWEROFF);
823         } else if (c.cpu_flags & CPU_SPARE) {
824                 return (P_SPARE);
825         } else if (c.cpu_flags & CPU_FAULTED) {
826                 return (P_FAULTED);
827         } else if ((c.cpu_flags & (CPU_READY | CPU_OFFLINE)) != CPU_READY) {
828                 return (P_OFFLINE);
829         } else if (c.cpu_flags & CPU_ENABLE) {
830                 return (P_ONLINE);
831         } else {
832                 return (P_NOINTR);
833         }
834 }

836 /*ARGSUSED*/
837 static long
838 dtracemdb_sysconf(void *varg, int name)
839 {
840         int max_ncpus;
841         processorid_t max_cpuid;

843         switch (name) {
844         case _SC_CPUID_MAX:
845                 if (mdb_readvar(&max_cpuid, "max_cpuid") == -1) {
846                         mdb_warn("failed to read 'max_cpuid'");
847                         return (-1);
848                 }

850                 return (max_cpuid);

852         case _SC_NPROCESSORS_MAX:
853                 if (mdb_readvar(&max_ncpus, "max_ncpus") == -1) {
```



```
854                         mdb_warn("failed to read 'max_ncpus'");
855                         return (-1);
856                 }

858                 return (max_ncpus);

860         default:
861                 mdb_warn("unexpected sysconf code %d\n", name);
862                 return (-1);
863         }
864 }

866 const dtrace_vector_t dtrace_mdbops = {
867         dtracemdb_ioctl,
868         dtracemdb_lookup_by_addr,
869         dtracemdb_stat,
870         dtracemdb_sysconf
871 };

873 typedef struct dtrace_dcmddata {
874         dtrace_hdl_t *dtdd_dtp;
875         int dtdd_cpu;
876         int dtdd_quiet;
877         int dtdd_flowindent;
878         int dtdd_heading;
879         FILE *dtdd_output;
880 } dtrace_dcmddata_t;

882 /*
883  * Helper to grab all the content from a file, spit it into a string, and erase
884  * and reset the file.
885  */
886 static void
887 print_and_truncate_file(FILE *fp)
888 {
889         long len;
890         char *out;

892         /* flush, find length of file, seek to beginning, initialize buffer */
893         if (fflush(fp) || (len = ftell(fp)) < 0 ||
894             fseek(fp, 0, SEEK_SET) < 0) {
895                 mdb_warn("couldn't prepare DTrace output file: %d\n", errno);
896                 return;
897         }

899         out = mdb_alloc(len + 1, UM_SLEEP);
900         out[len] = '\0';

902         /* read file into buffer, truncate file, and seek to beginning */
903         if ((fread(out, len + 1, sizeof (char), fp) == 0 && ferror(fp)) ||
904             ftruncate(fileno(fp), 0) < 0 || fseek(fp, 0, SEEK_SET) < 0) {
905                 mdb_warn("couldn't read DTrace output file: %d\n", errno);
906                 mdb_free(out, len + 1);
907                 return;
908         }

910         mdb_printf("%s", out);
911         mdb_free(out, len + 1);
912 }

914 /*ARGSUSED*/
915 static int
916 dtrace_dcmdrec(const dtrace_probedata_t *data,
917     const dtrace_recdesc_t *rec, void *arg)
918 {
919         dtrace_dcmddata_t *dd = arg;
```

```
921         print_and_truncate_file(dd->dtdd_output);

923         if (rec == NULL) {
924                 /*
925                  * We have processed the final record; output the newline if
926                  * we're not in quiet mode.
927                  */
928                 if (!dd->dtdd_quiet)
929                         mdb_printf("\n");

931                 return (DTRACE_CONSUME_NEXT);
932         }

934         return (DTRACE_CONSUME_THIS);
935 }

937 /*ARGSUSED*/
938 static int
939 dtrace_dcmdprobe(const dtrace_probedata_t *data, void *arg)
940 {
941         dtrace_probedesc_t *pd = data->dtpda_pdesc;
942         processorid_t cpu = data->dtpda_cpu;
943         dtrace_dcmddata_t *dd = arg;
944         char name[DTRACE_FUNCNAMELEN + DTRACE_NAMELEN + 2];

946         if (dd->dtdd_cpu != -1UL && dd->dtdd_cpu != cpu)
947                 return (DTRACE_CONSUME_NEXT);

949         if (dd->dtdd_heading == 0) {
950                 if (!dd->dtdd_flowindent) {
951                         if (!dd->dtdd_quiet) {
952                                 mdb_printf("%3s %6s %32s\n",
953                                     "CPU", "ID", "FUNCTION:NAME");
954                         }
955                 } else {
956                         mdb_printf("%3s %-41s\n", "CPU", "FUNCTION");
957                 }
958                 dd->dtdd_heading = 1;
959         }

961         if (!dd->dtdd_flowindent) {
962                 if (!dd->dtdd_quiet) {
963                         (void) mdb_snprintf(name, sizeof (name), "%s:%s",
964                             pd->dtpd_func, pd->dtpd_name);

966                         mdb_printf("%3d %6d %32s ", cpu, pd->dtpd_id, name);
967                 }
968         } else {
969                 int indent = data->dtpda_indent;

971                 if (data->dtpda_flow == DTRACEFLOW_NONE) {
972                         (void) mdb_snprintf(name, sizeof (name), "%*s%s%s:%s",
973                             indent, "", data->dtpda_prefix, pd->dtpd_func,
974                             pd->dtpd_name);
975                 } else {
976                         (void) mdb_snprintf(name, sizeof (name), "%*s%s%s",
977                             indent, "", data->dtpda_prefix, pd->dtpd_func);
978                 }

980                 mdb_printf("%3d %-41s ", cpu, name);
981         }

983         return (DTRACE_CONSUME_THIS);
984 }
```



```
986 /*ARGSUSED*/
987 static int
988 dtrace_dcmderr(const dtrace_errdata_t *data, void *arg)
989 {
990         mdb_warn(data->dteda_msg);
991         return (DTRACE_HANDLE_OK);
992 }

994 /*ARGSUSED*/
995 static int
996 dtrace_dcmddrop(const dtrace_dropdata_t *data, void *arg)
997 {
998         mdb_warn(data->dtdda_msg);
999         return (DTRACE_HANDLE_OK);
1000 }

1002 /*ARGSUSED*/
1003 static int
1004 dtrace_dcmdbuffered(const dtrace_bufdata_t *bufdata, void *arg)
1005 {
1006         mdb_printf("%s", bufdata->dtbda_buffered);
1007         return (DTRACE_HANDLE_OK);
1008 }

1010 /*ARGSUSED*/
1011 int
1012 dtrace(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1013 {
1014         dtrace_state_t state;
1015         dtrace_hdl_t *dtp;
1016         int ncpu, err;
1017         uintptr_t c = -1UL;
1018         dtrace_dcmddata_t dd;
1019         dtrace_optval_t val;
1020         dtracemdb_data_t md;
1021         int rval = DCMD_ERR;
1022         dtrace_anon_t anon;

1024         if (!(flags & DCMD_ADDRSPEC))
1025                 return (DCMD_USAGE);

1027         if (mdb_getopts(argc, argv, 'c', MDB_OPT_UINTPTR, &c, NULL) != argc)
1028                 return (DCMD_USAGE);

1030         if (mdb_readvar(&ncpu, "_ncpu") == -1) {
1031                 mdb_warn("failed to read '_ncpu'");
1032                 return (DCMD_ERR);
1033         }

1035         if (mdb_vread(&state, sizeof (state), addr) == -1) {
1036                 mdb_warn("couldn't read dtrace_state_t at %p", addr);
1037                 return (DCMD_ERR);
1038         }

1040         if (state.dts_anon != NULL) {
1041                 addr = (uintptr_t)state.dts_anon;

1043                 if (mdb_vread(&state, sizeof (state), addr) == -1) {
1044                         mdb_warn("couldn't read anonymous state at %p", addr);
1045                         return (DCMD_ERR);
1046                 }
1047         }

1049         bzero(&md, sizeof (md));
1050         md.dtmd_state = &state;
```

```
1052         if ((dtp = dtrace_vopen(DTRACE_VERSION, DTRACE_O_NOSYS, &err,
1053             &dtrace_mdbops, &md)) == NULL) {
1054                 mdb_warn("failed to initialize dtrace: %s\n",
1055                     dtrace_errmsg(NULL, err));
1056                 return (DCMD_ERR);
1057         }
1059         /*
1060          * If this is the anonymous enabling, we need to set a bit indicating
1061          * that DTRACEOPT_GRABANON should be set.
1062          */
1063         if (mdb_readvar(&anon, "dtrace_anon") == -1) {
1064                 mdb_warn("failed to read 'dtrace_anon'");
1065                 return (DCMD_ERR);
1066         }
1068         md.dtmd_isanon = ((uintptr_t)anon.dta_state == addr);
1070         if (dtrace_go(dtp) != 0) {
1071                 mdb_warn("failed to initialize dtrace: %s\n",
1072                     dtrace_errmsg(dtp, dtrace_errno(dtp)));
1073                 goto err;
1074         }
1076         bzero(&dd, sizeof (dd));
1077         dd.dtdd_dtp = dtp;
1078         dd.dtdd_cpu = c;
1080         if (dtrace_getopt(dtp, "flowindent", &val) == -1) {
1081                 mdb_warn("couldn't get 'flowindent' option: %s\n",
1082                     dtrace_errmsg(dtp, dtrace_errno(dtp)));
1083                 goto err;
1084         }
1086         dd.dtdd_flowindent = (val != DTRACEOPT_UNSET);
1088         if (dtrace_getopt(dtp, "quiet", &val) == -1) {
1089                 mdb_warn("couldn't get 'quiet' option: %s\n",
1090                     dtrace_errmsg(dtp, dtrace_errno(dtp)));
1091                 goto err;
1092         }
1094         dd.dtdd_quiet = (val != DTRACEOPT_UNSET);
1096         if (dtrace_handle_err(dtp, dtrace_dcmderr, NULL) == -1) {
1097                 mdb_warn("couldn't add err handler: %s\n",
1098                     dtrace_errmsg(dtp, dtrace_errno(dtp)));
1099                 goto err;
1100         }
1102         if (dtrace_handle_drop(dtp, dtrace_dcmddrop, NULL) == -1) {
1103                 mdb_warn("couldn't add drop handler: %s\n",
1104                     dtrace_errmsg(dtp, dtrace_errno(dtp)));
1105                 goto err;
1106         }
1108         if (dtrace_handle_buffered(dtp, dtrace_dcmdbuffered, NULL) == -1) {
1109                 mdb_warn("couldn't add buffered handler: %s\n",
1110                     dtrace_errmsg(dtp, dtrace_errno(dtp)));
1111                 goto err;
1112         }
1114         if (dtrace_status(dtp) == -1) {
1115                 mdb_warn("couldn't get status: %s\n",
1116                     dtrace_errmsg(dtp, dtrace_errno(dtp)));
1117                 goto err;
```



```
1118         }
1120         if (dtrace_aggregate_snap(dtp) == -1) {
1121                 mdb_warn("couldn't snapshot aggregation: %s\n",
1122                     dtrace_errmsg(dtp, dtrace_errno(dtp)));
1123                 goto err;
1124         }
1126         if ((dd.dtdd_output = tmpfile()) == NULL) {
1127                 mdb_warn("couldn't open DTrace output file: %d\n", errno);
1128                 goto err;
1129         }
1131         if (dtrace_consume(dtp, dd.dtdd_output,
1132             dtrace_dcmdprobe, dtrace_dcmdrec, &dd) == -1) {
1133                 mdb_warn("couldn't consume DTrace buffers: %s\n",
1134                     dtrace_errmsg(dtp, dtrace_errno(dtp)));
1135         }
1137         if (dtrace_aggregate_print(dtp, NULL, NULL) == -1) {
1138                 mdb_warn("couldn't print aggregation: %s\n",
1139                     dtrace_errmsg(dtp, dtrace_errno(dtp)));
1140                 goto err;
1141         }
1143         rval = DCMD_OK;
1144 err:
1145         dtrace_close(dtp);
1146         fclose(dd.dtdd_output);
1147         return (rval);
1148 }
1150 static int
1151 dtrace_errhash_cmp(const void *l, const void *r)
1152 {
1153         uintptr_t lhs = *((uintptr_t *)l);
1154         uintptr_t rhs = *((uintptr_t *)r);
1155         dtrace_errhash_t lerr, rerr;
1156         char lmsg[256], rmsg[256];
1158         (void) mdb_vread(&lerr, sizeof (lerr), lhs);
1159         (void) mdb_vread(&rerr, sizeof (rerr), rhs);
1161         if (lerr.dter_msg == NULL)
1162                 return (-1);
1164         if (rerr.dter_msg == NULL)
1165                 return (1);
1167         (void) mdb_readstr(lmsg, sizeof (lmsg), (uintptr_t)lerr.dter_msg);
1168         (void) mdb_readstr(rmsg, sizeof (rmsg), (uintptr_t)rerr.dter_msg);
1170         return (strcmp(lmsg, rmsg));
1171 }
1173 int
1174 dtrace_errhash_init(mdb_walk_state_t *wsp)
1175 {
1176         GElf_Sym sym;
1177         uintptr_t *hash, addr;
1178         int i;
1180         if (wsp->walk_addr != NULL) {
1181                 mdb_warn("dtrace_errhash walk only supports global walks\n");
1182                 return (WALK_ERR);
1183         }
```

```
1185         if (mdb_lookup_by_name("dtrace_errhash", &sym) == -1) {
1186                 mdb_warn("couldn't find 'dtrace_errhash' (non-DEBUG kernel?)");
1187                 return (WALK_ERR);
1188         }

1190         addr = (uintptr_t)sym.st_value;
1191         hash = mdb_alloc(DTRACE_ERRHASHSZ * sizeof (uintptr_t),
1192             UM_SLEEP | UM_GC);

1194         for (i = 0; i < DTRACE_ERRHASHSZ; i++)
1195                 hash[i] = addr + i * sizeof (dtrace_errhash_t);

1197         qsort(hash, DTRACE_ERRHASHSZ, sizeof (uintptr_t), dtrace_errhash_cmp);

1199         wsp->walk_addr = 0;
1200         wsp->walk_data = hash;

1202         return (WALK_NEXT);
1203 }

1205 int
1206 dtrace_errhash_step(mdb_walk_state_t *wsp)
1207 {
1208         int ndx = (int)wsp->walk_addr;
1209         uintptr_t *hash = wsp->walk_data;
1210         dtrace_errhash_t err;
1211         uintptr_t addr;

1213         if (ndx >= DTRACE_ERRHASHSZ)
1214                 return (WALK_DONE);

1216         wsp->walk_addr = ndx + 1;
1217         addr = hash[ndx];

1219         if (mdb_vread(&err, sizeof (err), addr) == -1) {
1220                 mdb_warn("failed to read dtrace_errhash_t at %p", addr);
1221                 return (WALK_DONE);
1222         }

1224         if (err.dter_msg == NULL)
1225                 return (WALK_NEXT);

1227         return (wsp->walk_callback(addr, &err, wsp->walk_cbdata));
1228 }

1230 /*ARGSUSED*/
1231 int
1232 dtrace_errhash(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
1233 {
1234         dtrace_errhash_t err;
1235         char msg[256];

1237         if (!(flags & DCMD_ADDRSPEC)) {
1238                 if (mdb_walk_dcmd("dtrace_errhash", "dtrace_errhash",
1239                     argc, argv) == -1) {
1240                         mdb_warn("can't walk 'dtrace_errhash'");
1241                         return (DCMD_ERR);
1242                 }

1244                 return (DCMD_OK);
1245         }

1247         if (DCMD_HDRSPEC(flags))
1248                 mdb_printf("%8s %s\n", "COUNT", "ERROR");
```



```
1250         if (mdb_vread(&err, sizeof (err), addr) == -1) {
1251                 mdb_warn("failed to read dtrace_errhash_t at %p", addr);
1252                 return (DCMD_ERR);
1253         }

1255         addr = (uintptr_t)err.dter_msg;

1257         if (mdb_readstr(msg, sizeof (msg), addr) == -1) {
1258                 mdb_warn("failed to read error msg at %p", addr);
1259                 return (DCMD_ERR);
1260         }

1262         mdb_printf("%8d %s", err.dter_count, msg);

1264         /*
1265          * Some error messages include a newline -- only print the newline
1266          * if the message doesn't have one.
1267          */
1268         if (msg[strlen(msg) - 1] != '\n')
1269                 mdb_printf("\n");

1271         return (DCMD_OK);
1272 }

1274 int
1275 dtrace_helptrace_init(mdb_walk_state_t *wsp)
1276 {
1277         uint32_t next;
1278         uintptr_t buffer;
  25         int enabled;

1280         if (wsp->walk_addr != NULL) {
1281                 mdb_warn("dtrace_helptrace only supports global walks\n");
1282                 return (WALK_ERR);
1283         }

1285         if (mdb_readvar(&buffer, "dtrace_helptrace_buffer") == -1) {
1286                 mdb_warn("couldn't read 'dtrace_helptrace_buffer'");
  32         if (mdb_readvar(&enabled, "dtrace_helptrace_enabled") == -1) {
  33                 mdb_warn("couldn't read 'dtrace_helptrace_enabled'");
1287                 return (WALK_ERR);
1288         }

1290         if (buffer == NULL) {
  37         if (!enabled) {
1291                 mdb_warn("helper tracing is not enabled\n");
1292                 return (WALK_ERR);
1293         }

1295         if (mdb_readvar(&next, "dtrace_helptrace_next") == -1) {
1296                 mdb_warn("couldn't read 'dtrace_helptrace_next'");
1297                 return (WALK_ERR);
1298         }

1300         wsp->walk_addr = next;

1302         return (WALK_NEXT);
1303 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
  438208 Tue Jan 14 16:47:26 2014
new/usr/src/uts/common/dtrace/dtrace.c
4469 DTrace helper tracing should be dynamic
**********************************************************
```

_____unchanged_portion_omitted_

```
 266 static dtrace_id_t      dtrace_probeid_begin;   /* special BEGIN probe */
 267 static dtrace_id_t      dtrace_probeid_end;     /* special END probe */
 268 dtrace_id_t             dtrace_probeid_error;   /* special ERROR probe */

 270 /*
 271  * DTrace Helper Tracing Variables
 272  *
 273  * These variables should be set dynamically to enable helper tracing.  The
 274  * only variables that should be set are dtrace_helptrace_enable (which should
 275  * be set to a non-zero value to allocate helper tracing buffers on the next
 276  * open of /dev/dtrace) and dtrace_helptrace_disable (which should be set to a
 277  * non-zero value to deallocate helper tracing buffers on the next close of
 278  * /dev/dtrace).  When (and only when) helper tracing is disabled, the
 279  * buffer size may also be set via dtrace_helptrace_bufsize.
 280  */
 281 int                     dtrace_helptrace_enable = 0;
 282 int                     dtrace_helptrace_disable = 0;
 283 int                     dtrace_helptrace_bufsize = 16 * 1024 * 1024;
 272  */
 273 uint32_t dtrace_helptrace_next = 0;
 284 uint32_t                dtrace_helptrace_nlocals;
 285 static dtrace_helptrace_t *dtrace_helptrace_buffer;
 286 static uint32_t         dtrace_helptrace_next = 0;
 287 static int              dtrace_helptrace_wrapped = 0;
 275 char    *dtrace_helptrace_buffer;
 276 int     dtrace_helptrace_bufsize = 512 * 1024;

 278 #ifdef DEBUG
 279 int     dtrace_helptrace_enabled = 1;
 280 #else
 281 int     dtrace_helptrace_enabled = 0;
 282 #endif

 289 /*
 290  * DTrace Error Hashing
 291  *
 292  * On DEBUG kernels, DTrace will track the errors that has seen in a hash
 293  * table.  This is very useful for checking coverage of tests that are
 294  * expected to induce DIF or DOF processing errors, and may be useful for
 295  * debugging problems in the DIF code generator or in DOF generation .  The
 296  * error hash may be examined with the ::dtrace_errhash MDB dcmd.
 297  */
 298 #ifdef DEBUG
 299 static dtrace_errhash_t dtrace_errhash[DTRACE_ERRHASHSZ];
 300 static const char *dtrace_errlast;
 301 static kthread_t *dtrace_errthread;
 302 static kmutex_t dtrace_errlock;
 303 #endif

 305 /*
 306  * DTrace Macros and Constants
 307  *
 308  * These are various macros that are useful in various spots in the
 309  * implementation, along with a few random constants that have no meaning
 310  * outside of the implementation.  There is no real structure to this cpp
 311  * mishmash -- but is there ever?
 312  */
 313 #define DTRACE_HASHSTR(hash, probe)     \
 314         dtrace_hash_str(*((char **)((uintptr_t)(probe) + (hash)->dth_stroffs)))
```



```
 316 #define DTRACE_HASHNEXT(hash, probe)    \
 317         (dtrace_probe_t **)((uintptr_t)(probe) + (hash)->dth_nextoffs)

 319 #define DTRACE_HASHPREV(hash, probe)    \
 320         (dtrace_probe_t **)((uintptr_t)(probe) + (hash)->dth_prevoffs)

 322 #define DTRACE_HASHEQ(hash, lhs, rhs)   \
 323         (strcmp(*((char **)((uintptr_t)(lhs) + (hash)->dth_stroffs)), \
 324             *((char **)((uintptr_t)(rhs) + (hash)->dth_stroffs))) == 0)

 326 #define DTRACE_AGGHASHSIZE_SLEW         17

 328 #define DTRACE_V4MAPPED_OFFSET          (sizeof (uint32_t) * 3)

 330 /*
 331  * The key for a thread-local variable consists of the lower 61 bits of the
 332  * t_did, plus the 3 bits of the highest active interrupt above LOCK_LEVEL.
 333  * We add DIF_VARIABLE_MAX to t_did to assure that the thread key is never
 334  * equal to a variable identifier.  This is necessary (but not sufficient) to
 335  * assure that global associative arrays never collide with thread-local
 336  * variables.  To guarantee that they cannot collide, we must also define the
 337  * order for keying dynamic variables.  That order is:
 338  *
 339  *   [ key0 ] ... [ keyn ] [ variable-key ] [ tls-key ]
 340  *
 341  * Because the variable-key and the tls-key are in orthogonal spaces, there is
 342  * no way for a global variable key signature to match a thread-local key
 343  * signature.
 344  */
 345 #define DTRACE_TLS_THRKEY(where) { \
 346         uint_t intr = 0; \
 347         uint_t actv = CPU->cpu_intr_actv >> (LOCK_LEVEL + 1); \
 348         for (; actv; actv >>= 1) \
 349                 intr++; \
 350         ASSERT(intr < (1 << 3)); \
 351         (where) = ((curthread->t_did + DIF_VARIABLE_MAX) & \
 352             (((uint64_t)1 << 61) - 1)) | ((uint64_t)intr << 61); \
 353 }
```

_____unchanged_portion_omitted_

```
14321 /*
14322  * DTrace Helper Functions
14323  */
14324 static void
14325 dtrace_helper_trace(dtrace_helper_action_t *helper,
14326     dtrace_mstate_t *mstate, dtrace_vstate_t *vstate, int where)
14327 {
14328         uint32_t size, next, nnext, i;
14329         dtrace_helptrace_t *ent, *buffer;
14324         dtrace_helptrace_t *ent;
14330         uint16_t flags = cpu_core[CPU->cpu_id].cpuc_dtrace_flags;

14332         if ((buffer = dtrace_helptrace_buffer) == NULL)
14327         if (!dtrace_helptrace_enabled)
14333                 return;

14335         ASSERT(vstate->dtvs_nlocals <= dtrace_helptrace_nlocals);

14337         /*
14338          * What would a tracing framework be without its own tracing
14339          * framework?  (Well, a hell of a lot simpler, for starters...)
14340          */
14341         size = sizeof (dtrace_helptrace_t) + dtrace_helptrace_nlocals *
14342             sizeof (uint64_t) - sizeof (uint64_t);
```

```
14344         /*
14345          * Iterate until we can allocate a slot in the trace buffer.
14346          */
14347         do {
14348                 next = dtrace_helptrace_next;
14349
14350                 if (next + size < dtrace_helptrace_bufsize) {
14351                         nnext = next + size;
14352                 } else {
14353                         nnext = size;
14354                 }
14355         } while (dtrace_cas32(&dtrace_helptrace_next, next, nnext) != next);
14356
14357         /*
14358          * We have our slot; fill it in.
14359          */
14360         if (nnext == size) {
14361                 dtrace_helptrace_wrapped++;
14355         if (nnext == size)
14362                 next = 0;
14363         }
14364 #endif /* ! codereview */
14365
14366         ent = (dtrace_helptrace_t *)((uintptr_t)buffer + next);
14357         ent = (dtrace_helptrace_t *)&dtrace_helptrace_buffer[next];
14367         ent->dtht_helper = helper;
14368         ent->dtht_where = where;
14369         ent->dtht_nlocals = vstate->dtvs_nlocals;
14370
14371         ent->dtht_fltoffs = (mstate->dtms_present & DTRACE_MSTATE_FLTOFFS) ?
14372             mstate->dtms_fltoffs : -1;
14373         ent->dtht_fault = DTRACE_FLAGS2FLT(flags);
14374         ent->dtht_illval = cpu_core[CPU->cpu_id].cpuc_dtrace_illval;
14375
14376         for (i = 0; i < vstate->dtvs_nlocals; i++) {
14377                 dtrace_statvar_t *svar;
14378
14379                 if ((svar = vstate->dtvs_locals[i]) == NULL)
14380                         continue;
14381
14382                 ASSERT(svar->dtsv_size >= NCPU * sizeof (uint64_t));
14383                 ent->dtht_locals[i] =
14384                     ((uint64_t *)(uintptr_t)svar->dtsv_data)[CPU->cpu_id];
14385         }
14386 }
14387
14388 static uint64_t
14389 dtrace_helper(int which, dtrace_mstate_t *mstate,
14390     dtrace_state_t *state, uint64_t arg0, uint64_t arg1)
14391 {
14392         uint16_t *flags = &cpu_core[CPU->cpu_id].cpuc_dtrace_flags;
14393         uint64_t sarg0 = mstate->dtms_arg[0];
14394         uint64_t sarg1 = mstate->dtms_arg[1];
14395         uint64_t rval;
14396         dtrace_helpers_t *helpers = curproc->p_dtrace_helpers;
14397         dtrace_helper_action_t *helper;
14398         dtrace_vstate_t *vstate;
14399         dtrace_difo_t *pred;
14400         int i, trace = dtrace_helptrace_buffer != NULL;
14391         int i, trace = dtrace_helptrace_enabled;
14401
14402         ASSERT(which >= 0 && which < DTRACE_NHELPER_ACTIONS);
14403
14404         if (helpers == NULL)
14405                 return (0);
```



```
14407         if ((helper = helpers->dthps_actions[which]) == NULL)
14408                 return (0);
14409
14410         vstate = &helpers->dthps_vstate;
14411         mstate->dtms_arg[0] = arg0;
14412         mstate->dtms_arg[1] = arg1;
14413
14414         /*
14415          * Now iterate over each helper.  If its predicate evaluates to 'true',
14416          * we'll call the corresponding actions.  Note that the below calls
14417          * to dtrace_dif_emulate() may set faults in machine state.  This is
14418          * okay:  our caller (the outer dtrace_dif_emulate()) will simply plow
14419          * the stored DIF offset with its own (which is the desired behavior).
14420          * Also, note the calls to dtrace_dif_emulate() may allocate scratch
14421          * from machine state; this is okay, too.
14422          */
14423         for (; helper != NULL; helper = helper->dtha_next) {
14424                 if ((pred = helper->dtha_predicate) != NULL) {
14425                         if (trace)
14426                                 dtrace_helper_trace(helper, mstate, vstate, 0);
14427
14428                         if (!dtrace_dif_emulate(pred, mstate, vstate, state))
14429                                 goto next;
14430
14431                         if (*flags & CPU_DTRACE_FAULT)
14432                                 goto err;
14433                 }
14434
14435                 for (i = 0; i < helper->dtha_nactions; i++) {
14436                         if (trace)
14437                                 dtrace_helper_trace(helper,
14438                                     mstate, vstate, i + 1);
14439
14440                         rval = dtrace_dif_emulate(helper->dtha_actions[i],
14441                             mstate, vstate, state);
14442
14443                         if (*flags & CPU_DTRACE_FAULT)
14444                                 goto err;
14445                 }
14446
14447 next:
14448                 if (trace)
14449                         dtrace_helper_trace(helper, mstate, vstate,
14450                             DTRACE_HELPTRACE_NEXT);
14451         }
14452
14453         if (trace)
14454                 dtrace_helper_trace(helper, mstate, vstate,
14455                     DTRACE_HELPTRACE_DONE);
14456
14457         /*
14458          * Restore the arg0 that we saved upon entry.
14459          */
14460         mstate->dtms_arg[0] = sarg0;
14461         mstate->dtms_arg[1] = sarg1;
14462
14463         return (rval);
14464
14465 err:
14466         if (trace)
14467                 dtrace_helper_trace(helper, mstate, vstate,
14468                     DTRACE_HELPTRACE_ERR);
14469
14470         /*
14471          * Restore the arg0 that we saved upon entry.
14472          */
```

```
14473         mstate->dtms_arg[0] = sarg0;
14474         mstate->dtms_arg[1] = sarg1;

14476         return (NULL);
14477 }
______unchanged_portion_omitted_

15603 /*
15604  * DTrace Driver Cookbook Functions
15605  */
15606 /*ARGSUSED*/
15607 static int
15608 dtrace_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
15609 {
15610         dtrace_provider_id_t id;
15611         dtrace_state_t *state = NULL;
15612         dtrace_enabling_t *enab;

15614         mutex_enter(&cpu_lock);
15615         mutex_enter(&dtrace_provider_lock);
15616         mutex_enter(&dtrace_lock);

15618         if (ddi_soft_state_init(&dtrace_softstate,
15619             sizeof (dtrace_state_t), 0) != 0) {
15620                 cmn_err(CE_NOTE, "/dev/dtrace failed to initialize soft state");
15621                 mutex_exit(&cpu_lock);
15622                 mutex_exit(&dtrace_provider_lock);
15623                 mutex_exit(&dtrace_lock);
15624                 return (DDI_FAILURE);
15625         }

15627         if (ddi_create_minor_node(devi, DTRACEMNR_DTRACE, S_IFCHR,
15628             DTRACEMNRN_DTRACE, DDI_PSEUDO, NULL) == DDI_FAILURE ||
15629             ddi_create_minor_node(devi, DTRACEMNR_HELPER, S_IFCHR,
15630             DTRACEMNRN_HELPER, DDI_PSEUDO, NULL) == DDI_FAILURE) {
15631                 cmn_err(CE_NOTE, "/dev/dtrace couldn't create minor nodes");
15632                 ddi_remove_minor_node(devi, NULL);
15633                 ddi_soft_state_fini(&dtrace_softstate);
15634                 mutex_exit(&cpu_lock);
15635                 mutex_exit(&dtrace_provider_lock);
15636                 mutex_exit(&dtrace_lock);
15637                 return (DDI_FAILURE);
15638         }

15640         ddi_report_dev(devi);
15641         dtrace_devi = devi;

15643         dtrace_modload = dtrace_module_loaded;
15644         dtrace_modunload = dtrace_module_unloaded;
15645         dtrace_cpu_init = dtrace_cpu_setup_initial;
15646         dtrace_helpers_cleanup = dtrace_helpers_destroy;
15647         dtrace_helpers_fork = dtrace_helpers_duplicate;
15648         dtrace_cpustart_init = dtrace_suspend;
15649         dtrace_cpustart_fini = dtrace_resume;
15650         dtrace_debugger_init = dtrace_suspend;
15651         dtrace_debugger_fini = dtrace_resume;

15653         register_cpu_setup_func((cpu_setup_func_t *)dtrace_cpu_setup, NULL);

15655         ASSERT(MUTEX_HELD(&cpu_lock));

15657         dtrace_arena = vmem_create("dtrace", (void *)1, UINT32_MAX, 1,
15658             NULL, NULL, NULL, 0, VM_SLEEP | VMC_IDENTIFIER);
15659         dtrace_minor = vmem_create("dtrace_minor", (void *)DTRACEMNRN_CLONE,
15660             UINT32_MAX - DTRACEMNRN_CLONE, 1, NULL, NULL, NULL, 0,
15661             VM_SLEEP | VMC_IDENTIFIER);
```



```
15662         dtrace_taskq = taskq_create("dtrace_taskq", 1, maxclsyspri,
15663             1, INT_MAX, 0);

15665         dtrace_state_cache = kmem_cache_create("dtrace_state_cache",
15666             sizeof (dtrace_dstate_percpu_t) * NCPU, DTRACE_STATE_ALIGN,
15667             NULL, NULL, NULL, NULL, NULL, 0);

15669         ASSERT(MUTEX_HELD(&cpu_lock));
15670         dtrace_bymod = dtrace_hash_create(offsetof(dtrace_probe_t, dtpr_mod),
15671             offsetof(dtrace_probe_t, dtpr_nextmod),
15672             offsetof(dtrace_probe_t, dtpr_prevmod));

15674         dtrace_byfunc = dtrace_hash_create(offsetof(dtrace_probe_t, dtpr_func),
15675             offsetof(dtrace_probe_t, dtpr_nextfunc),
15676             offsetof(dtrace_probe_t, dtpr_prevfunc));

15678         dtrace_byname = dtrace_hash_create(offsetof(dtrace_probe_t, dtpr_name),
15679             offsetof(dtrace_probe_t, dtpr_nextname),
15680             offsetof(dtrace_probe_t, dtpr_prevname));

15682         if (dtrace_retain_max < 1) {
15683                 cmn_err(CE_WARN, "illegal value (%lu) for dtrace_retain_max; "
15684                     "setting to 1", dtrace_retain_max);
15685                 dtrace_retain_max = 1;
15686         }

15688         /*
15689          * Now discover our toxic ranges.
15690          */
15691         dtrace_toxic_ranges(dtrace_toxrange_add);

15693         /*
15694          * Before we register ourselves as a provider to our own framework,
15695          * we would like to assert that dtrace_provider is NULL -- but that's
15696          * not true if we were loaded as a dependency of a DTrace provider.
15697          * Once we've registered, we can assert that dtrace_provider is our
15698          * pseudo provider.
15699          */
15700         (void) dtrace_register("dtrace", &dtrace_provider_attr,
15701             DTRACE_PRIV_NONE, 0, &dtrace_provider_ops, NULL, &id);

15703         ASSERT(dtrace_provider != NULL);
15704         ASSERT((dtrace_provider_id_t)dtrace_provider == id);

15706         dtrace_probeid_begin = dtrace_probe_create((dtrace_provider_id_t)
15707             dtrace_provider, NULL, NULL, "BEGIN", 0, NULL);
15708         dtrace_probeid_end = dtrace_probe_create((dtrace_provider_id_t)
15709             dtrace_provider, NULL, NULL, "END", 0, NULL);
15710         dtrace_probeid_error = dtrace_probe_create((dtrace_provider_id_t)
15711             dtrace_provider, NULL, NULL, "ERROR", 1, NULL);

15713         dtrace_anon_property();
15714         mutex_exit(&cpu_lock);

15716         /*
15708          * If DTrace helper tracing is enabled, we need to allocate the
15709          * trace buffer and initialize the values.
15710          */
15711         if (dtrace_helptrace_enabled) {
15712                 ASSERT(dtrace_helptrace_buffer == NULL);
15713                 dtrace_helptrace_buffer =
15714                     kmem_zalloc(dtrace_helptrace_bufsize, KM_SLEEP);
15715                 dtrace_helptrace_next = 0;
15716         }

15718         /*
```

```
15717          * If there are already providers, we must ask them to provide their
15718          * probes, and then match any anonymous enabling against them.  Note
15719          * that there should be no other retained enablings at this time:
15720          * the only retained enablings at this time should be the anonymous
15721          * enabling.
15722          */
15723         if (dtrace_anon.dta_enabling != NULL) {
15724                 ASSERT(dtrace_retained == dtrace_anon.dta_enabling);

15726                 dtrace_enabling_provide(NULL);
15727                 state = dtrace_anon.dta_state;

15729                 /*
15730                  * We couldn't hold cpu_lock across the above call to
15731                  * dtrace_enabling_provide(), but we must hold it to actually
15732                  * enable the probes.  We have to drop all of our locks, pick
15733                  * up cpu_lock, and regain our locks before matching the
15734                  * retained anonymous enabling.
15735                  */
15736                 mutex_exit(&dtrace_lock);
15737                 mutex_exit(&dtrace_provider_lock);

15739                 mutex_enter(&cpu_lock);
15740                 mutex_enter(&dtrace_provider_lock);
15741                 mutex_enter(&dtrace_lock);

15743                 if ((enab = dtrace_anon.dta_enabling) != NULL)
15744                         (void) dtrace_enabling_match(enab, NULL);

15746                 mutex_exit(&cpu_lock);
15747         }

15749         mutex_exit(&dtrace_lock);
15750         mutex_exit(&dtrace_provider_lock);

15752         if (state != NULL) {
15753                 /*
15754                  * If we created any anonymous state, set it going now.
15755                  */
15756                 (void) dtrace_state_go(state, &dtrace_anon.dta_beganon);
15757         }

15759         return (DDI_SUCCESS);
15760 }

15762 /*ARGSUSED*/
15763 static int
15764 dtrace_open(dev_t *devp, int flag, int otyp, cred_t *cred_p)
15765 {
15766         dtrace_state_t *state;
15767         uint32_t priv;
15768         uid_t uid;
15769         zoneid_t zoneid;

15771         if (getminor(*devp) == DTRACEMNRN_HELPER)
15772                 return (0);

15774         /*
15775          * If this wasn't an open with the "helper" minor, then it must be
15776          * the "dtrace" minor.
15777          */
15778         if (getminor(*devp) != DTRACEMNRN_DTRACE)
15779                 return (ENXIO);

15781         /*
15782          * If no DTRACE_PRIV_* bits are set in the credential, then the
```



```
15783          * caller lacks sufficient permission to do anything with DTrace.
15784          */
15785         dtrace_cred2priv(cred_p, &priv, &uid, &zoneid);
15786         if (priv == DTRACE_PRIV_NONE)
15787                 return (EACCES);

15789         /*
15790          * Ask all providers to provide all their probes.
15791          */
15792         mutex_enter(&dtrace_provider_lock);
15793         dtrace_probe_provide(NULL, NULL);
15794         mutex_exit(&dtrace_provider_lock);

15796         mutex_enter(&cpu_lock);
15797         mutex_enter(&dtrace_lock);
15798         dtrace_opens++;
15799         dtrace_membar_producer();

15801         /*
15802          * If the kernel debugger is active (that is, if the kernel debugger
15803          * modified text in some way), we won't allow the open.
15804          */
15805         if (kdi_dtrace_set(KDI_DTSET_DTRACE_ACTIVATE) != 0) {
15806                 dtrace_opens--;
15807                 mutex_exit(&cpu_lock);
15808                 mutex_exit(&dtrace_lock);
15809                 return (EBUSY);
15810         }

15812         if (dtrace_helptrace_enable && dtrace_helptrace_buffer == NULL) {
15813                 /*
15814                  * If DTrace helper tracing is enabled, we need to allocate the
15815                  * trace buffer and initialize the values.
15816                  */
15817                 dtrace_helptrace_buffer =
15818                     kmem_zalloc(dtrace_helptrace_bufsize, KM_SLEEP);
15819                 dtrace_helptrace_next = 0;
15820                 dtrace_helptrace_wrapped = 0;
15821                 dtrace_helptrace_enable = 0;
15822         }

15824 #endif /* ! codereview */
15825         state = dtrace_state_create(devp, cred_p);
15826         mutex_exit(&cpu_lock);

15828         if (state == NULL) {
15829                 if (--dtrace_opens == 0 && dtrace_anon.dta_enabling == NULL)
15830                         (void) kdi_dtrace_set(KDI_DTSET_DTRACE_DEACTIVATE);
15831                 mutex_exit(&dtrace_lock);
15832                 return (EAGAIN);
15833         }

15835         mutex_exit(&dtrace_lock);

15837         return (0);
15838 }

15840 /*ARGSUSED*/
15841 static int
15842 dtrace_close(dev_t dev, int flag, int otyp, cred_t *cred_p)
15843 {
15844         minor_t minor = getminor(dev);
15845         dtrace_state_t *state;
15846         dtrace_helptrace_t *buf = NULL;
15847 #endif /* ! codereview */
```

```
15849         if (minor == DTRACEMNRN_HELPER)
15850                 return (0);

15852         state = ddi_get_soft_state(dtrace_softstate, minor);

15854         mutex_enter(&cpu_lock);
15855         mutex_enter(&dtrace_lock);

15857         if (state->dts_anon) {
15858                 /*
15859                  * There is anonymous state. Destroy that first.
15860                  */
15861                 ASSERT(dtrace_anon.dta_state == NULL);
15862                 dtrace_state_destroy(state->dts_anon);
15863         }

15865         if (dtrace_helptrace_disable) {
15866                 /*
15867                  * If we have been told to disable helper tracing, set the
15868                  * buffer to NULL before calling into dtrace_state_destroy();
15869                  * we take advantage of its dtrace_sync() to know that no
15870                  * CPU is in probe context with enabled helper tracing
15871                  * after it returns.
15872                  */
15873                 buf = dtrace_helptrace_buffer;
15874                 dtrace_helptrace_buffer = NULL;
15875         }

15877 #endif /* ! codereview */
15878         dtrace_state_destroy(state);
15879         ASSERT(dtrace_opens > 0);

15881         /*
15882          * Only relinquish control of the kernel debugger interface when there
15883          * are no consumers and no anonymous enablings.
15884          */
15885         if (--dtrace_opens == 0 && dtrace_anon.dta_enabling == NULL)
15886                 (void) kdi_dtrace_set(KDI_DTSET_DTRACE_DEACTIVATE);

15888         if (buf != NULL) {
15889                 kmem_free(buf, dtrace_helptrace_bufsize);
15890                 dtrace_helptrace_disable = 0;
15891         }

15893 #endif /* ! codereview */
15894         mutex_exit(&dtrace_lock);
15895         mutex_exit(&cpu_lock);

15897         return (0);
15898 }

15900 /*ARGSUSED*/
15901 static int
15902 dtrace_ioctl_helper(int cmd, intptr_t arg, int *rv)
15903 {
15904         int rval;
15905         dof_helper_t help, *dhp = NULL;

15907         switch (cmd) {
15908         case DTRACEHIOC_ADDDOF:
15909                 if (copyin((void *)arg, &help, sizeof (help)) != 0) {
15910                         dtrace_dof_error(NULL, "failed to copyin DOF helper");
15911                         return (EFAULT);
15912                 }

15914                 dhp = &help;
```



```
15915                 arg = (intptr_t)help.dofhp_dof;
15916                 /*FALLTHROUGH*/

15918         case DTRACEHIOC_ADD: {
15919                 dof_hdr_t *dof = dtrace_dof_copyin(arg, &rval);

15921                 if (dof == NULL)
15922                         return (rval);

15924                 mutex_enter(&dtrace_lock);

15926                 /*
15927                  * dtrace_helper_slurp() takes responsibility for the dof --
15928                  * it may free it now or it may save it and free it later.
15929                  */
15930                 if ((rval = dtrace_helper_slurp(dof, dhp)) != -1) {
15931                         *rv = rval;
15932                         rval = 0;
15933                 } else {
15934                         rval = EINVAL;
15935                 }

15937                 mutex_exit(&dtrace_lock);
15938                 return (rval);
15939         }

15941         case DTRACEHIOC_REMOVE: {
15942                 mutex_enter(&dtrace_lock);
15943                 rval = dtrace_helper_destroygen(arg);
15944                 mutex_exit(&dtrace_lock);

15946                 return (rval);
15947         }

15949         default:
15950                 break;
15951         }

15953         return (ENOTTY);
15954 }

15956 /*ARGSUSED*/
15957 static int
15958 dtrace_ioctl(dev_t dev, int cmd, intptr_t arg, int md, cred_t *cr, int *rv)
15959 {
15960         minor_t minor = getminor(dev);
15961         dtrace_state_t *state;
15962         int rval;

15964         if (minor == DTRACEMNRN_HELPER)
15965                 return (dtrace_ioctl_helper(cmd, arg, rv));

15967         state = ddi_get_soft_state(dtrace_softstate, minor);

15969         if (state->dts_anon) {
15970                 ASSERT(dtrace_anon.dta_state == NULL);
15971                 state = state->dts_anon;
15972         }

15974         switch (cmd) {
15975         case DTRACEIOC_PROVIDER: {
15976                 dtrace_providerdesc_t pvd;
15977                 dtrace_provider_t *pvp;

15979                 if (copyin((void *)arg, &pvd, sizeof (pvd)) != 0)
15980                         return (EFAULT);
```

```
15982                 pvd.dtvd_name[DTRACE_PROVNAMELEN - 1] = '\0';
15983                 mutex_enter(&dtrace_provider_lock);

15985                 for (pvp = dtrace_provider; pvp != NULL; pvp = pvp->dtpv_next) {
15986                         if (strcmp(pvp->dtpv_name, pvd.dtvd_name) == 0)
15987                                 break;
15988                 }

15990                 mutex_exit(&dtrace_provider_lock);

15992                 if (pvp == NULL)
15993                         return (ESRCH);

15995                 bcopy(&pvp->dtpv_priv, &pvd.dtvd_priv, sizeof (dtrace_ppriv_t));
15996                 bcopy(&pvp->dtpv_attr, &pvd.dtvd_attr, sizeof (dtrace_pattr_t));
15997                 if (copyout(&pvd, (void *)arg, sizeof (pvd)) != 0)
15998                         return (EFAULT);

16000                 return (0);
16001         }

16003         case DTRACEIOC_EPROBE: {
16004                 dtrace_eprobedesc_t epdesc;
16005                 dtrace_ecb_t *ecb;
16006                 dtrace_action_t *act;
16007                 void *buf;
16008                 size_t size;
16009                 uintptr_t dest;
16010                 int nrecs;

16012                 if (copyin((void *)arg, &epdesc, sizeof (epdesc)) != 0)
16013                         return (EFAULT);

16015                 mutex_enter(&dtrace_lock);

16017                 if ((ecb = dtrace_epid2ecb(state, epdesc.dtepd_epid)) == NULL) {
16018                         mutex_exit(&dtrace_lock);
16019                         return (EINVAL);
16020                 }

16022                 if (ecb->dte_probe == NULL) {
16023                         mutex_exit(&dtrace_lock);
16024                         return (EINVAL);
16025                 }

16027                 epdesc.dtepd_probeid = ecb->dte_probe->dtpr_id;
16028                 epdesc.dtepd_uarg = ecb->dte_uarg;
16029                 epdesc.dtepd_size = ecb->dte_size;

16031                 nrecs = epdesc.dtepd_nrecs;
16032                 epdesc.dtepd_nrecs = 0;
16033                 for (act = ecb->dte_action; act != NULL; act = act->dta_next) {
16034                         if (DTRACEACT_ISAGG(act->dta_kind) || act->dta_intuple)
16035                                 continue;

16037                         epdesc.dtepd_nrecs++;
16038                 }

16040                 /*
16041                  * Now that we have the size, we need to allocate a temporary
16042                  * buffer in which to store the complete description.  We need
16043                  * the temporary buffer to be able to drop dtrace_lock()
16044                  * across the copyout(), below.
16045                  */
16046                 size = sizeof (dtrace_eprobedesc_t) +
```



```
16047                     (epdesc.dtepd_nrecs * sizeof (dtrace_recdesc_t));

16049                 buf = kmem_alloc(size, KM_SLEEP);
16050                 dest = (uintptr_t)buf;

16052                 bcopy(&epdesc, (void *)dest, sizeof (epdesc));
16053                 dest += offsetof(dtrace_eprobedesc_t, dtepd_rec[0]);

16055                 for (act = ecb->dte_action; act != NULL; act = act->dta_next) {
16056                         if (DTRACEACT_ISAGG(act->dta_kind) || act->dta_intuple)
16057                                 continue;

16059                         if (nrecs-- == 0)
16060                                 break;

16062                         bcopy(&act->dta_rec, (void *)dest,
16063                             sizeof (dtrace_recdesc_t));
16064                         dest += sizeof (dtrace_recdesc_t);
16065                 }

16067                 mutex_exit(&dtrace_lock);

16069                 if (copyout(buf, (void *)arg, dest - (uintptr_t)buf) != 0) {
16070                         kmem_free(buf, size);
16071                         return (EFAULT);
16072                 }

16074                 kmem_free(buf, size);
16075                 return (0);
16076         }

16078         case DTRACEIOC_AGGDESC: {
16079                 dtrace_aggdesc_t aggdesc;
16080                 dtrace_action_t *act;
16081                 dtrace_aggregation_t *agg;
16082                 int nrecs;
16083                 uint32_t offs;
16084                 dtrace_recdesc_t *lrec;
16085                 void *buf;
16086                 size_t size;
16087                 uintptr_t dest;

16089                 if (copyin((void *)arg, &aggdesc, sizeof (aggdesc)) != 0)
16090                         return (EFAULT);

16092                 mutex_enter(&dtrace_lock);

16094                 if ((agg = dtrace_aggid2agg(state, aggdesc.dtagd_id)) == NULL) {
16095                         mutex_exit(&dtrace_lock);
16096                         return (EINVAL);
16097                 }

16099                 aggdesc.dtagd_epid = agg->dtag_ecb->dte_epid;

16101                 nrecs = aggdesc.dtagd_nrecs;
16102                 aggdesc.dtagd_nrecs = 0;

16104                 offs = agg->dtag_base;
16105                 lrec = &agg->dtag_action.dta_rec;
16106                 aggdesc.dtagd_size = lrec->dtrd_offset + lrec->dtrd_size - offs;

16108                 for (act = agg->dtag_first; ; act = act->dta_next) {
16109                         ASSERT(act->dta_intuple ||
16110                             DTRACEACT_ISAGG(act->dta_kind));

16112                         /*
```

```
16113                          * If this action has a record size of zero, it
16114                          * denotes an argument to the aggregating action.
16115                          * Because the presence of this record doesn't (or
16116                          * shouldn't) affect the way the data is interpreted,
16117                          * we don't copy it out to save user-level the
16118                          * confusion of dealing with a zero-length record.
16119                          */
16120                         if (act->dta_rec.dtrd_size == 0) {
16121                                 ASSERT(agg->dtag_hasarg);
16122                                 continue;
16123                         }

16125                         aggdesc.dtagd_nrecs++;

16127                         if (act == &agg->dtag_action)
16128                                 break;
16129                 }

16131                 /*
16132                  * Now that we have the size, we need to allocate a temporary
16133                  * buffer in which to store the complete description.  We need
16134                  * the temporary buffer to be able to drop dtrace_lock()
16135                  * across the copyout(), below.
16136                  */
16137                 size = sizeof (dtrace_aggdesc_t) +
16138                     (aggdesc.dtagd_nrecs * sizeof (dtrace_recdesc_t));

16140                 buf = kmem_alloc(size, KM_SLEEP);
16141                 dest = (uintptr_t)buf;

16143                 bcopy(&aggdesc, (void *)dest, sizeof (aggdesc));
16144                 dest += offsetof(dtrace_aggdesc_t, dtagd_rec[0]);

16146                 for (act = agg->dtag_first; ; act = act->dta_next) {
16147                         dtrace_recdesc_t rec = act->dta_rec;

16149                         /*
16150                          * See the comment in the above loop for why we pass
16151                          * over zero-length records.
16152                          */
16153                         if (rec.dtrd_size == 0) {
16154                                 ASSERT(agg->dtag_hasarg);
16155                                 continue;
16156                         }

16158                         if (nrecs-- == 0)
16159                                 break;

16161                         rec.dtrd_offset -= offs;
16162                         bcopy(&rec, (void *)dest, sizeof (rec));
16163                         dest += sizeof (dtrace_recdesc_t);

16165                         if (act == &agg->dtag_action)
16166                                 break;
16167                 }

16169                 mutex_exit(&dtrace_lock);

16171                 if (copyout(buf, (void *)arg, dest - (uintptr_t)buf) != 0) {
16172                         kmem_free(buf, size);
16173                         return (EFAULT);
16174                 }

16176                 kmem_free(buf, size);
16177                 return (0);
16178         }
```



```
16180         case DTRACEIOC_ENABLE: {
16181                 dof_hdr_t *dof;
16182                 dtrace_enabling_t *enab = NULL;
16183                 dtrace_vstate_t *vstate;
16184                 int err = 0;

16186                 *rv = 0;

16188                 /*
16189                  * If a NULL argument has been passed, we take this as our
16190                  * cue to reevaluate our enablings.
16191                  */
16192                 if (arg == NULL) {
16193                         dtrace_enabling_matchall();

16195                         return (0);
16196                 }

16198                 if ((dof = dtrace_dof_copyin(arg, &rval)) == NULL)
16199                         return (rval);

16201                 mutex_enter(&cpu_lock);
16202                 mutex_enter(&dtrace_lock);
16203                 vstate = &state->dts_vstate;

16205                 if (state->dts_activity != DTRACE_ACTIVITY_INACTIVE) {
16206                         mutex_exit(&dtrace_lock);
16207                         mutex_exit(&cpu_lock);
16208                         dtrace_dof_destroy(dof);
16209                         return (EBUSY);
16210                 }

16212                 if (dtrace_dof_slurp(dof, vstate, cr, &enab, 0, B_TRUE) != 0) {
16213                         mutex_exit(&dtrace_lock);
16214                         mutex_exit(&cpu_lock);
16215                         dtrace_dof_destroy(dof);
16216                         return (EINVAL);
16217                 }

16219                 if ((rval = dtrace_dof_options(dof, state)) != 0) {
16220                         dtrace_enabling_destroy(enab);
16221                         mutex_exit(&dtrace_lock);
16222                         mutex_exit(&cpu_lock);
16223                         dtrace_dof_destroy(dof);
16224                         return (rval);
16225                 }

16227                 if ((err = dtrace_enabling_match(enab, rv)) == 0) {
16228                         err = dtrace_enabling_retain(enab);
16229                 } else {
16230                         dtrace_enabling_destroy(enab);
16231                 }

16233                 mutex_exit(&cpu_lock);
16234                 mutex_exit(&dtrace_lock);
16235                 dtrace_dof_destroy(dof);

16237                 return (err);
16238         }

16240         case DTRACEIOC_REPLICATE: {
16241                 dtrace_repldesc_t desc;
16242                 dtrace_probedesc_t *match = &desc.dtrpd_match;
16243                 dtrace_probedesc_t *create = &desc.dtrpd_create;
16244                 int err;
```

```
16246                 if (copyin((void *)arg, &desc, sizeof (desc)) != 0)
16247                         return (EFAULT);

16249                 match->dtpd_provider[DTRACE_PROVNAMELEN - 1] = '\0';
16250                 match->dtpd_mod[DTRACE_MODNAMELEN - 1] = '\0';
16251                 match->dtpd_func[DTRACE_FUNCNAMELEN - 1] = '\0';
16252                 match->dtpd_name[DTRACE_NAMELEN - 1] = '\0';

16254                 create->dtpd_provider[DTRACE_PROVNAMELEN - 1] = '\0';
16255                 create->dtpd_mod[DTRACE_MODNAMELEN - 1] = '\0';
16256                 create->dtpd_func[DTRACE_FUNCNAMELEN - 1] = '\0';
16257                 create->dtpd_name[DTRACE_NAMELEN - 1] = '\0';

16259                 mutex_enter(&dtrace_lock);
16260                 err = dtrace_enabling_replicate(state, match, create);
16261                 mutex_exit(&dtrace_lock);

16263                 return (err);
16264         }

16266         case DTRACEIOC_PROBEMATCH:
16267         case DTRACEIOC_PROBES: {
16268                 dtrace_probe_t *probe = NULL;
16269                 dtrace_probedesc_t desc;
16270                 dtrace_probekey_t pkey;
16271                 dtrace_id_t i;
16272                 int m = 0;
16273                 uint32_t priv;
16274                 uid_t uid;
16275                 zoneid_t zoneid;

16277                 if (copyin((void *)arg, &desc, sizeof (desc)) != 0)
16278                         return (EFAULT);

16280                 desc.dtpd_provider[DTRACE_PROVNAMELEN - 1] = '\0';
16281                 desc.dtpd_mod[DTRACE_MODNAMELEN - 1] = '\0';
16282                 desc.dtpd_func[DTRACE_FUNCNAMELEN - 1] = '\0';
16283                 desc.dtpd_name[DTRACE_NAMELEN - 1] = '\0';

16285                 /*
16286                  * Before we attempt to match this probe, we want to give
16287                  * all providers the opportunity to provide it.
16288                  */
16289                 if (desc.dtpd_id == DTRACE_IDNONE) {
16290                         mutex_enter(&dtrace_provider_lock);
16291                         dtrace_probe_provide(&desc, NULL);
16292                         mutex_exit(&dtrace_provider_lock);
16293                         desc.dtpd_id++;
16294                 }

16296                 if (cmd == DTRACEIOC_PROBEMATCH)  {
16297                         dtrace_probekey(&desc, &pkey);
16298                         pkey.dtpk_id = DTRACE_IDNONE;
16299                 }

16301                 dtrace_cred2priv(cr, &priv, &uid, &zoneid);

16303                 mutex_enter(&dtrace_lock);

16305                 if (cmd == DTRACEIOC_PROBEMATCH) {
16306                         for (i = desc.dtpd_id; i <= dtrace_nprobes; i++) {
16307                                 if ((probe = dtrace_probes[i - 1]) != NULL &&
16308                                     (m = dtrace_match_probe(probe, &pkey,
16309                                     priv, uid, zoneid)) != 0)
16310                                         break;
```



```
16311                         }

16313                         if (m < 0) {
16314                                 mutex_exit(&dtrace_lock);
16315                                 return (EINVAL);
16316                         }

16318                 } else {
16319                         for (i = desc.dtpd_id; i <= dtrace_nprobes; i++) {
16320                                 if ((probe = dtrace_probes[i - 1]) != NULL &&
16321                                     dtrace_match_priv(probe, priv, uid, zoneid))
16322                                         break;
16323                         }
16324                 }

16326                 if (probe == NULL) {
16327                         mutex_exit(&dtrace_lock);
16328                         return (ESRCH);
16329                 }

16331                 dtrace_probe_description(probe, &desc);
16332                 mutex_exit(&dtrace_lock);

16334                 if (copyout(&desc, (void *)arg, sizeof (desc)) != 0)
16335                         return (EFAULT);

16337                 return (0);
16338         }

16340         case DTRACEIOC_PROBEARG: {
16341                 dtrace_argdesc_t desc;
16342                 dtrace_probe_t *probe;
16343                 dtrace_provider_t *prov;

16345                 if (copyin((void *)arg, &desc, sizeof (desc)) != 0)
16346                         return (EFAULT);

16348                 if (desc.dtargd_id == DTRACE_IDNONE)
16349                         return (EINVAL);

16351                 if (desc.dtargd_ndx == DTRACE_ARGNONE)
16352                         return (EINVAL);

16354                 mutex_enter(&dtrace_provider_lock);
16355                 mutex_enter(&mod_lock);
16356                 mutex_enter(&dtrace_lock);

16358                 if (desc.dtargd_id > dtrace_nprobes) {
16359                         mutex_exit(&dtrace_lock);
16360                         mutex_exit(&mod_lock);
16361                         mutex_exit(&dtrace_provider_lock);
16362                         return (EINVAL);
16363                 }

16365                 if ((probe = dtrace_probes[desc.dtargd_id - 1]) == NULL) {
16366                         mutex_exit(&dtrace_lock);
16367                         mutex_exit(&mod_lock);
16368                         mutex_exit(&dtrace_provider_lock);
16369                         return (EINVAL);
16370                 }

16372                 mutex_exit(&dtrace_lock);

16374                 prov = probe->dtpr_provider;

16376                 if (prov->dtpv_pops.dtps_getargdesc == NULL) {
```

```
16377                         /*
16378                          * There isn't any typed information for this probe.
16379                          * Set the argument number to DTRACE_ARGNONE.
16380                          */
16381                         desc.dtargd_ndx = DTRACE_ARGNONE;
16382                 } else {
16383                         desc.dtargd_native[0] = '\0';
16384                         desc.dtargd_xlate[0] = '\0';
16385                         desc.dtargd_mapping = desc.dtargd_ndx;

16387                         prov->dtpv_pops.dtps_getargdesc(prov->dtpv_arg,
16388                             probe->dtpr_id, probe->dtpr_arg, &desc);
16389                 }

16391                 mutex_exit(&mod_lock);
16392                 mutex_exit(&dtrace_provider_lock);

16394                 if (copyout(&desc, (void *)arg, sizeof (desc)) != 0)
16395                         return (EFAULT);

16397                 return (0);
16398         }

16400         case DTRACEIOC_GO: {
16401                 processorid_t cpuid;
16402                 rval = dtrace_state_go(state, &cpuid);

16404                 if (rval != 0)
16405                         return (rval);

16407                 if (copyout(&cpuid, (void *)arg, sizeof (cpuid)) != 0)
16408                         return (EFAULT);

16410                 return (0);
16411         }

16413         case DTRACEIOC_STOP: {
16414                 processorid_t cpuid;

16416                 mutex_enter(&dtrace_lock);
16417                 rval = dtrace_state_stop(state, &cpuid);
16418                 mutex_exit(&dtrace_lock);

16420                 if (rval != 0)
16421                         return (rval);

16423                 if (copyout(&cpuid, (void *)arg, sizeof (cpuid)) != 0)
16424                         return (EFAULT);

16426                 return (0);
16427         }

16429         case DTRACEIOC_DOFGET: {
16430                 dof_hdr_t hdr, *dof;
16431                 uint64_t len;

16433                 if (copyin((void *)arg, &hdr, sizeof (hdr)) != 0)
16434                         return (EFAULT);

16436                 mutex_enter(&dtrace_lock);
16437                 dof = dtrace_dof_create(state);
16438                 mutex_exit(&dtrace_lock);

16440                 len = MIN(hdr.dofh_loadsz, dof->dofh_loadsz);
16441                 rval = copyout(dof, (void *)arg, len);
16442                 dtrace_dof_destroy(dof);
```



```
16444                 return (rval == 0 ? 0 : EFAULT);
16445         }

16447         case DTRACEIOC_AGGSNAP:
16448         case DTRACEIOC_BUFSNAP: {
16449                 dtrace_bufdesc_t desc;
16450                 caddr_t cached;
16451                 dtrace_buffer_t *buf;

16453                 if (copyin((void *)arg, &desc, sizeof (desc)) != 0)
16454                         return (EFAULT);

16456                 if (desc.dtbd_cpu < 0 || desc.dtbd_cpu >= NCPU)
16457                         return (EINVAL);

16459                 mutex_enter(&dtrace_lock);

16461                 if (cmd == DTRACEIOC_BUFSNAP) {
16462                         buf = &state->dts_buffer[desc.dtbd_cpu];
16463                 } else {
16464                         buf = &state->dts_aggbuffer[desc.dtbd_cpu];
16465                 }

16467                 if (buf->dtb_flags & (DTRACEBUF_RING | DTRACEBUF_FILL)) {
16468                         size_t sz = buf->dtb_offset;

16470                         if (state->dts_activity != DTRACE_ACTIVITY_STOPPED) {
16471                                 mutex_exit(&dtrace_lock);
16472                                 return (EBUSY);
16473                         }

16475                         /*
16476                          * If this buffer has already been consumed, we're
16477                          * going to indicate that there's nothing left here
16478                          * to consume.
16479                          */
16480                         if (buf->dtb_flags & DTRACEBUF_CONSUMED) {
16481                                 mutex_exit(&dtrace_lock);

16483                                 desc.dtbd_size = 0;
16484                                 desc.dtbd_drops = 0;
16485                                 desc.dtbd_errors = 0;
16486                                 desc.dtbd_oldest = 0;
16487                                 sz = sizeof (desc);

16489                                 if (copyout(&desc, (void *)arg, sz) != 0)
16490                                         return (EFAULT);

16492                                 return (0);
16493                         }

16495                         /*
16496                          * If this is a ring buffer that has wrapped, we want
16497                          * to copy the whole thing out.
16498                          */
16499                         if (buf->dtb_flags & DTRACEBUF_WRAPPED) {
16500                                 dtrace_buffer_polish(buf);
16501                                 sz = buf->dtb_size;
16502                         }

16504                         if (copyout(buf->dtb_tomax, desc.dtbd_data, sz) != 0) {
16505                                 mutex_exit(&dtrace_lock);
16506                                 return (EFAULT);
16507                         }
```

```
16509                         desc.dtbd_size = sz;
16510                         desc.dtbd_drops = buf->dtb_drops;
16511                         desc.dtbd_errors = buf->dtb_errors;
16512                         desc.dtbd_oldest = buf->dtb_xamot_offset;
16513                         desc.dtbd_timestamp = dtrace_gethrtime();

16515                         mutex_exit(&dtrace_lock);

16517                         if (copyout(&desc, (void *)arg, sizeof (desc)) != 0)
16518                                 return (EFAULT);

16520                         buf->dtb_flags |= DTRACEBUF_CONSUMED;

16522                         return (0);
16523                 }

16525                 if (buf->dtb_tomax == NULL) {
16526                         ASSERT(buf->dtb_xamot == NULL);
16527                         mutex_exit(&dtrace_lock);
16528                         return (ENOENT);
16529                 }

16531                 cached = buf->dtb_tomax;
16532                 ASSERT(!(buf->dtb_flags & DTRACEBUF_NOSWITCH));

16534                 dtrace_xcall(desc.dtbd_cpu,
16535                     (dtrace_xcall_t)dtrace_buffer_switch, buf);

16537                 state->dts_errors += buf->dtb_xamot_errors;

16539                 /*
16540                  * If the buffers did not actually switch, then the cross call
16541                  * did not take place -- presumably because the given CPU is
16542                  * not in the ready set.  If this is the case, we'll return
16543                  * ENOENT.
16544                  */
16545                 if (buf->dtb_tomax == cached) {
16546                         ASSERT(buf->dtb_xamot != cached);
16547                         mutex_exit(&dtrace_lock);
16548                         return (ENOENT);
16549                 }

16551                 ASSERT(cached == buf->dtb_xamot);

16553                 /*
16554                  * We have our snapshot; now copy it out.
16555                  */
16556                 if (copyout(buf->dtb_xamot, desc.dtbd_data,
16557                     buf->dtb_xamot_offset) != 0) {
16558                         mutex_exit(&dtrace_lock);
16559                         return (EFAULT);
16560                 }

16562                 desc.dtbd_size = buf->dtb_xamot_offset;
16563                 desc.dtbd_drops = buf->dtb_xamot_drops;
16564                 desc.dtbd_errors = buf->dtb_xamot_errors;
16565                 desc.dtbd_oldest = 0;
16566                 desc.dtbd_timestamp = buf->dtb_switched;

16568                 mutex_exit(&dtrace_lock);

16570                 /*
16571                  * Finally, copy out the buffer description.
16572                  */
16573                 if (copyout(&desc, (void *)arg, sizeof (desc)) != 0)
16574                         return (EFAULT);
```



```
16576                 return (0);
16577         }

16579         case DTRACEIOC_CONF: {
16580                 dtrace_conf_t conf;

16582                 bzero(&conf, sizeof (conf));
16583                 conf.dtc_difversion = DIF_VERSION;
16584                 conf.dtc_difintregs = DIF_DIR_NREGS;
16585                 conf.dtc_diftupregs = DIF_DTR_NREGS;
16586                 conf.dtc_ctfmodel = CTF_MODEL_NATIVE;

16588                 if (copyout(&conf, (void *)arg, sizeof (conf)) != 0)
16589                         return (EFAULT);

16591                 return (0);
16592         }

16594         case DTRACEIOC_STATUS: {
16595                 dtrace_status_t stat;
16596                 dtrace_dstate_t *dstate;
16597                 int i, j;
16598                 uint64_t nerrs;

16600                 /*
16601                  * See the comment in dtrace_state_deadman() for the reason
16602                  * for setting dts_laststatus to INT64_MAX before setting
16603                  * it to the correct value.
16604                  */
16605                 state->dts_laststatus = INT64_MAX;
16606                 dtrace_membar_producer();
16607                 state->dts_laststatus = dtrace_gethrtime();

16609                 bzero(&stat, sizeof (stat));

16611                 mutex_enter(&dtrace_lock);

16613                 if (state->dts_activity == DTRACE_ACTIVITY_INACTIVE) {
16614                         mutex_exit(&dtrace_lock);
16615                         return (ENOENT);
16616                 }

16618                 if (state->dts_activity == DTRACE_ACTIVITY_DRAINING)
16619                         stat.dtst_exiting = 1;

16621                 nerrs = state->dts_errors;
16622                 dstate = &state->dts_vstate.dtvs_dynvars;

16624                 for (i = 0; i < NCPU; i++) {
16625                         dtrace_dstate_percpu_t *dcpu = &dstate->dtds_percpu[i];

16627                         stat.dtst_dyndrops += dcpu->dtdsc_drops;
16628                         stat.dtst_dyndrops_dirty += dcpu->dtdsc_dirty_drops;
16629                         stat.dtst_dyndrops_rinsing += dcpu->dtdsc_rinsing_drops;

16631                         if (state->dts_buffer[i].dtb_flags & DTRACEBUF_FULL)
16632                                 stat.dtst_filled++;

16634                         nerrs += state->dts_buffer[i].dtb_errors;

16636                         for (j = 0; j < state->dts_nspeculations; j++) {
16637                                 dtrace_speculation_t *spec;
16638                                 dtrace_buffer_t *buf;

16640                                 spec = &state->dts_speculations[j];
```

```
16641                                   buf = &spec->dtsp_buffer[i];
16642                                   stat.dtst_specdrops += buf->dtb_xamot_drops;
16643                           }
16644                   }

16646                   stat.dtst_specdrops_busy = state->dts_speculations_busy;
16647                   stat.dtst_specdrops_unavail = state->dts_speculations_unavail;
16648                   stat.dtst_stkstroverflows = state->dts_stkstroverflows;
16649                   stat.dtst_dblerrors = state->dts_dblerrors;
16650                   stat.dtst_killed =
16651                       (state->dts_activity == DTRACE_ACTIVITY_KILLED);
16652                   stat.dtst_errors = nerrs;

16654                   mutex_exit(&dtrace_lock);

16656                   if (copyout(&stat, (void *)arg, sizeof (stat)) != 0)
16657                           return (EFAULT);

16659                   return (0);
16660           }

16662           case DTRACEIOC_FORMAT: {
16663                   dtrace_fmtdesc_t fmt;
16664                   char *str;
16665                   int len;

16667                   if (copyin((void *)arg, &fmt, sizeof (fmt)) != 0)
16668                           return (EFAULT);

16670                   mutex_enter(&dtrace_lock);

16672                   if (fmt.dtfd_format == 0 ||
16673                       fmt.dtfd_format > state->dts_nformats) {
16674                           mutex_exit(&dtrace_lock);
16675                           return (EINVAL);
16676                   }

16678                   /*
16679                    * Format strings are allocated contiguously and they are
16680                    * never freed; if a format index is less than the number
16681                    * of formats, we can assert that the format map is non-NULL
16682                    * and that the format for the specified index is non-NULL.
16683                    */
16684                   ASSERT(state->dts_formats != NULL);
16685                   str = state->dts_formats[fmt.dtfd_format - 1];
16686                   ASSERT(str != NULL);

16688                   len = strlen(str) + 1;

16690                   if (len > fmt.dtfd_length) {
16691                           fmt.dtfd_length = len;

16693                           if (copyout(&fmt, (void *)arg, sizeof (fmt)) != 0) {
16694                                   mutex_exit(&dtrace_lock);
16695                                   return (EINVAL);
16696                           }
16697                   } else {
16698                           if (copyout(str, fmt.dtfd_string, len) != 0) {
16699                                   mutex_exit(&dtrace_lock);
16700                                   return (EINVAL);
16701                           }
16702                   }

16704                   mutex_exit(&dtrace_lock);
16705                   return (0);
16706           }
```



```
16708           default:
16709                   break;
16710           }

16712           return (ENOTTY);
16713   }

16715   /*ARGSUSED*/
16716   static int
16717   dtrace_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
16718   {
16719           dtrace_state_t *state;

16721           switch (cmd) {
16722           case DDI_DETACH:
16723                   break;

16725           case DDI_SUSPEND:
16726                   return (DDI_SUCCESS);

16728           default:
16729                   return (DDI_FAILURE);
16730           }

16732           mutex_enter(&cpu_lock);
16733           mutex_enter(&dtrace_provider_lock);
16734           mutex_enter(&dtrace_lock);

16736           ASSERT(dtrace_opens == 0);

16738           if (dtrace_helpers > 0) {
16739                   mutex_exit(&dtrace_provider_lock);
16740                   mutex_exit(&dtrace_lock);
16741                   mutex_exit(&cpu_lock);
16742                   return (DDI_FAILURE);
16743           }

16745           if (dtrace_unregister((dtrace_provider_id_t)dtrace_provider) != 0) {
16746                   mutex_exit(&dtrace_provider_lock);
16747                   mutex_exit(&dtrace_lock);
16748                   mutex_exit(&cpu_lock);
16749                   return (DDI_FAILURE);
16750           }

16752           dtrace_provider = NULL;

16754           if ((state = dtrace_anon_grab()) != NULL) {
16755                   /*
16756                    * If there were ECBs on this state, the provider should
16757                    * have not been allowed to detach; assert that there is
16758                    * none.
16759                    */
16760                   ASSERT(state->dts_necbs == 0);
16761                   dtrace_state_destroy(state);

16763                   /*
16764                    * If we're being detached with anonymous state, we need to
16765                    * indicate to the kernel debugger that DTrace is now inactive.
16766                    */
16767                   (void) kdi_dtrace_set(KDI_DTSET_DTRACE_DEACTIVATE);
16768           }

16770           bzero(&dtrace_anon, sizeof (dtrace_anon_t));
16771           unregister_cpu_setup_func((cpu_setup_func_t *)dtrace_cpu_setup, NULL);
16772           dtrace_cpu_init = NULL;
```

```
16773         dtrace_helpers_cleanup = NULL;
16774         dtrace_helpers_fork = NULL;
16775         dtrace_cpustart_init = NULL;
16776         dtrace_cpustart_fini = NULL;
16777         dtrace_debugger_init = NULL;
16778         dtrace_debugger_fini = NULL;
16779         dtrace_modload = NULL;
16780         dtrace_modunload = NULL;

16782         ASSERT(dtrace_getf == 0);
16783         ASSERT(dtrace_closef == NULL);

16785         mutex_exit(&cpu_lock);

15814         if (dtrace_helptrace_enabled) {
15815                 kmem_free(dtrace_helptrace_buffer, dtrace_helptrace_bufsize);
15816                 dtrace_helptrace_buffer = NULL;
15817         }

16787         kmem_free(dtrace_probes, dtrace_nprobes * sizeof (dtrace_probe_t *));
16788         dtrace_probes = NULL;
16789         dtrace_nprobes = 0;

16791         dtrace_hash_destroy(dtrace_bymod);
16792         dtrace_hash_destroy(dtrace_byfunc);
16793         dtrace_hash_destroy(dtrace_byname);
16794         dtrace_bymod = NULL;
16795         dtrace_byfunc = NULL;
16796         dtrace_byname = NULL;

16798         kmem_cache_destroy(dtrace_state_cache);
16799         vmem_destroy(dtrace_minor);
16800         vmem_destroy(dtrace_arena);

16802         if (dtrace_toxrange != NULL) {
16803                 kmem_free(dtrace_toxrange,
16804                     dtrace_toxranges_max * sizeof (dtrace_toxrange_t));
16805                 dtrace_toxrange = NULL;
16806                 dtrace_toxranges = 0;
16807                 dtrace_toxranges_max = 0;
16808         }

16810         ddi_remove_minor_node(dtrace_devi, NULL);
16811         dtrace_devi = NULL;

16813         ddi_soft_state_fini(&dtrace_softstate);

16815         ASSERT(dtrace_vtime_references == 0);
16816         ASSERT(dtrace_opens == 0);
16817         ASSERT(dtrace_retained == NULL);

16819         mutex_exit(&dtrace_lock);
16820         mutex_exit(&dtrace_provider_lock);

16822         /*
16823          * We don't destroy the task queue until after we have dropped our
16824          * locks (taskq_destroy() may block on running tasks).  To prevent
16825          * attempting to do work after we have effectively detached but before
16826          * the task queue has been destroyed, all tasks dispatched via the
16827          * task queue must check that DTrace is still attached before
16828          * performing any operation.
16829          */
16830         taskq_destroy(dtrace_taskq);
16831         dtrace_taskq = NULL;

16833         return (DDI_SUCCESS);
```



```
16834 }
```

*unchanged_portion_omitted_*