```
**********************************************************
    3027 Sat Jun 23 09:31:25 2012
new/usr/src/cmd/dtrace/test/tst/common/privs/tst.providers.ksh
2917 DTrace in a zone should have limited provider access
**********************************************************
```

```
#
# 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) 2012, Joyent, Inc. All rights reserved.
#

#
# First, make sure that we can successfully enable the io provider
#
if ! dtrace -P io -n BEGIN'{exit(0)}' > /dev/null 2>&1 ; then
        echo failed to enable io provider with full privs
        exit 1
fi

ppriv -s A=basic,dtrace_proc,dtrace_user $$

#
# Now make sure that we cannot enable the io provider with reduced privs
#
if ! dtrace -x errtags -P io -n BEGIN'{exit(1)}' 2>&1 | \
    grep D_PDESC_ZERO > /dev/null 2>&1 ; then
        echo successfully enabled the io provider with reduced privs
        exit 1
fi

#
# Keeping our reduced privs, we want to assure that we can see every provider
# that we think we should be able to see -- and that we can see curpsinfo
# state but can't otherwise see arguments.
#
/usr/sbin/dtrace -wq -Cs /dev/stdin <<EOF

int seen[string];
int err;

#define CANENABLE(provider) \
provider::: \
/err == 0 && progenyof(\$pid) && !seen["provider"]/ \
{ \
        trace(arg0); \
        printf("\nsuccessful trace of arg0 in %s:%s:%s:%s\n", \
            probeprov, probemod, probefunc, probename); \
        exit(++err); \
} \
 \
provider::: \
/progenyof(\$pid)/ \
{ \
        seen["provider"]++; \
} \
 \
provider::: \
/progenyof(\$pid)/ \
{ \
        errstr = "provider"; \
        this->ignore = stringof(curpsinfo->pr_psargs); \
        errstr = ""; \
} \
 \
END \
/err == 0 && !seen["provider"]/ \
{ \
        printf("no probes from provider\n"); \
        exit(++err); \
} \
 \
END \
/err == 0/ \
{ \
        printf("saw %d probes from provider\n", seen["provider"]); \
}

CANENABLE(proc)
CANENABLE(sched)
CANENABLE(vminfo)
CANENABLE(sysinfo)

BEGIN
{
        /*
         * We'll kick off a system of a do-nothing command -- which should be
         * enough to kick proc, sched, vminfo and sysinfo probes.
         */
        system("echo > /dev/null");
}

ERROR
/err == 0 && errstr != ""/
{
        printf("fatal error: couldn't read curpsinfo->pr_psargs in ");
        printf("%s-provided probe\n", errstr);
        exit(++err);
}

proc:::exit
/progenyof(\$pid)/
{
        exit(0);
}

tick-10ms
/i++ > 500/
{
        printf("exit probe did not seem to fire\n");
        exit(++err);
}
EOF
```

```
**********************************************************
  419259 Sat Jun 23 09:31:26 2012
new/usr/src/uts/common/dtrace/dtrace.c
2917 DTrace in a zone should have limited provider access
**********************************************************
______unchanged_portion_omitted_

1282 /*
1283  * Determine if the dte_cond of the specified ECB allows for processing of
1284  * the current probe to continue.  Note that this routine may allow continued
1285  * processing, but with access(es) stripped from the mstate's dtms_access
1286  * field.
1287  */
1288 static int
1289 dtrace_priv_probe(dtrace_state_t *state, dtrace_mstate_t *mstate,
1290     dtrace_ecb_t *ecb)
1291 {
1292         dtrace_probe_t *probe = ecb->dte_probe;
1293         dtrace_provider_t *prov = probe->dtpr_provider;
1294         dtrace_pops_t *pops = &prov->dtpv_pops;
1295         int mode = DTRACE_MODE_NOPRIV_DROP;

1297         ASSERT(ecb->dte_cond);

1299         if (pops->dtps_mode != NULL) {
1300                 mode = pops->dtps_mode(prov->dtpv_arg,
1301                     probe->dtpr_id, probe->dtpr_arg);

1303                 ASSERT(mode & (DTRACE_MODE_USER | DTRACE_MODE_KERNEL));
1304                 ASSERT(mode & (DTRACE_MODE_NOPRIV_RESTRICT |
1305                     DTRACE_MODE_NOPRIV_DROP));
1303                 ASSERT((mode & DTRACE_MODE_USER) ||
1304                     (mode & DTRACE_MODE_KERNEL));
1305                 ASSERT((mode & DTRACE_MODE_NOPRIV_RESTRICT) ||
1306                     (mode & DTRACE_MODE_NOPRIV_DROP));
1306         }

1308         /*
1309          * If the dte_cond bits indicate that this consumer is only allowed to
1310          * see user-mode firings of this probe, check that the probe was fired
1311          * while in a user context.  If that's not the case, use the policy
1312          * specified by the provider to determine if we drop the probe or
1313          * merely restrict operation.
1311          * see user-mode firings of this probe, call the provider's dtps_mode()
1312          * entry point to check that the probe was fired while in a user
1313          * context.  If that's not the case, use the policy specified by the
1314          * provider to determine if we drop the probe or merely restrict
1315          * operation.
1314          */
1315         if (ecb->dte_cond & DTRACE_COND_USERMODE) {
1316                 ASSERT(mode != DTRACE_MODE_NOPRIV_DROP);

1318                 if (!(mode & DTRACE_MODE_USER)) {
1319                         if (mode & DTRACE_MODE_NOPRIV_DROP)
1320                                 return (0);

1322                         mstate->dtms_access &= ~DTRACE_ACCESS_ARGS;
1323                 }
1324         }

1326         /*
1327          * This is more subtle than it looks. We have to be absolutely certain
1328          * that CRED() isn't going to change out from under us so it's only
1329          * legit to examine that structure if we're in constrained situations.
1330          * Currently, the only times we'll this check is if a non-super-user
1331          * has enabled the profile or syscall providers -- providers that
```



```
1332          * allow visibility of all processes. For the profile case, the check
1333          * above will ensure that we're examining a user context.
1334          */
1335         if (ecb->dte_cond & DTRACE_COND_OWNER) {
1336                 cred_t *cr;
1337                 cred_t *s_cr = state->dts_cred.dcr_cred;
1338                 proc_t *proc;

1340                 ASSERT(s_cr != NULL);

1342                 if ((cr = CRED()) == NULL ||
1343                     s_cr->cr_uid != cr->cr_uid ||
1344                     s_cr->cr_uid != cr->cr_ruid ||
1345                     s_cr->cr_uid != cr->cr_suid ||
1346                     s_cr->cr_gid != cr->cr_gid ||
1347                     s_cr->cr_gid != cr->cr_rgid ||
1348                     s_cr->cr_gid != cr->cr_sgid ||
1349                     (proc = ttoproc(curthread)) == NULL ||
1350                     (proc->p_flag & SNOCD)) {
1351                         if (mode & DTRACE_MODE_NOPRIV_DROP)
1352                                 return (0);

1354                         mstate->dtms_access &= ~DTRACE_ACCESS_PROC;
1355                 }
1356         }

1358         /*
1359          * If our dte_cond is set to DTRACE_COND_ZONEOWNER and we are not
1360          * in our zone, check to see if our mode policy is to restrict rather
1361          * than to drop; if to restrict, strip away both DTRACE_ACCESS_PROC
1362          * and DTRACE_ACCESS_ARGS
1363          */
1364         if (ecb->dte_cond & DTRACE_COND_ZONEOWNER) {
1365                 cred_t *cr;
1366                 cred_t *s_cr = state->dts_cred.dcr_cred;

1368                 ASSERT(s_cr != NULL);

1370                 if ((cr = CRED()) == NULL ||
1371                     s_cr->cr_zone->zone_id != cr->cr_zone->zone_id) {
1372                         if (mode & DTRACE_MODE_NOPRIV_DROP)
1373                                 return (0);

1375                         mstate->dtms_access &=
1376                             ~(DTRACE_ACCESS_PROC | DTRACE_ACCESS_ARGS);
1377                 }
1378         }

1380         /*
1381          * By merits of being in this code path at all, we have limited
1382          * privileges.  If the provider has indicated that limited privileges
1383          * are to denote restricted operation, strip off the ability to access
1384          * arguments.
1385          */
1386         if (mode & DTRACE_MODE_LIMITEDPRIV_RESTRICT)
1387                 mstate->dtms_access &= ~DTRACE_ACCESS_ARGS;

1389         return (1);
1390 }
______unchanged_portion_omitted_
```

```
**********************************************************
   55331 Sat Jun 23 09:31:27 2012
new/usr/src/uts/common/dtrace/sdt_subr.c
2917 DTrace in a zone should have limited provider access
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright (c) 2004, 2010, Oracle and/or its affiliates. All rights reserved.
  23  * Copyright (c) 2012, Joyent, Inc. All rights reserved.
  24  */

  26 #include <sys/sdt_impl.h>

  28 static dtrace_pattr_t vtrace_attr = {
  29 { DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_ISA },
  30 { DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
  31 { DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
  32 { DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
  33 { DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_ISA },
  34 };
```

*unchanged_portion_omitted_*

```
 100 sdt_provider_t sdt_providers[] = {
 101         { "vtrace", "__vtrace_", &vtrace_attr },
 102         { "sysinfo", "__cpu_sysinfo_", &info_attr, DTRACE_PRIV_USER },
 103         { "vminfo", "__cpu_vminfo_", &info_attr, DTRACE_PRIV_USER },
 104         { "fpuinfo", "__fpuinfo_", &fpu_attr },
 105         { "sched", "__sched_", &stab_attr, DTRACE_PRIV_USER },
 106         { "proc", "__proc_", &stab_attr, DTRACE_PRIV_USER },
 107         { "io", "__io_", &stab_attr },
 108         { "ip", "__ip_", &stab_attr },
 109         { "tcp", "__tcp_", &stab_attr },
 110         { "udp", "__udp_", &stab_attr },
 111         { "mib", "__mib_", &stab_attr },
 112         { "fsinfo", "__fsinfo_", &fsinfo_attr },
 113         { "iscsi", "__iscsi_", &iscsi_attr },
 114         { "nfsv3", "__nfsv3_", &stab_attr },
 115         { "nfsv4", "__nfsv4_", &stab_attr },
 116         { "xpv", "__xpv_", &xpv_attr },
 117         { "fc", "__fc_", &fc_attr },
 118         { "srp", "__srp_", &fc_attr },
 119         { "sysevent", "__sysevent_", &stab_attr },
 120         { "sdt", NULL, &sdt_attr },
 100         { "vtrace", "__vtrace_", &vtrace_attr, 0 },
 101         { "sysinfo", "__cpu_sysinfo_", &info_attr, 0 },
 102         { "vminfo", "__cpu_vminfo_", &info_attr, 0 },
 103         { "fpuinfo", "__fpuinfo_", &fpu_attr, 0 },
```



```
 104         { "sched", "__sched_", &stab_attr, 0 },
 105         { "proc", "__proc_", &stab_attr, 0 },
 106         { "io", "__io_", &stab_attr, 0 },
 107         { "ip", "__ip_", &stab_attr, 0 },
 108         { "tcp", "__tcp_", &stab_attr, 0 },
 109         { "udp", "__udp_", &stab_attr, 0 },
 110         { "mib", "__mib_", &stab_attr, 0 },
 111         { "fsinfo", "__fsinfo_", &fsinfo_attr, 0 },
 112         { "iscsi", "__iscsi_", &iscsi_attr, 0 },
 113         { "nfsv3", "__nfsv3_", &stab_attr, 0 },
 114         { "nfsv4", "__nfsv4_", &stab_attr, 0 },
 115         { "xpv", "__xpv_", &xpv_attr, 0 },
 116         { "fc", "__fc_", &fc_attr, 0 },
 117         { "srp", "__srp_", &fc_attr, 0 },
 118         { "sysevent", "__sysevent_", &stab_attr, 0 },
 119         { "sdt", NULL, &sdt_attr, 0 },
 121         { NULL }
 122 };
```

*unchanged_portion_omitted_*

```
1158 /*ARGSUSED*/
1159 int
1160 sdt_mode(void *arg, dtrace_id_t id, void *parg)
1161 {
1162         /*
1163          * We tell DTrace that we're in kernel mode, that the firing needs to
1164          * be dropped for anything that doesn't have necessary privileges, and
1165          * that it needs to be restricted for anything that has restricted
1166          * (i.e., not all-zone) privileges.
1167          */
1168         return (DTRACE_MODE_KERNEL | DTRACE_MODE_NOPRIV_DROP |
1169             DTRACE_MODE_LIMITEDPRIV_RESTRICT);
1170 }

1172 /*ARGSUSED*/
1173 void
1174 sdt_getargdesc(void *arg, dtrace_id_t id, void *parg, dtrace_argdesc_t *desc)
1175 {
1176         sdt_probe_t *sdp = parg;
1177         int i;

1179         desc->dtargd_native[0] = '\0';
1180         desc->dtargd_xlate[0] = '\0';

1182         for (i = 0; sdt_args[i].sda_provider != NULL; i++) {
1183                 sdt_argdesc_t *a = &sdt_args[i];

1185                 if (strcmp(sdp->sdp_provider->sdtp_name, a->sda_provider) != 0)
1186                         continue;

1188                 if (a->sda_name != NULL &&
1189                     strcmp(sdp->sdp_name, a->sda_name) != 0)
1190                         continue;

1192                 if (desc->dtargd_ndx != a->sda_ndx)
1193                         continue;

1195                 if (a->sda_native != NULL)
1196                         (void) strcpy(desc->dtargd_native, a->sda_native);

1198                 if (a->sda_xlate != NULL)
1199                         (void) strcpy(desc->dtargd_xlate, a->sda_xlate);

1201                 desc->dtargd_mapping = a->sda_mapping;
1202                 return;
1203         }
```

```
1205         desc->dtargd_ndx = DTRACE_ARGNONE;
1206 }
______unchanged_portion_omitted_
```

```
**********************************************************
  101025 Sat Jun 23 09:31:28 2012
new/usr/src/uts/common/sys/dtrace.h
2917 DTrace in a zone should have limited provider access
**********************************************************
```

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

/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 * Copyright (c) 2011, Joyent, Inc. All rights reserved.
 */

#ifndef _SYS_DTRACE_H
#define _SYS_DTRACE_H

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * DTrace Dynamic Tracing Software: Kernel Interfaces
 *
 * Note: The contents of this file are private to the implementation of the
 * Solaris system and DTrace subsystem and are subject to change at any time
 * without notice.  Applications and drivers using these interfaces will fail
 * to run on future releases.  These interfaces should not be used for any
 * purpose except those expressly outlined in dtrace(7D) and libdtrace(3LIB).
 * Please refer to the "Solaris Dynamic Tracing Guide" for more information.
 */

#ifndef _ASM

#include <sys/types.h>
#include <sys/modctl.h>
#include <sys/processor.h>
#include <sys/systm.h>
#include <sys/ctf_api.h>
#include <sys/cyclic.h>
#include <sys/int_limits.h>

/*
 * DTrace Universal Constants and Typedefs
```



```
 */
#define DTRACE_CPUALL           -1      /* all CPUs */
#define DTRACE_IDNONE           0       /* invalid probe identifier */
#define DTRACE_EPIDNONE         0       /* invalid enabled probe identifier */
#define DTRACE_AGGIDNONE        0       /* invalid aggregation identifier */
#define DTRACE_AGGVARIDNONE     0       /* invalid aggregation variable ID */
#define DTRACE_CACHEIDNONE      0       /* invalid predicate cache */
#define DTRACE_PROVNONE         0       /* invalid provider identifier */
#define DTRACE_METAPROVNONE     0       /* invalid meta-provider identifier */
#define DTRACE_ARGNONE          -1      /* invalid argument index */

#define DTRACE_PROVNAMELEN      64
#define DTRACE_MODNAMELEN       64
#define DTRACE_FUNCNAMELEN      128
#define DTRACE_NAMELEN          64
#define DTRACE_FULLNAMELEN      (DTRACE_PROVNAMELEN + DTRACE_MODNAMELEN + \
                                 DTRACE_FUNCNAMELEN + DTRACE_NAMELEN + 4)
#define DTRACE_ARGTYPELEN       128

typedef uint32_t dtrace_id_t;           /* probe identifier */
typedef uint32_t dtrace_epid_t;         /* enabled probe identifier */
typedef uint32_t dtrace_aggid_t;        /* aggregation identifier */
typedef int64_t dtrace_aggvarid_t;      /* aggregation variable identifier */
typedef uint16_t dtrace_actkind_t;      /* action kind */
typedef int64_t dtrace_optval_t;        /* option value */
typedef uint32_t dtrace_cacheid_t;      /* predicate cache identifier */

typedef enum dtrace_probespec {
        DTRACE_PROBESPEC_NONE = -1,
        DTRACE_PROBESPEC_PROVIDER = 0,
        DTRACE_PROBESPEC_MOD,
        DTRACE_PROBESPEC_FUNC,
        DTRACE_PROBESPEC_NAME
} dtrace_probespec_t;
_____unchanged_portion_omitted_

#define DTRACEMNR_DTRACE        "dtrace"        /* node for DTrace ops */
#define DTRACEMNR_HELPER        "helper"        /* node for helpers */
#define DTRACEMNRN_DTRACE       0               /* minor for DTrace ops */
#define DTRACEMNRN_HELPER       1               /* minor for helpers */
#define DTRACEMNRN_CLONE        2               /* first clone minor */

#ifdef _KERNEL

/*
 * DTrace Provider API
 *
 * The following functions are implemented by the DTrace framework and are
 * used to implement separate in-kernel DTrace providers.  Common functions
 * are provided in uts/common/os/dtrace.c.  ISA-dependent subroutines are
 * defined in uts/<isa>/dtrace/dtrace_asm.s or uts/<isa>/dtrace/dtrace_isa.c.
 *
 * The provider API has two halves:  the API that the providers consume from
 * DTrace, and the API that providers make available to DTrace.
 *
 * 1 Framework-to-Provider API
 *
 * 1.1  Overview
 *
 * The Framework-to-Provider API is represented by the dtrace_pops structure
 * that the provider passes to the framework when registering itself.  This
 * structure consists of the following members:
 *
 *   dtps_provide()          <-- Provide all probes, all modules
 *   dtps_provide_module()   <-- Provide all probes in specified module
 *   dtps_enable()           <-- Enable specified probe
```

```
1355  *   dtps_disable()          <-- Disable specified probe
1356  *   dtps_suspend()          <-- Suspend specified probe
1357  *   dtps_resume()           <-- Resume specified probe
1358  *   dtps_getargdesc()       <-- Get the argument description for args[X]
1359  *   dtps_getargval()        <-- Get the value for an argX or args[X] variable
1360  *   dtps_mode()             <-- Return the mode of the fired probe
1361  *   dtps_destroy()          <-- Destroy all state associated with this probe
1362  *
1363  * 1.2  void dtps_provide(void *arg, const dtrace_probedesc_t *spec)
1364  *
1365  * 1.2.1  Overview
1366  *
1367  *   Called to indicate that the provider should provide all probes.  If the
1368  *   specified description is non-NULL, dtps_provide() is being called because
1369  *   no probe matched a specified probe -- if the provider has the ability to
1370  *   create custom probes, it may wish to create a probe that matches the
1371  *   specified description.
1372  *
1373  * 1.2.2  Arguments and notes
1374  *
1375  *   The first argument is the cookie as passed to dtrace_register().  The
1376  *   second argument is a pointer to a probe description that the provider may
1377  *   wish to consider when creating custom probes.  The provider is expected to
1378  *   call back into the DTrace framework via dtrace_probe_create() to create
1379  *   any necessary probes.  dtps_provide() may be called even if the provider
1380  *   has made available all probes; the provider should check the return value
1381  *   of dtrace_probe_create() to handle this case.  Note that the provider need
1382  *   not implement both dtps_provide() and dtps_provide_module(); see
1383  *   "Arguments and Notes" for dtrace_register(), below.
1384  *
1385  * 1.2.3  Return value
1386  *
1387  *   None.
1388  *
1389  * 1.2.4  Caller's context
1390  *
1391  *   dtps_provide() is typically called from open() or ioctl() context, but may
1392  *   be called from other contexts as well.  The DTrace framework is locked in
1393  *   such a way that providers may not register or unregister.  This means that
1394  *   the provider may not call any DTrace API that affects its registration with
1395  *   the framework, including dtrace_register(), dtrace_unregister(),
1396  *   dtrace_invalidate(), and dtrace_condense().  However, the context is such
1397  *   that the provider may (and indeed, is expected to) call probe-related
1398  *   DTrace routines, including dtrace_probe_create(), dtrace_probe_lookup(),
1399  *   and dtrace_probe_arg().
1400  *
1401  * 1.3  void dtps_provide_module(void *arg, struct modctl *mp)
1402  *
1403  * 1.3.1  Overview
1404  *
1405  *   Called to indicate that the provider should provide all probes in the
1406  *   specified module.
1407  *
1408  * 1.3.2  Arguments and notes
1409  *
1410  *   The first argument is the cookie as passed to dtrace_register().  The
1411  *   second argument is a pointer to a modctl structure that indicates the
1412  *   module for which probes should be created.
1413  *
1414  * 1.3.3  Return value
1415  *
1416  *   None.
1417  *
1418  * 1.3.4  Caller's context
1419  *
1420  *   dtps_provide_module() may be called from open() or ioctl() context, but
```



```
1421  *   may also be called from a module loading context.  mod_lock is held, and
1422  *   the DTrace framework is locked in such a way that providers may not
1423  *   register or unregister.  This means that the provider may not call any
1424  *   DTrace API that affects its registration with the framework, including
1425  *   dtrace_register(), dtrace_unregister(), dtrace_invalidate(), and
1426  *   dtrace_condense().  However, the context is such that the provider may (and
1427  *   indeed, is expected to) call probe-related DTrace routines, including
1428  *   dtrace_probe_create(), dtrace_probe_lookup(), and dtrace_probe_arg().  Note
1429  *   that the provider need not implement both dtps_provide() and
1430  *   dtps_provide_module(); see "Arguments and Notes" for dtrace_register(),
1431  *   below.
1432  *
1433  * 1.4  int dtps_enable(void *arg, dtrace_id_t id, void *parg)
1434  *
1435  * 1.4.1  Overview
1436  *
1437  *   Called to enable the specified probe.
1438  *
1439  * 1.4.2  Arguments and notes
1440  *
1441  *   The first argument is the cookie as passed to dtrace_register().  The
1442  *   second argument is the identifier of the probe to be enabled.  The third
1443  *   argument is the probe argument as passed to dtrace_probe_create().
1444  *   dtps_enable() will be called when a probe transitions from not being
1445  *   enabled at all to having one or more ECB.  The number of ECBs associated
1446  *   with the probe may change without subsequent calls into the provider.
1447  *   When the number of ECBs drops to zero, the provider will be explicitly
1448  *   told to disable the probe via dtps_disable().  dtrace_probe() should never
1449  *   be called for a probe identifier that hasn't been explicitly enabled via
1450  *   dtps_enable().
1451  *
1452  * 1.4.3  Return value
1453  *
1454  *   On success, dtps_enable() should return 0. On failure, -1 should be
1455  *   returned.
1456  *
1457  * 1.4.4  Caller's context
1458  *
1459  *   The DTrace framework is locked in such a way that it may not be called
1460  *   back into at all.  cpu_lock is held.  mod_lock is not held and may not
1461  *   be acquired.
1462  *
1463  * 1.5  void dtps_disable(void *arg, dtrace_id_t id, void *parg)
1464  *
1465  * 1.5.1  Overview
1466  *
1467  *   Called to disable the specified probe.
1468  *
1469  * 1.5.2  Arguments and notes
1470  *
1471  *   The first argument is the cookie as passed to dtrace_register().  The
1472  *   second argument is the identifier of the probe to be disabled.  The third
1473  *   argument is the probe argument as passed to dtrace_probe_create().
1474  *   dtps_disable() will be called when a probe transitions from being enabled
1475  *   to having zero ECBs.  dtrace_probe() should never be called for a probe
1476  *   identifier that has been explicitly enabled via dtps_disable().
1477  *
1478  * 1.5.3  Return value
1479  *
1480  *   None.
1481  *
1482  * 1.5.4  Caller's context
1483  *
1484  *   The DTrace framework is locked in such a way that it may not be called
1485  *   back into at all.  cpu_lock is held.  mod_lock is not held and may not
1486  *   be acquired.
```

```
 *
 * 1.6  void dtps_suspend(void *arg, dtrace_id_t id, void *parg)
 *
 * 1.6.1  Overview
 *
 *   Called to suspend the specified enabled probe.  This entry point is for
 *   providers that may need to suspend some or all of their probes when CPUs
 *   are being powered on or when the boot monitor is being entered for a
 *   prolonged period of time.
 *
 * 1.6.2  Arguments and notes
 *
 *   The first argument is the cookie as passed to dtrace_register().  The
 *   second argument is the identifier of the probe to be suspended.  The
 *   third argument is the probe argument as passed to dtrace_probe_create().
 *   dtps_suspend will only be called on an enabled probe.  Providers that
 *   provide a dtps_suspend entry point will want to take roughly the action
 *   that it takes for dtps_disable.
 *
 * 1.6.3  Return value
 *
 *   None.
 *
 * 1.6.4  Caller's context
 *
 *   Interrupts are disabled.  The DTrace framework is in a state such that the
 *   specified probe cannot be disabled or destroyed for the duration of
 *   dtps_suspend().  As interrupts are disabled, the provider is afforded
 *   little latitude; the provider is expected to do no more than a store to
 *   memory.
 *
 * 1.7  void dtps_resume(void *arg, dtrace_id_t id, void *parg)
 *
 * 1.7.1  Overview
 *
 *   Called to resume the specified enabled probe.  This entry point is for
 *   providers that may need to resume some or all of their probes after the
 *   completion of an event that induced a call to dtps_suspend().
 *
 * 1.7.2  Arguments and notes
 *
 *   The first argument is the cookie as passed to dtrace_register().  The
 *   second argument is the identifier of the probe to be resumed.  The
 *   third argument is the probe argument as passed to dtrace_probe_create().
 *   dtps_resume will only be called on an enabled probe.  Providers that
 *   provide a dtps_resume entry point will want to take roughly the action
 *   that it takes for dtps_enable.
 *
 * 1.7.3  Return value
 *
 *   None.
 *
 * 1.7.4  Caller's context
 *
 *   Interrupts are disabled.  The DTrace framework is in a state such that the
 *   specified probe cannot be disabled or destroyed for the duration of
 *   dtps_resume().  As interrupts are disabled, the provider is afforded
 *   little latitude; the provider is expected to do no more than a store to
 *   memory.
 *
 * 1.8  void dtps_getargdesc(void *arg, dtrace_id_t id, void *parg,
 *           dtrace_argdesc_t *desc)
 *
 * 1.8.1  Overview
 *
 *   Called to retrieve the argument description for an args[X] variable.
```



```
 *
 * 1.8.2  Arguments and notes
 *
 *   The first argument is the cookie as passed to dtrace_register(). The
 *   second argument is the identifier of the current probe. The third
 *   argument is the probe argument as passed to dtrace_probe_create(). The
 *   fourth argument is a pointer to the argument description.  This
 *   description is both an input and output parameter:  it contains the
 *   index of the desired argument in the dtargd_ndx field, and expects
 *   the other fields to be filled in upon return.  If there is no argument
 *   corresponding to the specified index, the dtargd_ndx field should be set
 *   to DTRACE_ARGNONE.
 *
 * 1.8.3  Return value
 *
 *   None.  The dtargd_ndx, dtargd_native, dtargd_xlate and dtargd_mapping
 *   members of the dtrace_argdesc_t structure are all output values.
 *
 * 1.8.4  Caller's context
 *
 *   dtps_getargdesc() is called from ioctl() context. mod_lock is held, and
 *   the DTrace framework is locked in such a way that providers may not
 *   register or unregister.  This means that the provider may not call any
 *   DTrace API that affects its registration with the framework, including
 *   dtrace_register(), dtrace_unregister(), dtrace_invalidate(), and
 *   dtrace_condense().
 *
 * 1.9  uint64_t dtps_getargval(void *arg, dtrace_id_t id, void *parg,
 *              int argno, int aframes)
 *
 * 1.9.1  Overview
 *
 *   Called to retrieve a value for an argX or args[X] variable.
 *
 * 1.9.2  Arguments and notes
 *
 *   The first argument is the cookie as passed to dtrace_register(). The
 *   second argument is the identifier of the current probe. The third
 *   argument is the probe argument as passed to dtrace_probe_create(). The
 *   fourth argument is the number of the argument (the X in the example in
 *   1.9.1). The fifth argument is the number of stack frames that were used
 *   to get from the actual place in the code that fired the probe to
 *   dtrace_probe() itself, the so-called artificial frames. This argument may
 *   be used to descend an appropriate number of frames to find the correct
 *   values. If this entry point is left NULL, the dtrace_getarg() built-in
 *   function is used.
 *
 * 1.9.3  Return value
 *
 *   The value of the argument.
 *
 * 1.9.4  Caller's context
 *
 *   This is called from within dtrace_probe() meaning that interrupts
 *   are disabled. No locks should be taken within this entry point.
 *
 * 1.10  int dtps_mode(void *arg, dtrace_id_t id, void *parg)
 *
 * 1.10.1  Overview
 *
 *   Called to determine the mode of a fired probe.
 *
 * 1.10.2  Arguments and notes
 *
 *   The first argument is the cookie as passed to dtrace_register(). The
 *   second argument is the identifier of the current probe.  The third
```

```
1619  *   argument is the probe argument as passed to dtrace_probe_create().  This
1620  *   entry point must not be left NULL for providers whose probes allow for
1621  *   mixed mode tracing, that is to say those unanchored probes that can fire
1622  *   during kernel- or user-mode execution.
1623  *
1624  * 1.10.3  Return value
1625  *
1626  *   A bitwise OR that encapsulates both the mode (either DTRACE_MODE_KERNEL
1627  *   or DTRACE_MODE_USER) and the policy when the privilege of the enabling
1628  *   is insufficient for that mode (a combination of DTRACE_MODE_NOPRIV_DROP,
1629  *   DTRACE_MODE_NOPRIV_RESTRICT, and DTRACE_MODE_LIMITEDPRIV_RESTRICT).  If
1630  *   DTRACE_MODE_NOPRIV_DROP bit is set, insufficient privilege will result
1631  *   in the probe firing being silently ignored for the enabling; if the
1632  *   DTRACE_NODE_NOPRIV_RESTRICT bit is set, insufficient privilege will not
1633  *   prevent probe processing for the enabling, but restrictions will be in
1634  *   place that induce a UPRIV fault upon attempt to examine probe arguments
1635  *   or current process state.  If the DTRACE_MODE_LIMITEDPRIV_RESTRICT bit
1636  *   is set, similar restrictions will be placed upon operation if the
1637  *   privilege is sufficient to process the enabling, but does not otherwise
1638  *   entitle the enabling to all zones.  The DTRACE_MODE_NOPRIV_DROP and
1639  *   DTRACE_MODE_NOPRIV_RESTRICT are mutually exclusive (and one of these
1640  *   two policies must be specified), but either may be combined (or not)
1641  *   with DTRACE_MODE_LIMITEDPRIV_RESTRICT.
1628  *   is insufficient for that mode (either DTRACE_MODE_NOPRIV_DROP or
1629  *   DTRACE_MODE_NOPRIV_RESTRICT).  If the policy is DTRACE_MODE_NOPRIV_DROP,
1630  *   insufficient privilege will result in the probe firing being silently
1631  *   ignored for the enabling; if the policy is DTRACE_NODE_NOPRIV_RESTRICT,
1632  *   insufficient privilege will not prevent probe processing for the
1633  *   enabling, but restrictions will be in place that induce a UPRIV fault
1634  *   upon attempt to examine probe arguments or current process state.
1642  *
1643  * 1.10.4  Caller's context
1644  *
1645  *   This is called from within dtrace_probe() meaning that interrupts
1646  *   are disabled. No locks should be taken within this entry point.
1647  *
1648  * 1.11 void dtps_destroy(void *arg, dtrace_id_t id, void *parg)
1649  *
1650  * 1.11.1 Overview
1651  *
1652  *   Called to destroy the specified probe.
1653  *
1654  * 1.11.2 Arguments and notes
1655  *
1656  *   The first argument is the cookie as passed to dtrace_register().  The
1657  *   second argument is the identifier of the probe to be destroyed.  The third
1658  *   argument is the probe argument as passed to dtrace_probe_create().  The
1659  *   provider should free all state associated with the probe.  The framework
1660  *   guarantees that dtps_destroy() is only called for probes that have either
1661  *   been disabled via dtps_disable() or were never enabled via dtps_enable().
1662  *   Once dtps_disable() has been called for a probe, no further call will be
1663  *   made specifying the probe.
1664  *
1665  * 1.11.3 Return value
1666  *
1667  *   None.
1668  *
1669  * 1.11.4 Caller's context
1670  *
1671  *   The DTrace framework is locked in such a way that it may not be called
1672  *   back into at all.  mod_lock is held.  cpu_lock is not held, and may not be
1673  *   acquired.
1674  *
1675  *
1676  * 2 Provider-to-Framework API
1677  *
```



```
1678  * 2.1  Overview
1679  *
1680  * The Provider-to-Framework API provides the mechanism for the provider to
1681  * register itself with the DTrace framework, to create probes, to lookup
1682  * probes and (most importantly) to fire probes.  The Provider-to-Framework
1683  * consists of:
1684  *
1685  *   dtrace_register()       <-- Register a provider with the DTrace framework
1686  *   dtrace_unregister()     <-- Remove a provider's DTrace registration
1687  *   dtrace_invalidate()     <-- Invalidate the specified provider
1688  *   dtrace_condense()       <-- Remove a provider's unenabled probes
1689  *   dtrace_attached()       <-- Indicates whether or not DTrace has attached
1690  *   dtrace_probe_create()   <-- Create a DTrace probe
1691  *   dtrace_probe_lookup()   <-- Lookup a DTrace probe based on its name
1692  *   dtrace_probe_arg()      <-- Return the probe argument for a specific probe
1693  *   dtrace_probe()          <-- Fire the specified probe
1694  *
1695  * 2.2  int dtrace_register(const char *name, const dtrace_pattr_t *pap,
1696  *          uint32_t priv, cred_t *cr, const dtrace_pops_t *pops, void *arg,
1697  *          dtrace_provider_id_t *idp)
1698  *
1699  * 2.2.1  Overview
1700  *
1701  *   dtrace_register() registers the calling provider with the DTrace
1702  *   framework.  It should generally be called by DTrace providers in their
1703  *   attach(9E) entry point.
1704  *
1705  * 2.2.2  Arguments and Notes
1706  *
1707  *   The first argument is the name of the provider.  The second argument is a
1708  *   pointer to the stability attributes for the provider.  The third argument
1709  *   is the privilege flags for the provider, and must be some combination of:
1710  *
1711  *     DTRACE_PRIV_NONE     <= All users may enable probes from this provider
1712  *
1713  *     DTRACE_PRIV_PROC     <= Any user with privilege of PRIV_DTRACE_PROC may
1714  *                             enable probes from this provider
1715  *
1716  *     DTRACE_PRIV_USER     <= Any user with privilege of PRIV_DTRACE_USER may
1717  *                             enable probes from this provider
1718  *
1719  *     DTRACE_PRIV_KERNEL   <= Any user with privilege of PRIV_DTRACE_KERNEL
1720  *                             may enable probes from this provider
1721  *
1722  *     DTRACE_PRIV_OWNER    <= This flag places an additional constraint on
1723  *                             the privilege requirements above. These probes
1724  *                             require either (a) a user ID matching the user
1725  *                             ID of the cred passed in the fourth argument
1726  *                             or (b) the PRIV_PROC_OWNER privilege.
1727  *
1728  *     DTRACE_PRIV_ZONEOWNER<= This flag places an additional constraint on
1729  *                             the privilege requirements above. These probes
1730  *                             require either (a) a zone ID matching the zone
1731  *                             ID of the cred passed in the fourth argument
1732  *                             or (b) the PRIV_PROC_ZONE privilege.
1733  *
1734  *   Note that these flags designate the _visibility_ of the probes, not
1735  *   the conditions under which they may or may not fire.
1736  *
1737  *   The fourth argument is the credential that is associated with the
1738  *   provider.  This argument should be NULL if the privilege flags don't
1739  *   include DTRACE_PRIV_OWNER or DTRACE_PRIV_ZONEOWNER.  If non-NULL, the
1740  *   framework stashes the uid and zoneid represented by this credential
1741  *   for use at probe-time, in implicit predicates.  These limit visibility
1742  *   of the probes to users and/or zones which have sufficient privilege to
1743  *   access them.
```

```
1744  *
1745  *   The fifth argument is a DTrace provider operations vector, which provides
1746  *   the implementation for the Framework-to-Provider API.  (See Section 1,
1747  *   above.)  This must be non-NULL, and each member must be non-NULL.  The
1748  *   exceptions to this are (1) the dtps_provide() and dtps_provide_module()
1749  *   members (if the provider so desires, _one_ of these members may be left
1750  *   NULL -- denoting that the provider only implements the other) and (2)
1751  *   the dtps_suspend() and dtps_resume() members, which must either both be
1752  *   NULL or both be non-NULL.
1753  *
1754  *   The sixth argument is a cookie to be specified as the first argument for
1755  *   each function in the Framework-to-Provider API.  This argument may have
1756  *   any value.
1757  *
1758  *   The final argument is a pointer to dtrace_provider_id_t.  If
1759  *   dtrace_register() successfully completes, the provider identifier will be
1760  *   stored in the memory pointed to be this argument.  This argument must be
1761  *   non-NULL.
1762  *
1763  * 2.2.3  Return value
1764  *
1765  *   On success, dtrace_register() returns 0 and stores the new provider's
1766  *   identifier into the memory pointed to by the idp argument.  On failure,
1767  *   dtrace_register() returns an errno:
1768  *
1769  *     EINVAL   The arguments passed to dtrace_register() were somehow invalid.
1770  *              This may because a parameter that must be non-NULL was NULL,
1771  *              because the name was invalid (either empty or an illegal
1772  *              provider name) or because the attributes were invalid.
1773  *
1774  *   No other failure code is returned.
1775  *
1776  * 2.2.4  Caller's context
1777  *
1778  *   dtrace_register() may induce calls to dtrace_provide(); the provider must
1779  *   hold no locks across dtrace_register() that may also be acquired by
1780  *   dtrace_provide().  cpu_lock and mod_lock must not be held.
1781  *
1782  * 2.3  int dtrace_unregister(dtrace_provider_t id)
1783  *
1784  * 2.3.1  Overview
1785  *
1786  *   Unregisters the specified provider from the DTrace framework.  It should
1787  *   generally be called by DTrace providers in their detach(9E) entry point.
1788  *
1789  * 2.3.2  Arguments and Notes
1790  *
1791  *   The only argument is the provider identifier, as returned from a
1792  *   successful call to dtrace_register().  As a result of calling
1793  *   dtrace_unregister(), the DTrace framework will call back into the provider
1794  *   via the dtps_destroy() entry point.  Once dtrace_unregister() successfully
1795  *   completes, however, the DTrace framework will no longer make calls through
1796  *   the Framework-to-Provider API.
1797  *
1798  * 2.3.3  Return value
1799  *
1800  *   On success, dtrace_unregister returns 0.  On failure, dtrace_unregister()
1801  *   returns an errno:
1802  *
1803  *     EBUSY    There are currently processes that have the DTrace pseudodevice
1804  *              open, or there exists an anonymous enabling that hasn't yet
1805  *              been claimed.
1806  *
1807  *   No other failure code is returned.
1808  *
1809  * 2.3.4  Caller's context
```



```
1810  *
1811  *   Because a call to dtrace_unregister() may induce calls through the
1812  *   Framework-to-Provider API, the caller may not hold any lock across
1813  *   dtrace_register() that is also acquired in any of the Framework-to-
1814  *   Provider API functions.  Additionally, mod_lock may not be held.
1815  *
1816  * 2.4  void dtrace_invalidate(dtrace_provider_id_t id)
1817  *
1818  * 2.4.1  Overview
1819  *
1820  *   Invalidates the specified provider.  All subsequent probe lookups for the
1821  *   specified provider will fail, but its probes will not be removed.
1822  *
1823  * 2.4.2  Arguments and note
1824  *
1825  *   The only argument is the provider identifier, as returned from a
1826  *   successful call to dtrace_register().  In general, a provider's probes
1827  *   always remain valid; dtrace_invalidate() is a mechanism for invalidating
1828  *   an entire provider, regardless of whether or not probes are enabled or
1829  *   not.  Note that dtrace_invalidate() will _not_ prevent already enabled
1830  *   probes from firing -- it will merely prevent any new enablings of the
1831  *   provider's probes.
1832  *
1833  * 2.5 int dtrace_condense(dtrace_provider_id_t id)
1834  *
1835  * 2.5.1  Overview
1836  *
1837  *   Removes all the unenabled probes for the given provider. This function is
1838  *   not unlike dtrace_unregister(), except that it doesn't remove the
1839  *   provider just as many of its associated probes as it can.
1840  *
1841  * 2.5.2  Arguments and Notes
1842  *
1843  *   As with dtrace_unregister(), the sole argument is the provider identifier
1844  *   as returned from a successful call to dtrace_register().  As a result of
1845  *   calling dtrace_condense(), the DTrace framework will call back into the
1846  *   given provider's dtps_destroy() entry point for each of the provider's
1847  *   unenabled probes.
1848  *
1849  * 2.5.3  Return value
1850  *
1851  *   Currently, dtrace_condense() always returns 0.  However, consumers of this
1852  *   function should check the return value as appropriate; its behavior may
1853  *   change in the future.
1854  *
1855  * 2.5.4  Caller's context
1856  *
1857  *   As with dtrace_unregister(), the caller may not hold any lock across
1858  *   dtrace_condense() that is also acquired in the provider's entry points.
1859  *   Also, mod_lock may not be held.
1860  *
1861  * 2.6 int dtrace_attached()
1862  *
1863  * 2.6.1  Overview
1864  *
1865  *   Indicates whether or not DTrace has attached.
1866  *
1867  * 2.6.2  Arguments and Notes
1868  *
1869  *   For most providers, DTrace makes initial contact beyond registration.
1870  *   That is, once a provider has registered with DTrace, it waits to hear
1871  *   from DTrace to create probes.  However, some providers may wish to
1872  *   proactively create probes without first being told by DTrace to do so.
1873  *   If providers wish to do this, they must first call dtrace_attached() to
1874  *   determine if DTrace itself has attached.  If dtrace_attached() returns 0,
1875  *   the provider must not make any other Provider-to-Framework API call.
```

```
 *
 * 2.6.3  Return value
 *
 *   dtrace_attached() returns 1 if DTrace has attached, 0 otherwise.
 *
 * 2.7  int dtrace_probe_create(dtrace_provider_t id, const char *mod,
 *          const char *func, const char *name, int aframes, void *arg)
 *
 * 2.7.1  Overview
 *
 *   Creates a probe with specified module name, function name, and name.
 *
 * 2.7.2  Arguments and Notes
 *
 *   The first argument is the provider identifier, as returned from a
 *   successful call to dtrace_register().  The second, third, and fourth
 *   arguments are the module name, function name, and probe name,
 *   respectively.  Of these, module name and function name may both be NULL
 *   (in which case the probe is considered to be unanchored), or they may both
 *   be non-NULL.  The name must be non-NULL, and must point to a non-empty
 *   string.
 *
 *   The fifth argument is the number of artificial stack frames that will be
 *   found on the stack when dtrace_probe() is called for the new probe.  These
 *   artificial frames will be automatically be pruned should the stack() or
 *   stackdepth() functions be called as part of one of the probe's ECBs.  If
 *   the parameter doesn't add an artificial frame, this parameter should be
 *   zero.
 *
 *   The final argument is a probe argument that will be passed back to the
 *   provider when a probe-specific operation is called.  (e.g., via
 *   dtps_enable(), dtps_disable(), etc.)
 *
 *   Note that it is up to the provider to be sure that the probe that it
 *   creates does not already exist -- if the provider is unsure of the probe's
 *   existence, it should assure its absence with dtrace_probe_lookup() before
 *   calling dtrace_probe_create().
 *
 * 2.7.3  Return value
 *
 *   dtrace_probe_create() always succeeds, and always returns the identifier
 *   of the newly-created probe.
 *
 * 2.7.4  Caller's context
 *
 *   While dtrace_probe_create() is generally expected to be called from
 *   dtps_provide() and/or dtps_provide_module(), it may be called from other
 *   non-DTrace contexts.  Neither cpu_lock nor mod_lock may be held.
 *
 * 2.8  dtrace_id_t dtrace_probe_lookup(dtrace_provider_t id, const char *mod,
 *          const char *func, const char *name)
 *
 * 2.8.1  Overview
 *
 *   Looks up a probe based on provdider and one or more of module name,
 *   function name and probe name.
 *
 * 2.8.2  Arguments and Notes
 *
 *   The first argument is the provider identifier, as returned from a
 *   successful call to dtrace_register().  The second, third, and fourth
 *   arguments are the module name, function name, and probe name,
 *   respectively.  Any of these may be NULL; dtrace_probe_lookup() will return
 *   the identifier of the first probe that is provided by the specified
 *   provider and matches all of the non-NULL matching criteria.
 *   dtrace_probe_lookup() is generally used by a provider to be check the
```



```
 *   existence of a probe before creating it with dtrace_probe_create().
 *
 * 2.8.3  Return value
 *
 *   If the probe exists, returns its identifier.  If the probe does not exist,
 *   return DTRACE_IDNONE.
 *
 * 2.8.4  Caller's context
 *
 *   While dtrace_probe_lookup() is generally expected to be called from
 *   dtps_provide() and/or dtps_provide_module(), it may also be called from
 *   other non-DTrace contexts.  Neither cpu_lock nor mod_lock may be held.
 *
 * 2.9  void *dtrace_probe_arg(dtrace_provider_t id, dtrace_id_t probe)
 *
 * 2.9.1  Overview
 *
 *   Returns the probe argument associated with the specified probe.
 *
 * 2.9.2  Arguments and Notes
 *
 *   The first argument is the provider identifier, as returned from a
 *   successful call to dtrace_register().  The second argument is a probe
 *   identifier, as returned from dtrace_probe_lookup() or
 *   dtrace_probe_create().  This is useful if a probe has multiple
 *   provider-specific components to it:  the provider can create the probe
 *   once with provider-specific state, and then add to the state by looking
 *   up the probe based on probe identifier.
 *
 * 2.9.3  Return value
 *
 *   Returns the argument associated with the specified probe.  If the
 *   specified probe does not exist, or if the specified probe is not provided
 *   by the specified provider, NULL is returned.
 *
 * 2.9.4  Caller's context
 *
 *   While dtrace_probe_arg() is generally expected to be called from
 *   dtps_provide() and/or dtps_provide_module(), it may also be called from
 *   other non-DTrace contexts.  Neither cpu_lock nor mod_lock may be held.
 *
 * 2.10  void dtrace_probe(dtrace_id_t probe, uintptr_t arg0, uintptr_t arg1,
 *             uintptr_t arg2, uintptr_t arg3, uintptr_t arg4)
 *
 * 2.10.1  Overview
 *
 *   The epicenter of DTrace:  fires the specified probes with the specified
 *   arguments.
 *
 * 2.10.2  Arguments and Notes
 *
 *   The first argument is a probe identifier as returned by
 *   dtrace_probe_create() or dtrace_probe_lookup().  The second through sixth
 *   arguments are the values to which the D variables "arg0" through "arg4"
 *   will be mapped.
 *
 *   dtrace_probe() should be called whenever the specified probe has fired --
 *   however the provider defines it.
 *
 * 2.10.3  Return value
 *
 *   None.
 *
 * 2.10.4  Caller's context
 *
 *   dtrace_probe() may be called in virtually any context:  kernel, user,
```

```
2008  *   interrupt, high-level interrupt, with arbitrary adaptive locks held, with
2009  *   dispatcher locks held, with interrupts disabled, etc.  The only latitude
2010  *   that must be afforded to DTrace is the ability to make calls within
2011  *   itself (and to its in-kernel subroutines) and the ability to access
2012  *   arbitrary (but mapped) memory.  On some platforms, this constrains
2013  *   context.  For example, on UltraSPARC, dtrace_probe() cannot be called
2014  *   from any context in which TL is greater than zero.  dtrace_probe() may
2015  *   also not be called from any routine which may be called by dtrace_probe()
2016  *   -- which includes functions in the DTrace framework and some in-kernel
2017  *   DTrace subroutines.  All such functions "dtrace_"; providers that
2018  *   instrument the kernel arbitrarily should be sure to not instrument these
2019  *   routines.
2020  */
2021 typedef struct dtrace_pops {
2022         void (*dtps_provide)(void *arg, const dtrace_probedesc_t *spec);
2023         void (*dtps_provide_module)(void *arg, struct modctl *mp);
2024         int (*dtps_enable)(void *arg, dtrace_id_t id, void *parg);
2025         void (*dtps_disable)(void *arg, dtrace_id_t id, void *parg);
2026         void (*dtps_suspend)(void *arg, dtrace_id_t id, void *parg);
2027         void (*dtps_resume)(void *arg, dtrace_id_t id, void *parg);
2028         void (*dtps_getargdesc)(void *arg, dtrace_id_t id, void *parg,
2029             dtrace_argdesc_t *desc);
2030         uint64_t (*dtps_getargval)(void *arg, dtrace_id_t id, void *parg,
2031             int argno, int aframes);
2032         int (*dtps_mode)(void *arg, dtrace_id_t id, void *parg);
2033         void (*dtps_destroy)(void *arg, dtrace_id_t id, void *parg);
2034 } dtrace_pops_t;

2036 #define DTRACE_MODE_KERNEL                      0x01
2037 #define DTRACE_MODE_USER                        0x02
2038 #define DTRACE_MODE_NOPRIV_DROP                 0x10
2039 #define DTRACE_MODE_NOPRIV_RESTRICT             0x20
2040 #define DTRACE_MODE_LIMITEDPRIV_RESTRICT        0x40

2042 typedef uintptr_t       dtrace_provider_id_t;

2044 extern int dtrace_register(const char *, const dtrace_pattr_t *, uint32_t,
2045     cred_t *, const dtrace_pops_t *, void *, dtrace_provider_id_t *);
2046 extern int dtrace_unregister(dtrace_provider_id_t);
2047 extern int dtrace_condense(dtrace_provider_id_t);
2048 extern void dtrace_invalidate(dtrace_provider_id_t);
2049 extern dtrace_id_t dtrace_probe_lookup(dtrace_provider_id_t, const char *,
2050     const char *, const char *);
2051 extern dtrace_id_t dtrace_probe_create(dtrace_provider_id_t, const char *,
2052     const char *, const char *, int, void *);
2053 extern void *dtrace_probe_arg(dtrace_provider_id_t, dtrace_id_t);
2054 extern void dtrace_probe(dtrace_id_t, uintptr_t arg0, uintptr_t arg1,
2055     uintptr_t arg2, uintptr_t arg3, uintptr_t arg4);

2057 /*
2058  * DTrace Meta Provider API
2059  *
2060  * The following functions are implemented by the DTrace framework and are
2061  * used to implement meta providers. Meta providers plug into the DTrace
2062  * framework and are used to instantiate new providers on the fly. At
2063  * present, there is only one type of meta provider and only one meta
2064  * provider may be registered with the DTrace framework at a time. The
2065  * sole meta provider type provides user-land static tracing facilities
2066  * by taking meta probe descriptions and adding a corresponding provider
2067  * into the DTrace framework.
2068  *
2069  * 1 Framework-to-Provider
2070  *
2071  * 1.1 Overview
2072  *
2073  * The Framework-to-Provider API is represented by the dtrace_mops structure
```



```
2074  * that the meta provider passes to the framework when registering itself as
2075  * a meta provider. This structure consists of the following members:
2076  *
2077  *   dtms_create_probe()        <-- Add a new probe to a created provider
2078  *   dtms_provide_pid()         <-- Create a new provider for a given process
2079  *   dtms_remove_pid()          <-- Remove a previously created provider
2080  *
2081  * 1.2  void dtms_create_probe(void *arg, void *parg,
2082  *           dtrace_helper_probedesc_t *probedesc);
2083  *
2084  * 1.2.1  Overview
2085  *
2086  *   Called by the DTrace framework to create a new probe in a provider
2087  *   created by this meta provider.
2088  *
2089  * 1.2.2  Arguments and notes
2090  *
2091  *   The first argument is the cookie as passed to dtrace_meta_register().
2092  *   The second argument is the provider cookie for the associated provider;
2093  *   this is obtained from the return value of dtms_provide_pid(). The third
2094  *   argument is the helper probe description.
2095  *
2096  * 1.2.3  Return value
2097  *
2098  *   None
2099  *
2100  * 1.2.4  Caller's context
2101  *
2102  *   dtms_create_probe() is called from either ioctl() or module load context.
2103  *   The DTrace framework is locked in such a way that meta providers may not
2104  *   register or unregister. This means that the meta provider cannot call
2105  *   dtrace_meta_register() or dtrace_meta_unregister(). However, the context is
2106  *   such that the provider may (and is expected to) call provider-related
2107  *   DTrace provider APIs including dtrace_probe_create().
2108  *
2109  * 1.3  void *dtms_provide_pid(void *arg, dtrace_meta_provider_t *mprov,
2110  *            pid_t pid)
2111  *
2112  * 1.3.1  Overview
2113  *
2114  *   Called by the DTrace framework to instantiate a new provider given the
2115  *   description of the provider and probes in the mprov argument. The
2116  *   meta provider should call dtrace_register() to insert the new provider
2117  *   into the DTrace framework.
2118  *
2119  * 1.3.2  Arguments and notes
2120  *
2121  *   The first argument is the cookie as passed to dtrace_meta_register().
2122  *   The second argument is a pointer to a structure describing the new
2123  *   helper provider. The third argument is the process identifier for
2124  *   process associated with this new provider. Note that the name of the
2125  *   provider as passed to dtrace_register() should be the contatenation of
2126  *   the dtmpb_provname member of the mprov argument and the processs
2127  *   identifier as a string.
2128  *
2129  * 1.3.3  Return value
2130  *
2131  *   The cookie for the provider that the meta provider creates. This is
2132  *   the same value that it passed to dtrace_register().
2133  *
2134  * 1.3.4  Caller's context
2135  *
2136  *   dtms_provide_pid() is called from either ioctl() or module load context.
2137  *   The DTrace framework is locked in such a way that meta providers may not
2138  *   register or unregister. This means that the meta provider cannot call
2139  *   dtrace_meta_register() or dtrace_meta_unregister(). However, the context
```

```
2140  *   is such that the provider may -- and is expected to --  call
2141  *   provider-related DTrace provider APIs including dtrace_register().
2142  *
2143  * 1.4  void dtms_remove_pid(void *arg, dtrace_meta_provider_t *mprov,
2144  *           pid_t pid)
2145  *
2146  * 1.4.1  Overview
2147  *
2148  *   Called by the DTrace framework to remove a provider that had previously
2149  *   been instantiated via the dtms_provide_pid() entry point. The meta
2150  *   provider need not remove the provider immediately, but this entry
2151  *   point indicates that the provider should be removed as soon as possible
2152  *   using the dtrace_unregister() API.
2153  *
2154  * 1.4.2  Arguments and notes
2155  *
2156  *   The first argument is the cookie as passed to dtrace_meta_register().
2157  *   The second argument is a pointer to a structure describing the helper
2158  *   provider. The third argument is the process identifier for process
2159  *   associated with this new provider.
2160  *
2161  * 1.4.3  Return value
2162  *
2163  *   None
2164  *
2165  * 1.4.4  Caller's context
2166  *
2167  *   dtms_remove_pid() is called from either ioctl() or exit() context.
2168  *   The DTrace framework is locked in such a way that meta providers may not
2169  *   register or unregister. This means that the meta provider cannot call
2170  *   dtrace_meta_register() or dtrace_meta_unregister(). However, the context
2171  *   is such that the provider may -- and is expected to -- call
2172  *   provider-related DTrace provider APIs including dtrace_unregister().
2173  */
2174 typedef struct dtrace_helper_probedesc {
2175         char *dthpb_mod;                       /* probe module */
2176         char *dthpb_func;                      /* probe function */
2177         char *dthpb_name;                      /* probe name */
2178         uint64_t dthpb_base;                   /* base address */
2179         uint32_t *dthpb_offs;                  /* offsets array */
2180         uint32_t *dthpb_enoffs;                /* is-enabled offsets array */
2181         uint32_t dthpb_noffs;                  /* offsets count */
2182         uint32_t dthpb_nenoffs;                /* is-enabled offsets count */
2183         uint8_t *dthpb_args;                   /* argument mapping array */
2184         uint8_t dthpb_xargc;                   /* translated argument count */
2185         uint8_t dthpb_nargc;                   /* native argument count */
2186         char *dthpb_xtypes;                    /* translated types strings */
2187         char *dthpb_ntypes;                    /* native types strings */
2188 } dtrace_helper_probedesc_t;
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    2764 Sat Jun 23 09:31:29 2012
new/usr/src/uts/common/sys/sdt_impl.h
2917 DTrace in a zone should have limited provider access
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 */

#ifndef _SYS_SDT_IMPL_H
#define _SYS_SDT_IMPL_H

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

#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/dtrace.h>

#if defined(__i386) || defined(__amd64)
typedef uint8_t sdt_instr_t;
#else
typedef uint32_t sdt_instr_t;
#endif

typedef struct sdt_provider {
        char                    *sdtp_name;     /* name of provider */
        char                    *sdtp_prefix;   /* prefix for probe names */
        dtrace_pattr_t          *sdtp_attr;     /* stability attributes */
        uint32_t                sdtp_priv;      /* privilege, if any */
        dtrace_provider_id_t    sdtp_id;        /* provider ID */
} sdt_provider_t;
```

*unchanged_portion_omitted_*

```
extern void sdt_getargdesc(void *, dtrace_id_t, void *, dtrace_argdesc_t *);
extern int sdt_mode(void *, dtrace_id_t, void *);

#ifdef  __cplusplus
}
#endif
```

*unchanged_portion_omitted_*

```
**********************************************************
   13145 Sat Jun 23 09:31:29 2012
new/usr/src/uts/intel/dtrace/sdt.c
2917 DTrace in a zone should have limited provider access
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 */

#include <sys/modctl.h>
#include <sys/sunddi.h>
#include <sys/dtrace.h>
#include <sys/kobj.h>
#include <sys/stat.h>
#include <sys/conf.h>
#include <vm/seg_kmem.h>
#include <sys/stack.h>
#include <sys/frame.h>
#include <sys/dtrace_impl.h>
#include <sys/cmn_err.h>
#include <sys/sysmacros.h>
#include <sys/privregs.h>
#include <sys/sdt_impl.h>

#define SDT_PATCHVAL    0xf0
#define SDT_ADDR2NDX(addr)      ((((uintptr_t)(addr)) >> 4) & sdt_probetab_mask)
#define SDT_PROBETAB_SIZE       0x1000          /* 4k entries -- 16K total */

static dev_info_t               *sdt_devi;
static int                      sdt_verbose = 0;
static sdt_probe_t              **sdt_probetab;
static int                      sdt_probetab_size;
static int                      sdt_probetab_mask;

/*ARGSUSED*/
static int
sdt_invop(uintptr_t addr, uintptr_t *stack, uintptr_t eax)
{
        uintptr_t stack0, stack1, stack2, stack3, stack4;
        int i = 0;
        sdt_probe_t *sdt = sdt_probetab[SDT_ADDR2NDX(addr)];
```



```
#ifdef __amd64
        /*
         * On amd64, stack[0] contains the dereferenced stack pointer,
         * stack[1] contains savfp, stack[2] contains savpc.  We want
         * to step over these entries.
         */
        i += 3;
#endif

        for (; sdt != NULL; sdt = sdt->sdp_hashnext) {
                if ((uintptr_t)sdt->sdp_patchpoint == addr) {
                        /*
                         * When accessing the arguments on the stack, we must
                         * protect against accessing beyond the stack.  We can
                         * safely set NOFAULT here -- we know that interrupts
                         * are already disabled.
                         */
                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
                        stack0 = stack[i++];
                        stack1 = stack[i++];
                        stack2 = stack[i++];
                        stack3 = stack[i++];
                        stack4 = stack[i++];
                        DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT |
                            CPU_DTRACE_BADADDR);

                        dtrace_probe(sdt->sdp_id, stack0, stack1,
                            stack2, stack3, stack4);

                        return (DTRACE_INVOP_NOP);
                }
        }

        return (0);
}
```

*unchanged_portion_omitted_*

```
/*ARGSUSED*/
static int
sdt_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
{
        sdt_provider_t *prov;

        if (ddi_create_minor_node(devi, "sdt", S_IFCHR,
            0, DDI_PSEUDO, NULL) == DDI_FAILURE) {
                cmn_err(CE_NOTE, "/dev/sdt couldn't create minor node");
                ddi_remove_minor_node(devi, NULL);
                return (DDI_FAILURE);
        }

        ddi_report_dev(devi);
        sdt_devi = devi;

        if (sdt_probetab_size == 0)
                sdt_probetab_size = SDT_PROBETAB_SIZE;

        sdt_probetab_mask = sdt_probetab_size - 1;
        sdt_probetab =
            kmem_zalloc(sdt_probetab_size * sizeof (sdt_probe_t *), KM_SLEEP);
        dtrace_invop_add(sdt_invop);

        for (prov = sdt_providers; prov->sdtp_name != NULL; prov++) {
                uint32_t priv;

                if (prov->sdtp_priv == DTRACE_PRIV_NONE) {
```

```
428                         priv = DTRACE_PRIV_KERNEL;
429                         sdt_pops.dtps_mode = NULL;
430                 } else {
431                         priv = prov->sdtp_priv;
432                         ASSERT(priv == DTRACE_PRIV_USER);
433                         sdt_pops.dtps_mode = sdt_mode;
434                 }

436                 if (dtrace_register(prov->sdtp_name, prov->sdtp_attr,
437                     priv, NULL, &sdt_pops, prov, &prov->sdtp_id) != 0) {
423                     DTRACE_PRIV_KERNEL, NULL,
424                     &sdt_pops, prov, &prov->sdtp_id) != 0) {
438                         cmn_err(CE_WARN, "failed to register sdt provider %s",
439                             prov->sdtp_name);
440                 }
441         }

443         return (DDI_SUCCESS);
444 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   11768 Sat Jun 23 09:31:29 2012
new/usr/src/uts/sparc/dtrace/sdt.c
2917 DTrace in a zone should have limited provider access
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 */

#include <sys/modctl.h>
#include <sys/sunddi.h>
#include <sys/dtrace.h>
#include <sys/kobj.h>
#include <sys/stat.h>
#include <sys/conf.h>
#include <vm/seg_kmem.h>
#include <sys/stack.h>
#include <sys/sdt_impl.h>

static dev_info_t               *sdt_devi;

int sdt_verbose = 0;

#define SDT_REG_G0              0
#define SDT_REG_O0              8
#define SDT_REG_O1              9
#define SDT_REG_O2              10
#define SDT_REG_O3              11
#define SDT_REG_O4              12
#define SDT_REG_O5              13
#define SDT_REG_I0              24
#define SDT_REG_I1              25
#define SDT_REG_I2              26
#define SDT_REG_I3              27
#define SDT_REG_I4              28
#define SDT_REG_I5              29

#define SDT_SIMM13_MASK         0x1fff
#define SDT_SIMM13_MAX          ((int32_t)0xfff)
#define SDT_CALL(from, to)      (((uint32_t)1 << 30) | \
                                (((uintptr_t)(to) - (uintptr_t)(from) >> 2) & \
```



```
                                0x3fffffff))
#define SDT_SAVE                (0x9de3a000 | (-SA(MINFRAME) & SDT_SIMM13_MASK))
#define SDT_RET                 0x81c7e008
#define SDT_RESTORE             0x81e80000

#define SDT_OP_SETHI            0x1000000
#define SDT_OP_OR               0x80100000

#define SDT_FMT2_RD_SHIFT       25
#define SDT_IMM22_SHIFT         10
#define SDT_IMM22_MASK          0x3fffff
#define SDT_IMM10_MASK          0x3ff

#define SDT_FMT3_RD_SHIFT       25
#define SDT_FMT3_RS1_SHIFT      14
#define SDT_FMT3_RS2_SHIFT      0
#define SDT_FMT3_IMM            (1 << 13)

#define SDT_MOV(rs, rd) \
        (SDT_OP_OR | (SDT_REG_G0 << SDT_FMT3_RS1_SHIFT) | \
        ((rs) << SDT_FMT3_RS2_SHIFT) | ((rd) << SDT_FMT3_RD_SHIFT))

#define SDT_ORLO(rs, val, rd) \
        (SDT_OP_OR | ((rs) << SDT_FMT3_RS1_SHIFT) | \
        ((rd) << SDT_FMT3_RD_SHIFT) | SDT_FMT3_IMM | ((val) & SDT_IMM10_MASK))

#define SDT_ORSIMM13(rs, val, rd) \
        (SDT_OP_OR | ((rs) << SDT_FMT3_RS1_SHIFT) | \
        ((rd) << SDT_FMT3_RD_SHIFT) | SDT_FMT3_IMM | ((val) & SDT_SIMM13_MASK))

#define SDT_SETHI(val, reg)     \
        (SDT_OP_SETHI | (reg << SDT_FMT2_RD_SHIFT) | \
        ((val >> SDT_IMM22_SHIFT) & SDT_IMM22_MASK))

#define SDT_ENTRY_SIZE  (11 * sizeof (uint32_t))

static void
sdt_initialize(sdt_probe_t *sdp, uint32_t **trampoline)
{
        uint32_t *instr = *trampoline;

        *instr++ = SDT_SAVE;

        if (sdp->sdp_id > (uint32_t)SDT_SIMM13_MAX)  {
                *instr++ = SDT_SETHI(sdp->sdp_id, SDT_REG_O0);
                *instr++ = SDT_ORLO(SDT_REG_O0, sdp->sdp_id, SDT_REG_O0);
        } else {
                *instr++ = SDT_ORSIMM13(SDT_REG_G0, sdp->sdp_id, SDT_REG_O0);
        }

        *instr++ = SDT_MOV(SDT_REG_I0, SDT_REG_O1);
        *instr++ = SDT_MOV(SDT_REG_I1, SDT_REG_O2);
        *instr++ = SDT_MOV(SDT_REG_I2, SDT_REG_O3);
        *instr++ = SDT_MOV(SDT_REG_I3, SDT_REG_O4);
        *instr = SDT_CALL(instr, dtrace_probe);
        instr++;
        *instr++ = SDT_MOV(SDT_REG_I4, SDT_REG_O5);

        *instr++ = SDT_RET;
        *instr++ = SDT_RESTORE;
        *trampoline = instr;
}
```

_____***unchanged_portion_omitted_***

```
static int
sdt_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
```

```
 357 {
 358         sdt_provider_t *prov;

 360         switch (cmd) {
 361         case DDI_ATTACH:
 362                 break;
 363         case DDI_RESUME:
 364                 return (DDI_SUCCESS);
 365         default:
 366                 return (DDI_FAILURE);
 367         }

 369         if (ddi_create_minor_node(devi, "sdt", S_IFCHR, 0,
 370             DDI_PSEUDO, NULL) == DDI_FAILURE) {
 371                 ddi_remove_minor_node(devi, NULL);
 372                 return (DDI_FAILURE);
 373         }

 375         ddi_report_dev(devi);
 376         sdt_devi = devi;

 378         for (prov = sdt_providers; prov->sdtp_name != NULL; prov++) {
 379                 uint32_t priv;

 381                 if (prov->sdtp_priv == DTRACE_PRIV_NONE) {
 382                         priv = DTRACE_PRIV_KERNEL;
 383                         sdt_pops.dtps_mode = NULL;
 384                 } else {
 385                         priv = prov->sdtp_priv;
 386                         ASSERT(priv == DTRACE_PRIV_USER);
 387                         sdt_pops.dtps_mode = sdt_mode;
 388                 }

 390                 if (dtrace_register(prov->sdtp_name, prov->sdtp_attr,
 391                     priv, NULL, &sdt_pops, prov, &prov->sdtp_id) != 0) {
 377                     DTRACE_PRIV_KERNEL, NULL,
 378                     &sdt_pops, prov, &prov->sdtp_id) != 0) {
 392                         cmn_err(CE_WARN, "failed to register sdt provider %s",
 393                             prov->sdtp_name);
 394                 }
 395         }

 397         return (DDI_SUCCESS);
 398 }
```

*unchanged_portion_omitted_*