```
**********************************************************
    2951 Tue Jan 14 16:49:58 2014
new/usr/src/cmd/dtrace/test/tst/common/aggs/tst.subr.d
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

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

/*
 * Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#include <sys/dtrace.h>

#define INTFUNC(x)                      \
        BEGIN                           \
        /*DSTYLED*/                     \
        {                               \
                subr++;                 \
                @[(long)x] = sum(1);    \
        /*DSTYLED*/                     \
        }

#define STRFUNC(x)                      \
        BEGIN                           \
        /*DSTYLED*/                     \
        {                               \
                subr++;                 \
                @str[x] = sum(1);       \
        /*DSTYLED*/                     \
        }

#define VOIDFUNC(x)                     \
        BEGIN                           \
        /*DSTYLED*/                     \
        {                               \
                subr++;                 \
        /*DSTYLED*/                     \
        }

INTFUNC(rand())
INTFUNC(mutex_owned(&‘cpu_lock))
INTFUNC(mutex_owner(&‘cpu_lock))
```



```
INTFUNC(mutex_type_adaptive(&‘cpu_lock))
INTFUNC(mutex_type_spin(&‘cpu_lock))
INTFUNC(rw_read_held(&‘vfssw_lock))
INTFUNC(rw_write_held(&‘vfssw_lock))
INTFUNC(rw_iswriter(&‘vfssw_lock))
INTFUNC(copyin(NULL, 1))
STRFUNC(copyinstr(NULL, 1))
INTFUNC(speculation())
INTFUNC(progenyof($pid))
INTFUNC(strlen("fooey"))
VOIDFUNC(copyout)
VOIDFUNC(copyoutstr)
INTFUNC(alloca(10))
VOIDFUNC(bcopy)
VOIDFUNC(copyinto)
INTFUNC(msgdsize(NULL))
INTFUNC(msgsize(NULL))
INTFUNC(getmajor(0))
INTFUNC(getminor(0))
STRFUNC(ddi_pathname(NULL, 0))
STRFUNC(strjoin("foo", "bar"))
STRFUNC(lltostr(12373))
STRFUNC(basename("/var/crash/systemtap"))
STRFUNC(dirname("/var/crash/systemtap"))
STRFUNC(cleanpath("/var/crash/systemtap"))
STRFUNC(strchr("The SystemTap, The.", ’t’))
STRFUNC(strrchr("The SystemTap, The.", ’t’))
STRFUNC(strstr("The SystemTap, The.", "The"))
STRFUNC(strtok("The SystemTap, The.", "T"))
STRFUNC(substr("The SystemTap, The.", 0))
INTFUNC(index("The SystemTap, The.", "The"))
INTFUNC(rindex("The SystemTap, The.", "The"))
INTFUNC(htons(0x1234))
INTFUNC(htonl(0x12345678))
INTFUNC(htonll(0x1234567890abcdefL))
INTFUNC(ntohs(0x1234))
INTFUNC(ntohl(0x12345678))
INTFUNC(ntohll(0x1234567890abcdefL))
STRFUNC(inet_ntoa((ipaddr_t *)alloca(sizeof (ipaddr_t))))
STRFUNC(inet_ntoa6((in6_addr_t *)alloca(sizeof (in6_addr_t))))
STRFUNC(inet_ntop(AF_INET, (void *)alloca(sizeof (ipaddr_t))))
STRFUNC(toupper("foo"))
STRFUNC(tolower("BAR"))
INTFUNC(getf(0))
#endif /* ! codereview */

BEGIN
/subr == DIF_SUBR_MAX + 1/
{
        exit(0);
}

BEGIN
{
        printf("found %d subroutines, expected %d\n", subr, DIF_SUBR_MAX + 1);
        exit(1);
}
```

```
**********************************************************
    2018 Tue Jan 14 16:49:58 2014
new/usr/src/cmd/dtrace/test/tst/common/privs/tst.fds.ksh
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

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

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

tmpin=/tmp/tst.fds.$$.d
tmpout1=/tmp/tst.fds.$$.out1
tmpout2=/tmp/tst.fds.$$.out2

cat > $tmpin <<EOF
#define DUMPFIELD(fd, fmt, field) \
        errmsg = "could not dump field"; \
        printf("%d: field =fmt\n", fd, fds[fd].field);

/*
 * Note that we are explicitly not looking at fi_mount -- it (by design) does
 * not work if not running with kernel permissions.
 */
#define DUMP(fd)        \
        DUMPFIELD(fd, %s, fi_name); \
        DUMPFIELD(fd, %s, fi_dirname); \
        DUMPFIELD(fd, %s, fi_pathname); \
        DUMPFIELD(fd, %d, fi_offset); \
        DUMPFIELD(fd, %s, fi_fs); \
        DUMPFIELD(fd, %o, fi_oflags);

BEGIN
{
        DUMP(0);
        DUMP(1);
        DUMP(2);
        DUMP(3);
        DUMP(4);
        exit(0);
}

ERROR
```



```
{
        printf("error: %s\n", errmsg);
        exit(1);
}
EOF

#
# First, with all privs
#
/usr/sbin/dtrace -q -Cs /dev/stdin < $tmpin > $tmpout2
mv $tmpout2 $tmpout1

#
# And now with only dtrace_proc and dtrace_user -- the output should be
# identical.
#
ppriv -s A=basic,dtrace_proc,dtrace_user $$

/usr/sbin/dtrace -q -Cs /dev/stdin < $tmpin > $tmpout2

echo ">>> $tmpout1"
cat $tmpout1

echo ">>> $tmpout2"
cat $tmpout2

rval=0

if ! cmp $tmpout1 $tmpout2 ; then
        rval=1
fi

rm $tmpout1 $tmpout2 $tmpin
exit $rval
#endif /* ! codereview */
```

```
**********************************************************
    2179 Tue Jan 14 16:49:59 2014
new/usr/src/cmd/dtrace/test/tst/common/privs/tst.getf.ksh
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

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

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

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

/usr/sbin/dtrace -q -Cs /dev/stdin <<EOF

#define CANREAD(field) \
        BEGIN { this->fp = getf(0); errmsg = "can't read field"; \
            printf("field: "); trace(this->fp->field); printf("\n"); }

#define CANTREAD(field) \
        BEGIN { errmsg = ""; this->fp = getf(0); trace(this->fp->field); \
            printf("\nable to successfully read field!"); exit(1); }

CANREAD(f_flag)
CANREAD(f_flag2)
CANREAD(f_vnode)
CANREAD(f_offset)
CANREAD(f_cred)
CANREAD(f_audit_data)
CANREAD(f_count)

/*
 * We can potentially read parts of our cred, but we can't dereference
 * through cr_zone.
 */
CANTREAD(f_cred->cr_zone->zone_id)

CANREAD(f_vnode->v_path)
CANREAD(f_vnode->v_op)
CANREAD(f_vnode->v_op->vnop_name)

CANTREAD(f_vnode->v_flag)
CANTREAD(f_vnode->v_count)
```



```
CANTREAD(f_vnode->v_pages)
CANTREAD(f_vnode->v_type)
CANTREAD(f_vnode->v_vfsmountedhere)
CANTREAD(f_vnode->v_op->vop_open)

BEGIN
{
        errmsg = "";
        this->fp = getf(0);
        this->fp2 = getf(1);

        trace(this->fp->f_vnode);
        printf("\nable to successfully read this->fp!");
        exit(1);
}

BEGIN
{
        errmsg = "";
        this->fp = getf(0);
}

BEGIN
{
        trace(this->fp->f_vnode);
        printf("\nable to successfully read this->fp from prior clause!");
}

BEGIN
{
        exit(0);
}

ERROR
/errmsg != ""/
{
        printf("fatal error: %s", errmsg);
        exit(1);
}

EOF
#endif /* ! codereview */
```

```
**********************************************************
    4157 Tue Jan 14 16:49:59 2014
new/usr/src/cmd/dtrace/test/tst/common/privs/tst.procpriv.ksh
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

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

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

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

#
# When we have dtrace_proc (but lack dtrace_kernel), we expect to be able to
# read certain curpsinfo/curlwpsinfo/curcpu fields even though they require
# reading in-kernel state.  However, there are other fields in these translated
# structures that we know we shouldn't be able to read, as they require reading
# in-kernel state that we cannot read with only dtrace_proc.  Finally, there
# are a few fields that we may or may not be able to read depending on the
# specifics of context.  This test therefore asserts that we can read what we
# think we should be able to, that we can't read what we think we shouldn't be
# able to, and (for purposes of completeness) that we are indifferent about
# what we cannot assert one way or the other.
#
/usr/sbin/dtrace -q -Cs /dev/stdin <<EOF

#define CANREAD(what, field) \
    BEGIN { errmsg = "can't read field from what"; printf("field: "); \
        trace(what->field); printf("\n"); }

#define CANTREAD(what, field) \
    BEGIN { errmsg = ""; trace(what->field); \
        printf("\nable to successfully read field from what!"); exit(1); }

#define MIGHTREAD(what, field) \
    BEGIN { errmsg = ""; printf("field: "); trace(what->field); printf("\n"); }

#define CANREADVAR(vname) \
    BEGIN { errmsg = "can't read vname"; printf("vname: "); \
        trace(vname); printf("\n"); }

#define CANTREADVAR(vname) \
```



```
    BEGIN { errmsg = ""; trace(vname); \
        printf("\nable to successfully read vname!"); exit(1); }

#define MIGHTREADVAR(vname) \
    BEGIN { errmsg = ""; printf("vname: "); trace(vname); printf("\n"); }

CANREAD(curpsinfo, pr_pid)
CANREAD(curpsinfo, pr_nlwp)
CANREAD(curpsinfo, pr_ppid)
CANREAD(curpsinfo, pr_uid)
CANREAD(curpsinfo, pr_euid)
CANREAD(curpsinfo, pr_gid)
CANREAD(curpsinfo, pr_egid)
CANREAD(curpsinfo, pr_addr)
CANREAD(curpsinfo, pr_start)
CANREAD(curpsinfo, pr_fname)
CANREAD(curpsinfo, pr_psargs)
CANREAD(curpsinfo, pr_argc)
CANREAD(curpsinfo, pr_argv)
CANREAD(curpsinfo, pr_envp)
CANREAD(curpsinfo, pr_dmodel)

/*
 * If our p_pgidp points to the same pid structure as our p_pidp, we will
 * be able to read pr_pgid -- but we won't if not.
 */
MIGHTREAD(curpsinfo, pr_pgid)

CANTREAD(curpsinfo, pr_sid)
CANTREAD(curpsinfo, pr_ttydev)
CANTREAD(curpsinfo, pr_projid)
CANTREAD(curpsinfo, pr_zoneid)
CANTREAD(curpsinfo, pr_contract)

CANREAD(curlwpsinfo, pr_flag)
CANREAD(curlwpsinfo, pr_lwpid)
CANREAD(curlwpsinfo, pr_addr)
CANREAD(curlwpsinfo, pr_wchan)
CANREAD(curlwpsinfo, pr_stype)
CANREAD(curlwpsinfo, pr_state)
CANREAD(curlwpsinfo, pr_sname)
CANREAD(curlwpsinfo, pr_syscall)
CANREAD(curlwpsinfo, pr_pri)
CANREAD(curlwpsinfo, pr_onpro)
CANREAD(curlwpsinfo, pr_bindpro)
CANREAD(curlwpsinfo, pr_bindpset)

CANTREAD(curlwpsinfo, pr_clname)
CANTREAD(curlwpsinfo, pr_lgrp)

CANREAD(curcpu, cpu_id)

CANTREAD(curcpu, cpu_pset)
CANTREAD(curcpu, cpu_chip)
CANTREAD(curcpu, cpu_lgrp)
CANTREAD(curcpu, cpu_info)

/*
 * We cannot assert one thing or another about the variable "root":  for those
 * with only dtrace_proc, it will be readable in the global but not readable in
 * the non-global.
 */
MIGHTREADVAR(root)

CANREADVAR(cpu)
CANTREADVAR(pset)
```

```
124 CANTREADVAR(cwd)
125 CANTREADVAR(chip)
126 CANTREADVAR(lgrp)

128 BEGIN
129 {
130         exit(0);
131 }

133 ERROR
134 /errmsg != ""/
135 {
136         printf("fatal error: %s", errmsg);
137         exit(1);
138 }
139 #endif /* ! codereview */
```

```
**********************************************************
    3027 Tue Jan 14 16:49:59 2014
new/usr/src/cmd/dtrace/test/tst/common/privs/tst.providers.ksh
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

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

#
# Copyright (c) 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");
```

```
124         exit(++err);
125 }
126 EOF
127 #endif /* ! codereview */
```

```
**********************************************************
   53829 Tue Jan 14 16:49:59 2014
new/usr/src/lib/libdtrace/common/dt_open.c
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

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

/*
 * Copyright (c) 2003, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 * Copyright (c) 2011, Joyent, Inc. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#include <sys/types.h>
#include <sys/modctl.h>
#include <sys/systeminfo.h>
#include <sys/resource.h>

#include <libelf.h>
#include <strings.h>
#include <alloca.h>
#include <limits.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <assert.h>

#define _POSIX_PTHREAD_SEMANTICS
#include <dirent.h>
#undef  _POSIX_PTHREAD_SEMANTICS

#include <dt_impl.h>
#include <dt_program.h>
#include <dt_module.h>
#include <dt_printf.h>
#include <dt_string.h>
#include <dt_provider.h>

/*
 * Stability and versioning definitions.  These #defines are used in the tables
```



```
 * of identifiers below to fill in the attribute and version fields associated
 * with each identifier.  The DT_ATTR_* macros are a convenience to permit more
 * concise declarations of common attributes such as Stable/Stable/Common.  The
 * DT_VERS_* macros declare the encoded integer values of all versions used so
 * far.  DT_VERS_LATEST must correspond to the latest version value among all
 * versions exported by the D compiler.  DT_VERS_STRING must be an ASCII string
 * that contains DT_VERS_LATEST within it along with any suffixes (e.g. Beta).
 * You must update DT_VERS_LATEST and DT_VERS_STRING when adding a new version,
 * and then add the new version to the _dtrace_versions[] array declared below.
 * Refer to the Solaris Dynamic Tracing Guide Stability and Versioning chapters
 * respectively for an explanation of these DTrace features and their values.
 *
 * NOTE: Although the DTrace versioning scheme supports the labeling and
 *       introduction of incompatible changes (e.g. dropping an interface in a
 *       major release), the libdtrace code does not currently support this.
 *       All versions are assumed to strictly inherit from one another.  If
 *       we ever need to provide divergent interfaces, this will need work.
 */
#define DT_ATTR_STABCMN { DTRACE_STABILITY_STABLE, \
        DTRACE_STABILITY_STABLE, DTRACE_CLASS_COMMON }

#define DT_ATTR_EVOLCMN { DTRACE_STABILITY_EVOLVING, \
        DTRACE_STABILITY_EVOLVING, DTRACE_CLASS_COMMON \
}

/*
 * The version number should be increased for every customer visible release
 * of DTrace. The major number should be incremented when a fundamental
 * change has been made that would affect all consumers, and would reflect
 * sweeping changes to DTrace or the D language. The minor number should be
 * incremented when a change is introduced that could break scripts that had
 * previously worked; for example, adding a new built-in variable could break
 * a script which was already using that identifier. The micro number should
 * be changed when introducing functionality changes or major bug fixes that
 * do not affect backward compatibility -- this is merely to make capabilities
 * easily determined from the version number. Minor bugs do not require any
 * modification to the version number.
 */
#define DT_VERS_1_0     DT_VERSION_NUMBER(1, 0, 0)
#define DT_VERS_1_1     DT_VERSION_NUMBER(1, 1, 0)
#define DT_VERS_1_2     DT_VERSION_NUMBER(1, 2, 0)
#define DT_VERS_1_2_1   DT_VERSION_NUMBER(1, 2, 1)
#define DT_VERS_1_2_2   DT_VERSION_NUMBER(1, 2, 2)
#define DT_VERS_1_3     DT_VERSION_NUMBER(1, 3, 0)
#define DT_VERS_1_4     DT_VERSION_NUMBER(1, 4, 0)
#define DT_VERS_1_4_1   DT_VERSION_NUMBER(1, 4, 1)
#define DT_VERS_1_5     DT_VERSION_NUMBER(1, 5, 0)
#define DT_VERS_1_6     DT_VERSION_NUMBER(1, 6, 0)
#define DT_VERS_1_6_1   DT_VERSION_NUMBER(1, 6, 1)
#define DT_VERS_1_6_2   DT_VERSION_NUMBER(1, 6, 2)
#define DT_VERS_1_6_3   DT_VERSION_NUMBER(1, 6, 3)
#define DT_VERS_1_7     DT_VERSION_NUMBER(1, 7, 0)
#define DT_VERS_1_7_1   DT_VERSION_NUMBER(1, 7, 1)
#define DT_VERS_1_8     DT_VERSION_NUMBER(1, 8, 0)
#define DT_VERS_1_8_1   DT_VERSION_NUMBER(1, 8, 1)
#define DT_VERS_1_9     DT_VERSION_NUMBER(1, 9, 0)
#define DT_VERS_1_9_1   DT_VERSION_NUMBER(1, 9, 1)
#define DT_VERS_1_10    DT_VERSION_NUMBER(1, 10, 0)
#define DT_VERS_LATEST  DT_VERS_1_10
#define DT_VERS_STRING  "Sun D 1.10"
#define DT_VERS_LATEST  DT_VERS_1_9_1
#define DT_VERS_STRING  "Sun D 1.9.1"

const dt_version_t _dtrace_versions[] = {
        DT_VERS_1_0,    /* D API 1.0.0 (PSARC 2001/466) Solaris 10 FCS */
        DT_VERS_1_1,    /* D API 1.1.0 Solaris Express 6/05 */
```

```
121         DT_VERS_1_2,    /* D API 1.2.0 Solaris 10 Update 1 */
122         DT_VERS_1_2_1,  /* D API 1.2.1 Solaris Express 4/06 */
123         DT_VERS_1_2_2,  /* D API 1.2.2 Solaris Express 6/06 */
124         DT_VERS_1_3,    /* D API 1.3 Solaris Express 10/06 */
125         DT_VERS_1_4,    /* D API 1.4 Solaris Express 2/07 */
126         DT_VERS_1_4_1,  /* D API 1.4.1 Solaris Express 4/07 */
127         DT_VERS_1_5,    /* D API 1.5 Solaris Express 7/07 */
128         DT_VERS_1_6,    /* D API 1.6 */
129         DT_VERS_1_6_1,  /* D API 1.6.1 */
130         DT_VERS_1_6_2,  /* D API 1.6.2 */
131         DT_VERS_1_6_3,  /* D API 1.6.3 */
132         DT_VERS_1_7,    /* D API 1.7 */
133         DT_VERS_1_7_1,  /* D API 1.7.1 */
134         DT_VERS_1_8,    /* D API 1.8 */
135         DT_VERS_1_8_1,  /* D API 1.8.1 */
136         DT_VERS_1_9,    /* D API 1.9 */
137         DT_VERS_1_9_1,  /* D API 1.9.1 */
138         DT_VERS_1_10,   /* D API 1.10 */
139 #endif /* ! codereview */
140         0
141 };

143 /*
144  * Table of global identifiers.  This is used to populate the global identifier
145  * hash when a new dtrace client open occurs.  For more info see dt_ident.h.
146  * The global identifiers that represent functions use the dt_idops_func ops
147  * and specify the private data pointer as a prototype string which is parsed
148  * when the identifier is first encountered.  These prototypes look like ANSI
149  * C function prototypes except that the special symbol "@" can be used as a
150  * wildcard to represent a single parameter of any type (i.e. any dt_node_t).
151  * The standard "..." notation can also be used to represent varargs.  An empty
152  * parameter list is taken to mean void (that is, no arguments are permitted).
153  * A parameter enclosed in square brackets (e.g. "[int]") denotes an optional
154  * argument.
155  */
156 static const dt_ident_t _dtrace_globals[] = {
157 { "alloca", DT_IDENT_FUNC, 0, DIF_SUBR_ALLOCA, DT_ATTR_STABCMN, DT_VERS_1_0,
158         &dt_idops_func, "void *(size_t)" },
159 { "arg0", DT_IDENT_SCALAR, 0, DIF_VAR_ARG0, DT_ATTR_STABCMN, DT_VERS_1_0,
160         &dt_idops_type, "int64_t" },
161 { "arg1", DT_IDENT_SCALAR, 0, DIF_VAR_ARG1, DT_ATTR_STABCMN, DT_VERS_1_0,
162         &dt_idops_type, "int64_t" },
163 { "arg2", DT_IDENT_SCALAR, 0, DIF_VAR_ARG2, DT_ATTR_STABCMN, DT_VERS_1_0,
164         &dt_idops_type, "int64_t" },
165 { "arg3", DT_IDENT_SCALAR, 0, DIF_VAR_ARG3, DT_ATTR_STABCMN, DT_VERS_1_0,
166         &dt_idops_type, "int64_t" },
167 { "arg4", DT_IDENT_SCALAR, 0, DIF_VAR_ARG4, DT_ATTR_STABCMN, DT_VERS_1_0,
168         &dt_idops_type, "int64_t" },
169 { "arg5", DT_IDENT_SCALAR, 0, DIF_VAR_ARG5, DT_ATTR_STABCMN, DT_VERS_1_0,
170         &dt_idops_type, "int64_t" },
171 { "arg6", DT_IDENT_SCALAR, 0, DIF_VAR_ARG6, DT_ATTR_STABCMN, DT_VERS_1_0,
172         &dt_idops_type, "int64_t" },
173 { "arg7", DT_IDENT_SCALAR, 0, DIF_VAR_ARG7, DT_ATTR_STABCMN, DT_VERS_1_0,
174         &dt_idops_type, "int64_t" },
175 { "arg8", DT_IDENT_SCALAR, 0, DIF_VAR_ARG8, DT_ATTR_STABCMN, DT_VERS_1_0,
176         &dt_idops_type, "int64_t" },
177 { "arg9", DT_IDENT_SCALAR, 0, DIF_VAR_ARG9, DT_ATTR_STABCMN, DT_VERS_1_0,
178         &dt_idops_type, "int64_t" },
179 { "args", DT_IDENT_ARRAY, 0, DIF_VAR_ARGS, DT_ATTR_STABCMN, DT_VERS_1_0,
180         &dt_idops_args, NULL },
181 { "avg", DT_IDENT_AGGFUNC, 0, DTRACEAGG_AVG, DT_ATTR_STABCMN, DT_VERS_1_0,
182         &dt_idops_func, "void(@)" },
183 { "basename", DT_IDENT_FUNC, 0, DIF_SUBR_BASENAME, DT_ATTR_STABCMN, DT_VERS_1_0,
184         &dt_idops_func, "string(const char *)" },
185 { "bcopy", DT_IDENT_FUNC, 0, DIF_SUBR_BCOPY, DT_ATTR_STABCMN, DT_VERS_1_0,
186         &dt_idops_func, "void(void *, void *, size_t)" },
```



```
187 { "breakpoint", DT_IDENT_ACTFUNC, 0, DT_ACT_BREAKPOINT,
188         DT_ATTR_STABCMN, DT_VERS_1_0,
189         &dt_idops_func, "void()" },
190 { "caller", DT_IDENT_SCALAR, 0, DIF_VAR_CALLER, DT_ATTR_STABCMN, DT_VERS_1_0,
191         &dt_idops_type, "uintptr_t" },
192 { "chill", DT_IDENT_ACTFUNC, 0, DT_ACT_CHILL, DT_ATTR_STABCMN, DT_VERS_1_0,
193         &dt_idops_func, "void(int)" },
194 { "cleanpath", DT_IDENT_FUNC, 0, DIF_SUBR_CLEANPATH, DT_ATTR_STABCMN,
195         DT_VERS_1_0, &dt_idops_func, "string(const char *)" },
196 { "clear", DT_IDENT_ACTFUNC, 0, DT_ACT_CLEAR, DT_ATTR_STABCMN, DT_VERS_1_0,
197         &dt_idops_func, "void(...)" },
198 { "commit", DT_IDENT_ACTFUNC, 0, DT_ACT_COMMIT, DT_ATTR_STABCMN, DT_VERS_1_0,
199         &dt_idops_func, "void(int)" },
200 { "copyin", DT_IDENT_FUNC, 0, DIF_SUBR_COPYIN, DT_ATTR_STABCMN, DT_VERS_1_0,
201         &dt_idops_func, "void *(uintptr_t, size_t)" },
202 { "copyinstr", DT_IDENT_FUNC, 0, DIF_SUBR_COPYINSTR,
203         DT_ATTR_STABCMN, DT_VERS_1_0,
204         &dt_idops_func, "string(uintptr_t, [size_t])" },
205 { "copyinto", DT_IDENT_FUNC, 0, DIF_SUBR_COPYINTO, DT_ATTR_STABCMN,
206         DT_VERS_1_0, &dt_idops_func, "void(uintptr_t, size_t, void *)" },
207 { "copyout", DT_IDENT_FUNC, 0, DIF_SUBR_COPYOUT, DT_ATTR_STABCMN, DT_VERS_1_0,
208         &dt_idops_func, "void(void *, uintptr_t, size_t)" },
209 { "copyoutstr", DT_IDENT_FUNC, 0, DIF_SUBR_COPYOUTSTR,
210         DT_ATTR_STABCMN, DT_VERS_1_0,
211         &dt_idops_func, "void(char *, uintptr_t, size_t)" },
212 { "count", DT_IDENT_AGGFUNC, 0, DTRACEAGG_COUNT, DT_ATTR_STABCMN, DT_VERS_1_0,
213         &dt_idops_func, "void()" },
214 { "curthread", DT_IDENT_SCALAR, 0, DIF_VAR_CURTHREAD,
215         { DTRACE_STABILITY_STABLE, DTRACE_STABILITY_PRIVATE,
216         DTRACE_CLASS_COMMON }, DT_VERS_1_0,
217         &dt_idops_type, "genunix`kthread_t *" },
218 { "ddi_pathname", DT_IDENT_FUNC, 0, DIF_SUBR_DDI_PATHNAME,
219         DT_ATTR_EVOLCMN, DT_VERS_1_0,
220         &dt_idops_func, "string(void *, int64_t)" },
221 { "denormalize", DT_IDENT_ACTFUNC, 0, DT_ACT_DENORMALIZE, DT_ATTR_STABCMN,
222         DT_VERS_1_0, &dt_idops_func, "void(...)" },
223 { "dirname", DT_IDENT_FUNC, 0, DIF_SUBR_DIRNAME, DT_ATTR_STABCMN, DT_VERS_1_0,
224         &dt_idops_func, "string(const char *)" },
225 { "discard", DT_IDENT_ACTFUNC, 0, DT_ACT_DISCARD, DT_ATTR_STABCMN, DT_VERS_1_0,
226         &dt_idops_func, "void(int)" },
227 { "epid", DT_IDENT_SCALAR, 0, DIF_VAR_EPID, DT_ATTR_STABCMN, DT_VERS_1_0,
228         &dt_idops_type, "uint_t" },
229 { "errno", DT_IDENT_SCALAR, 0, DIF_VAR_ERRNO, DT_ATTR_STABCMN, DT_VERS_1_0,
230         &dt_idops_type, "int" },
231 { "execname", DT_IDENT_SCALAR, 0, DIF_VAR_EXECNAME,
232         DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
233 { "exit", DT_IDENT_ACTFUNC, 0, DT_ACT_EXIT, DT_ATTR_STABCMN, DT_VERS_1_0,
234         &dt_idops_func, "void(int)" },
235 { "freopen", DT_IDENT_ACTFUNC, 0, DT_ACT_FREOPEN, DT_ATTR_STABCMN,
236         DT_VERS_1_1, &dt_idops_func, "void(@, ...)" },
237 { "ftruncate", DT_IDENT_ACTFUNC, 0, DT_ACT_FTRUNCATE, DT_ATTR_STABCMN,
238         DT_VERS_1_0, &dt_idops_func, "void()" },
239 { "func", DT_IDENT_ACTFUNC, 0, DT_ACT_SYM, DT_ATTR_STABCMN,
240         DT_VERS_1_2, &dt_idops_func, "_symaddr(uintptr_t)" },
241 { "getmajor", DT_IDENT_FUNC, 0, DIF_SUBR_GETMAJOR,
242         DT_ATTR_EVOLCMN, DT_VERS_1_0,
243         &dt_idops_func, "genunix`major_t(genunix`dev_t)" },
244 { "getminor", DT_IDENT_FUNC, 0, DIF_SUBR_GETMINOR,
245         DT_ATTR_EVOLCMN, DT_VERS_1_0,
246         &dt_idops_func, "genunix`minor_t(genunix`dev_t)" },
247 { "htonl", DT_IDENT_FUNC, 0, DIF_SUBR_HTONL, DT_ATTR_EVOLCMN, DT_VERS_1_3,
248         &dt_idops_func, "uint32_t(uint32_t)" },
249 { "htonll", DT_IDENT_FUNC, 0, DIF_SUBR_HTONLL, DT_ATTR_EVOLCMN, DT_VERS_1_3,
250         &dt_idops_func, "uint64_t(uint64_t)" },
251 { "htons", DT_IDENT_FUNC, 0, DIF_SUBR_HTONS, DT_ATTR_EVOLCMN, DT_VERS_1_3,
252         &dt_idops_func, "uint16_t(uint16_t)" },
```

```
253 { "getf", DT_IDENT_FUNC, 0, DIF_SUBR_GETF, DT_ATTR_STABCMN, DT_VERS_1_10,
254         &dt_idops_func, "file_t *(int)" },
255 #endif /* ! codereview */
256 { "gid", DT_IDENT_SCALAR, 0, DIF_VAR_GID, DT_ATTR_STABCMN, DT_VERS_1_0,
257         &dt_idops_type, "gid_t" },
258 { "id", DT_IDENT_SCALAR, 0, DIF_VAR_ID, DT_ATTR_STABCMN, DT_VERS_1_0,
259         &dt_idops_type, "uint_t" },
260 { "index", DT_IDENT_FUNC, 0, DIF_SUBR_INDEX, DT_ATTR_STABCMN, DT_VERS_1_1,
261         &dt_idops_func, "int(const char *, const char *, [int])" },
262 { "inet_ntoa", DT_IDENT_FUNC, 0, DIF_SUBR_INET_NTOA, DT_ATTR_STABCMN,
263         DT_VERS_1_5, &dt_idops_func, "string(ipaddr_t *)" },
264 { "inet_ntoa6", DT_IDENT_FUNC, 0, DIF_SUBR_INET_NTOA6, DT_ATTR_STABCMN,
265         DT_VERS_1_5, &dt_idops_func, "string(in6_addr_t *)" },
266 { "inet_ntop", DT_IDENT_FUNC, 0, DIF_SUBR_INET_NTOP, DT_ATTR_STABCMN,
267         DT_VERS_1_5, &dt_idops_func, "string(int, void *)" },
268 { "ipl", DT_IDENT_SCALAR, 0, DIF_VAR_IPL, DT_ATTR_STABCMN, DT_VERS_1_0,
269         &dt_idops_type, "uint_t" },
270 { "jstack", DT_IDENT_ACTFUNC, 0, DT_ACT_JSTACK, DT_ATTR_STABCMN, DT_VERS_1_0,
271         &dt_idops_func, "stack(...)" },
272 { "lltostr", DT_IDENT_FUNC, 0, DIF_SUBR_LLTOSTR, DT_ATTR_STABCMN, DT_VERS_1_0,
273         &dt_idops_func, "string(int64_t, [int])" },
274 { "llquantize", DT_IDENT_AGGFUNC, 0, DTRACEAGG_LLQUANTIZE, DT_ATTR_STABCMN,
275         DT_VERS_1_7, &dt_idops_func,
276         "void(@, int32_t, int32_t, int32_t, int32_t, ...)" },
277 { "lquantize", DT_IDENT_AGGFUNC, 0, DTRACEAGG_LQUANTIZE,
278         DT_ATTR_STABCMN, DT_VERS_1_0,
279         &dt_idops_func, "void(@, int32_t, int32_t, ...)" },
280 { "max", DT_IDENT_AGGFUNC, 0, DTRACEAGG_MAX, DT_ATTR_STABCMN, DT_VERS_1_0,
281         &dt_idops_func, "void(@)" },
282 { "min", DT_IDENT_AGGFUNC, 0, DTRACEAGG_MIN, DT_ATTR_STABCMN, DT_VERS_1_0,
283         &dt_idops_func, "void(@)" },
284 { "mod", DT_IDENT_ACTFUNC, 0, DT_ACT_MOD, DT_ATTR_STABCMN,
285         DT_VERS_1_2, &dt_idops_func, "_symaddr(uintptr_t)" },
286 { "msgdsize", DT_IDENT_FUNC, 0, DIF_SUBR_MSGDSIZE,
287         DT_ATTR_STABCMN, DT_VERS_1_0,
288         &dt_idops_func, "size_t(mblk_t *)" },
289 { "msgsize", DT_IDENT_FUNC, 0, DIF_SUBR_MSGSIZE,
290         DT_ATTR_STABCMN, DT_VERS_1_0,
291         &dt_idops_func, "size_t(mblk_t *)" },
292 { "mutex_owned", DT_IDENT_FUNC, 0, DIF_SUBR_MUTEX_OWNED,
293         DT_ATTR_EVOLCMN, DT_VERS_1_0,
294         &dt_idops_func, "int(genunix`kmutex_t *)" },
295 { "mutex_owner", DT_IDENT_FUNC, 0, DIF_SUBR_MUTEX_OWNER,
296         DT_ATTR_EVOLCMN, DT_VERS_1_0,
297         &dt_idops_func, "genunix`kthread_t *(genunix`kmutex_t *)" },
298 { "mutex_type_adaptive", DT_IDENT_FUNC, 0, DIF_SUBR_MUTEX_TYPE_ADAPTIVE,
299         DT_ATTR_EVOLCMN, DT_VERS_1_0,
300         &dt_idops_func, "int(genunix`kmutex_t *)" },
301 { "mutex_type_spin", DT_IDENT_FUNC, 0, DIF_SUBR_MUTEX_TYPE_SPIN,
302         DT_ATTR_EVOLCMN, DT_VERS_1_0,
303         &dt_idops_func, "int(genunix`kmutex_t *)" },
304 { "ntohl", DT_IDENT_FUNC, 0, DIF_SUBR_NTOHL, DT_ATTR_EVOLCMN, DT_VERS_1_3,
305         &dt_idops_func, "uint32_t(uint32_t)" },
306 { "ntohll", DT_IDENT_FUNC, 0, DIF_SUBR_NTOHLL, DT_ATTR_EVOLCMN, DT_VERS_1_3,
307         &dt_idops_func, "uint64_t(uint64_t)" },
308 { "ntohs", DT_IDENT_FUNC, 0, DIF_SUBR_NTOHS, DT_ATTR_EVOLCMN, DT_VERS_1_3,
309         &dt_idops_func, "uint16_t(uint16_t)" },
310 { "normalize", DT_IDENT_ACTFUNC, 0, DT_ACT_NORMALIZE, DT_ATTR_STABCMN,
311         DT_VERS_1_0, &dt_idops_func, "void(...)" },
312 { "panic", DT_IDENT_ACTFUNC, 0, DT_ACT_PANIC, DT_ATTR_STABCMN, DT_VERS_1_0,
313         &dt_idops_func, "void()" },
314 { "pid", DT_IDENT_SCALAR, 0, DIF_VAR_PID, DT_ATTR_STABCMN, DT_VERS_1_0,
315         &dt_idops_type, "pid_t" },
316 { "ppid", DT_IDENT_SCALAR, 0, DIF_VAR_PPID, DT_ATTR_STABCMN, DT_VERS_1_0,
317         &dt_idops_type, "pid_t" },
318 { "print", DT_IDENT_ACTFUNC, 0, DT_ACT_PRINT, DT_ATTR_STABCMN, DT_VERS_1_9,
```



```
319         &dt_idops_func, "void(@)" },
320 { "printa", DT_IDENT_ACTFUNC, 0, DT_ACT_PRINTA, DT_ATTR_STABCMN, DT_VERS_1_0,
321         &dt_idops_func, "void(@, ...)" },
322 { "printf", DT_IDENT_ACTFUNC, 0, DT_ACT_PRINTF, DT_ATTR_STABCMN, DT_VERS_1_0,
323         &dt_idops_func, "void(@, ...)" },
324 { "probefunc", DT_IDENT_SCALAR, 0, DIF_VAR_PROBEFUNC,
325         DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
326 { "probemod", DT_IDENT_SCALAR, 0, DIF_VAR_PROBEMOD,
327         DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
328 { "probename", DT_IDENT_SCALAR, 0, DIF_VAR_PROBENAME,
329         DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
330 { "probeprov", DT_IDENT_SCALAR, 0, DIF_VAR_PROBEPROV,
331         DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
332 { "progenyof", DT_IDENT_FUNC, 0, DIF_SUBR_PROGENYOF,
333         DT_ATTR_STABCMN, DT_VERS_1_0,
334         &dt_idops_func, "int(pid_t)" },
335 { "quantize", DT_IDENT_AGGFUNC, 0, DTRACEAGG_QUANTIZE,
336         DT_ATTR_STABCMN, DT_VERS_1_0,
337         &dt_idops_func, "void(@, ...)" },
338 { "raise", DT_IDENT_ACTFUNC, 0, DT_ACT_RAISE, DT_ATTR_STABCMN, DT_VERS_1_0,
339         &dt_idops_func, "void(int)" },
340 { "rand", DT_IDENT_FUNC, 0, DIF_SUBR_RAND, DT_ATTR_STABCMN, DT_VERS_1_0,
341         &dt_idops_func, "int()" },
342 { "rindex", DT_IDENT_FUNC, 0, DIF_SUBR_RINDEX, DT_ATTR_STABCMN, DT_VERS_1_1,
343         &dt_idops_func, "int(const char *, const char *, [int])" },
344 { "rw_iswriter", DT_IDENT_FUNC, 0, DIF_SUBR_RW_ISWRITER,
345         DT_ATTR_EVOLCMN, DT_VERS_1_0,
346         &dt_idops_func, "int(genunix`krwlock_t *)" },
347 { "rw_read_held", DT_IDENT_FUNC, 0, DIF_SUBR_RW_READ_HELD,
348         DT_ATTR_EVOLCMN, DT_VERS_1_0,
349         &dt_idops_func, "int(genunix`krwlock_t *)" },
350 { "rw_write_held", DT_IDENT_FUNC, 0, DIF_SUBR_RW_WRITE_HELD,
351         DT_ATTR_EVOLCMN, DT_VERS_1_0,
352         &dt_idops_func, "int(genunix`krwlock_t *)" },
353 { "self", DT_IDENT_PTR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0,
354         &dt_idops_type, "void" },
355 { "setopt", DT_IDENT_ACTFUNC, 0, DT_ACT_SETOPT, DT_ATTR_STABCMN,
356         DT_VERS_1_2, &dt_idops_func, "void(const char *, [const char *])" },
357 { "speculate", DT_IDENT_ACTFUNC, 0, DT_ACT_SPECULATE,
358         DT_ATTR_STABCMN, DT_VERS_1_0,
359         &dt_idops_func, "void(int)" },
360 { "speculation", DT_IDENT_FUNC, 0, DIF_SUBR_SPECULATION,
361         DT_ATTR_STABCMN, DT_VERS_1_0,
362         &dt_idops_func, "int()" },
363 { "stack", DT_IDENT_ACTFUNC, 0, DT_ACT_STACK, DT_ATTR_STABCMN, DT_VERS_1_0,
364         &dt_idops_func, "stack(...)" },
365 { "stackdepth", DT_IDENT_SCALAR, 0, DIF_VAR_STACKDEPTH,
366         DT_ATTR_STABCMN, DT_VERS_1_0,
367         &dt_idops_type, "uint32_t" },
368 { "stddev", DT_IDENT_AGGFUNC, 0, DTRACEAGG_STDDEV, DT_ATTR_STABCMN,
369         DT_VERS_1_6, &dt_idops_func, "void(@)" },
370 { "stop", DT_IDENT_ACTFUNC, 0, DT_ACT_STOP, DT_ATTR_STABCMN, DT_VERS_1_0,
371         &dt_idops_func, "void()" },
372 { "strchr", DT_IDENT_FUNC, 0, DIF_SUBR_STRCHR, DT_ATTR_STABCMN, DT_VERS_1_1,
373         &dt_idops_func, "string(const char *, char)" },
374 { "strlen", DT_IDENT_FUNC, 0, DIF_SUBR_STRLEN, DT_ATTR_STABCMN, DT_VERS_1_0,
375         &dt_idops_func, "size_t(const char *)" },
376 { "strjoin", DT_IDENT_FUNC, 0, DIF_SUBR_STRJOIN, DT_ATTR_STABCMN, DT_VERS_1_0,
377         &dt_idops_func, "string(const char *, const char *)" },
378 { "strrchr", DT_IDENT_FUNC, 0, DIF_SUBR_STRRCHR, DT_ATTR_STABCMN, DT_VERS_1_1,
379         &dt_idops_func, "string(const char *, char)" },
380 { "strstr", DT_IDENT_FUNC, 0, DIF_SUBR_STRSTR, DT_ATTR_STABCMN, DT_VERS_1_1,
381         &dt_idops_func, "string(const char *, const char *)" },
382 { "strtok", DT_IDENT_FUNC, 0, DIF_SUBR_STRTOK, DT_ATTR_STABCMN, DT_VERS_1_1,
383         &dt_idops_func, "string(const char *, const char *)" },
384 { "substr", DT_IDENT_FUNC, 0, DIF_SUBR_SUBSTR, DT_ATTR_STABCMN, DT_VERS_1_1,
```

```
385         &dt_idops_func, "string(const char *, int, [int])" },
386 { "sum", DT_IDENT_AGGFUNC, 0, DTRACEAGG_SUM, DT_ATTR_STABCMN, DT_VERS_1_0,
387         &dt_idops_func, "void(@)" },
388 { "sym", DT_IDENT_ACTFUNC, 0, DT_ACT_SYM, DT_ATTR_STABCMN,
389         DT_VERS_1_2, &dt_idops_func, "_symaddr(uintptr_t)" },
390 { "system", DT_IDENT_ACTFUNC, 0, DT_ACT_SYSTEM, DT_ATTR_STABCMN, DT_VERS_1_0,
391         &dt_idops_func, "void(@, ...)" },
392 { "this", DT_IDENT_PTR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0,
393         &dt_idops_type, "void" },
394 { "tid", DT_IDENT_SCALAR, 0, DIF_VAR_TID, DT_ATTR_STABCMN, DT_VERS_1_0,
395         &dt_idops_type, "id_t" },
396 { "timestamp", DT_IDENT_SCALAR, 0, DIF_VAR_TIMESTAMP,
397         DT_ATTR_STABCMN, DT_VERS_1_0,
398         &dt_idops_type, "uint64_t" },
399 { "tolower", DT_IDENT_FUNC, 0, DIF_SUBR_TOLOWER, DT_ATTR_STABCMN, DT_VERS_1_8,
400         &dt_idops_func, "string(const char *)" },
401 { "toupper", DT_IDENT_FUNC, 0, DIF_SUBR_TOUPPER, DT_ATTR_STABCMN, DT_VERS_1_8,
402         &dt_idops_func, "string(const char *)" },
403 { "trace", DT_IDENT_ACTFUNC, 0, DT_ACT_TRACE, DT_ATTR_STABCMN, DT_VERS_1_0,
404         &dt_idops_func, "void(@)" },
405 { "tracemem", DT_IDENT_ACTFUNC, 0, DT_ACT_TRACEMEM,
406         DT_ATTR_STABCMN, DT_VERS_1_0,
407         &dt_idops_func, "void(@, size_t, ...)" },
408 { "trunc", DT_IDENT_ACTFUNC, 0, DT_ACT_TRUNC, DT_ATTR_STABCMN,
409         DT_VERS_1_0, &dt_idops_func, "void(...)" },
410 { "uaddr", DT_IDENT_ACTFUNC, 0, DT_ACT_UADDR, DT_ATTR_STABCMN,
411         DT_VERS_1_2, &dt_idops_func, "_usymaddr(uintptr_t)" },
412 { "ucaller", DT_IDENT_SCALAR, 0, DIF_VAR_UCALLER, DT_ATTR_STABCMN,
413         DT_VERS_1_2, &dt_idops_type, "uint64_t" },
414 { "ufunc", DT_IDENT_ACTFUNC, 0, DT_ACT_USYM, DT_ATTR_STABCMN,
415         DT_VERS_1_2, &dt_idops_func, "_usymaddr(uintptr_t)" },
416 { "uid", DT_IDENT_SCALAR, 0, DIF_VAR_UID, DT_ATTR_STABCMN, DT_VERS_1_0,
417         &dt_idops_type, "uid_t" },
418 { "umod", DT_IDENT_ACTFUNC, 0, DT_ACT_UMOD, DT_ATTR_STABCMN,
419         DT_VERS_1_2, &dt_idops_func, "_usymaddr(uintptr_t)" },
420 { "uregs", DT_IDENT_ARRAY, 0, DIF_VAR_UREGS, DT_ATTR_STABCMN, DT_VERS_1_0,
421         &dt_idops_regs, NULL },
422 { "ustack", DT_IDENT_ACTFUNC, 0, DT_ACT_USTACK, DT_ATTR_STABCMN, DT_VERS_1_0,
423         &dt_idops_func, "stack(...)" },
424 { "ustackdepth", DT_IDENT_SCALAR, 0, DIF_VAR_USTACKDEPTH,
425         DT_ATTR_STABCMN, DT_VERS_1_2,
426         &dt_idops_type, "uint32_t" },
427 { "usym", DT_IDENT_ACTFUNC, 0, DT_ACT_USYM, DT_ATTR_STABCMN,
428         DT_VERS_1_2, &dt_idops_func, "_usymaddr(uintptr_t)" },
429 { "vmregs", DT_IDENT_ARRAY, 0, DIF_VAR_VMREGS, DT_ATTR_STABCMN, DT_VERS_1_7,
430         &dt_idops_regs, NULL },
431 { "vtimestamp", DT_IDENT_SCALAR, 0, DIF_VAR_VTIMESTAMP,
432         DT_ATTR_STABCMN, DT_VERS_1_0,
433         &dt_idops_type, "uint64_t" },
434 { "walltimestamp", DT_IDENT_SCALAR, 0, DIF_VAR_WALLTIMESTAMP,
435         DT_ATTR_STABCMN, DT_VERS_1_0,
436         &dt_idops_type, "int64_t" },
437 { "zonename", DT_IDENT_SCALAR, 0, DIF_VAR_ZONENAME,
438         DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
439 { NULL, 0, 0, 0, { 0, 0, 0 }, 0, NULL, NULL }
440 };

442 /*
443  * Tables of ILP32 intrinsic integer and floating-point type templates to use
444  * to populate the dynamic "C" CTF type container.
445  */
446 static const dt_intrinsic_t _dtrace_intrinsics_32[] = {
447 { "void", { CTF_INT_SIGNED, 0, 0 }, CTF_K_INTEGER },
448 { "signed", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
449 { "unsigned", { 0, 0, 32 }, CTF_K_INTEGER },
450 { "char", { CTF_INT_SIGNED | CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
```



```
451 { "short", { CTF_INT_SIGNED, 0, 16 }, CTF_K_INTEGER },
452 { "int", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
453 { "long", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
454 { "long long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
455 { "signed char", { CTF_INT_SIGNED | CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
456 { "signed short", { CTF_INT_SIGNED, 0, 16 }, CTF_K_INTEGER },
457 { "signed int", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
458 { "signed long", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
459 { "signed long long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
460 { "unsigned char", { CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
461 { "unsigned short", { 0, 0, 16 }, CTF_K_INTEGER },
462 { "unsigned int", { 0, 0, 32 }, CTF_K_INTEGER },
463 { "unsigned long", { 0, 0, 32 }, CTF_K_INTEGER },
464 { "unsigned long long", { 0, 0, 64 }, CTF_K_INTEGER },
465 { "_Bool", { CTF_INT_BOOL, 0, 8 }, CTF_K_INTEGER },
466 { "float", { CTF_FP_SINGLE, 0, 32 }, CTF_K_FLOAT },
467 { "double", { CTF_FP_DOUBLE, 0, 64 }, CTF_K_FLOAT },
468 { "long double", { CTF_FP_LDOUBLE, 0, 128 }, CTF_K_FLOAT },
469 { "float imaginary", { CTF_FP_IMAGRY, 0, 32 }, CTF_K_FLOAT },
470 { "double imaginary", { CTF_FP_DIMAGRY, 0, 64 }, CTF_K_FLOAT },
471 { "long double imaginary", { CTF_FP_LDIMAGRY, 0, 128 }, CTF_K_FLOAT },
472 { "float complex", { CTF_FP_CPLX, 0, 64 }, CTF_K_FLOAT },
473 { "double complex", { CTF_FP_DCPLX, 0, 128 }, CTF_K_FLOAT },
474 { "long double complex", { CTF_FP_LDCPLX, 0, 256 }, CTF_K_FLOAT },
475 { NULL, { 0, 0, 0 }, 0 }
476 };

478 /*
479  * Tables of LP64 intrinsic integer and floating-point type templates to use
480  * to populate the dynamic "C" CTF type container.
481  */
482 static const dt_intrinsic_t _dtrace_intrinsics_64[] = {
483 { "void", { CTF_INT_SIGNED, 0, 0 }, CTF_K_INTEGER },
484 { "signed", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
485 { "unsigned", { 0, 0, 32 }, CTF_K_INTEGER },
486 { "char", { CTF_INT_SIGNED | CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
487 { "short", { CTF_INT_SIGNED, 0, 16 }, CTF_K_INTEGER },
488 { "int", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
489 { "long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
490 { "long long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
491 { "signed char", { CTF_INT_SIGNED | CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
492 { "signed short", { CTF_INT_SIGNED, 0, 16 }, CTF_K_INTEGER },
493 { "signed int", { CTF_INT_SIGNED, 0, 32 }, CTF_K_INTEGER },
494 { "signed long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
495 { "signed long long", { CTF_INT_SIGNED, 0, 64 }, CTF_K_INTEGER },
496 { "unsigned char", { CTF_INT_CHAR, 0, 8 }, CTF_K_INTEGER },
497 { "unsigned short", { 0, 0, 16 }, CTF_K_INTEGER },
498 { "unsigned int", { 0, 0, 32 }, CTF_K_INTEGER },
499 { "unsigned long", { 0, 0, 64 }, CTF_K_INTEGER },
500 { "unsigned long long", { 0, 0, 64 }, CTF_K_INTEGER },
501 { "_Bool", { CTF_INT_BOOL, 0, 8 }, CTF_K_INTEGER },
502 { "float", { CTF_FP_SINGLE, 0, 32 }, CTF_K_FLOAT },
503 { "double", { CTF_FP_DOUBLE, 0, 64 }, CTF_K_FLOAT },
504 { "long double", { CTF_FP_LDOUBLE, 0, 128 }, CTF_K_FLOAT },
505 { "float imaginary", { CTF_FP_IMAGRY, 0, 32 }, CTF_K_FLOAT },
506 { "double imaginary", { CTF_FP_DIMAGRY, 0, 64 }, CTF_K_FLOAT },
507 { "long double imaginary", { CTF_FP_LDIMAGRY, 0, 128 }, CTF_K_FLOAT },
508 { "float complex", { CTF_FP_CPLX, 0, 64 }, CTF_K_FLOAT },
509 { "double complex", { CTF_FP_DCPLX, 0, 128 }, CTF_K_FLOAT },
510 { "long double complex", { CTF_FP_LDCPLX, 0, 256 }, CTF_K_FLOAT },
511 { NULL, { 0, 0, 0 }, 0 }
512 };

514 /*
515  * Tables of ILP32 typedefs to use to populate the dynamic "D" CTF container.
516  * These aliases ensure that D definitions can use typical <sys/types.h> names.
```

```
 */
static const dt_typedef_t _dtrace_typedefs_32[] = {
{ "char", "int8_t" },
{ "short", "int16_t" },
{ "int", "int32_t" },
{ "long long", "int64_t" },
{ "int", "intptr_t" },
{ "int", "ssize_t" },
{ "unsigned char", "uint8_t" },
{ "unsigned short", "uint16_t" },
{ "unsigned", "uint32_t" },
{ "unsigned long long", "uint64_t" },
{ "unsigned char", "uchar_t" },
{ "unsigned short", "ushort_t" },
{ "unsigned", "uint_t" },
{ "unsigned long", "ulong_t" },
{ "unsigned long long", "u_longlong_t" },
{ "int", "ptrdiff_t" },
{ "unsigned", "uintptr_t" },
{ "unsigned", "size_t" },
{ "long", "id_t" },
{ "long", "pid_t" },
{ NULL, NULL }
};

/*
 * Tables of LP64 typedefs to use to populate the dynamic "D" CTF container.
 * These aliases ensure that D definitions can use typical <sys/types.h> names.
 */
static const dt_typedef_t _dtrace_typedefs_64[] = {
{ "char", "int8_t" },
{ "short", "int16_t" },
{ "int", "int32_t" },
{ "long", "int64_t" },
{ "long", "intptr_t" },
{ "long", "ssize_t" },
{ "unsigned char", "uint8_t" },
{ "unsigned short", "uint16_t" },
{ "unsigned", "uint32_t" },
{ "unsigned long", "uint64_t" },
{ "unsigned char", "uchar_t" },
{ "unsigned short", "ushort_t" },
{ "unsigned", "uint_t" },
{ "unsigned long", "ulong_t" },
{ "unsigned long long", "u_longlong_t" },
{ "long", "ptrdiff_t" },
{ "unsigned long", "uintptr_t" },
{ "unsigned long", "size_t" },
{ "int", "id_t" },
{ "int", "pid_t" },
{ NULL, NULL }
};

/*
 * Tables of ILP32 integer type templates used to populate the dtp->dt_ints[]
 * cache when a new dtrace client open occurs.  Values are set by dtrace_open().
 */
static const dt_intdesc_t _dtrace_ints_32[] = {
{ "int", NULL, CTF_ERR, 0x7fffffffULL },
{ "unsigned int", NULL, CTF_ERR, 0xffffffffULL },
{ "long", NULL, CTF_ERR, 0x7fffffffULL },
{ "unsigned long", NULL, CTF_ERR, 0xffffffffULL },
{ "long long", NULL, CTF_ERR, 0x7fffffffffffffffULL },
{ "unsigned long long", NULL, CTF_ERR, 0xffffffffffffffffULL }
};
```



```
/*
 * Tables of LP64 integer type templates used to populate the dtp->dt_ints[]
 * cache when a new dtrace client open occurs.  Values are set by dtrace_open().
 */
static const dt_intdesc_t _dtrace_ints_64[] = {
{ "int", NULL, CTF_ERR, 0x7fffffffULL },
{ "unsigned int", NULL, CTF_ERR, 0xffffffffULL },
{ "long", NULL, CTF_ERR, 0x7fffffffffffffffULL },
{ "unsigned long", NULL, CTF_ERR, 0xffffffffffffffffULL },
{ "long long", NULL, CTF_ERR, 0x7fffffffffffffffULL },
{ "unsigned long long", NULL, CTF_ERR, 0xffffffffffffffffULL }
};

/*
 * Table of macro variable templates used to populate the macro identifier hash
 * when a new dtrace client open occurs.  Values are set by dtrace_update().
 */
static const dt_ident_t _dtrace_macros[] = {
{ "egid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
{ "euid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
{ "gid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
{ "pid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
{ "pgid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
{ "ppid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
{ "projid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
{ "sid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
{ "taskid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
{ "target", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
{ "uid", DT_IDENT_SCALAR, 0, 0, DT_ATTR_STABCMN, DT_VERS_1_0 },
{ NULL, 0, 0, 0, { 0, 0, 0 }, 0 }
};

/*
 * Hard-wired definition string to be compiled and cached every time a new
 * DTrace library handle is initialized.  This string should only be used to
 * contain definitions that should be present regardless of DTRACE_O_NOLIBS.
 */
static const char _dtrace_hardwire[] = "\
inline long NULL = 0; \n\
#pragma D binding \"1.0\" NULL\n\
";

/*
 * Default DTrace configuration to use when opening libdtrace DTRACE_O_NODEV.
 * If DTRACE_O_NODEV is not set, we load the configuration from the kernel.
 * The use of CTF_MODEL_NATIVE is more subtle than it might appear: we are
 * relying on the fact that when running dtrace(1M), isaexec will invoke the
 * binary with the same bitness as the kernel, which is what we want by default
 * when generating our DIF.  The user can override the choice using oflags.
 */
static const dtrace_conf_t _dtrace_conf = {
        DIF_VERSION,            /* dtc_difversion */
        DIF_DIR_NREGS,          /* dtc_difintregs */
        DIF_DTR_NREGS,          /* dtc_diftupregs */
        CTF_MODEL_NATIVE        /* dtc_ctfmodel */
};

const dtrace_attribute_t _dtrace_maxattr = {
        DTRACE_STABILITY_MAX,
        DTRACE_STABILITY_MAX,
        DTRACE_CLASS_MAX
};

const dtrace_attribute_t _dtrace_defattr = {
        DTRACE_STABILITY_STABLE,
        DTRACE_STABILITY_STABLE,
```

```
649         DTRACE_CLASS_COMMON
650 };

652 const dtrace_attribute_t _dtrace_symattr = {
653         DTRACE_STABILITY_PRIVATE,
654         DTRACE_STABILITY_PRIVATE,
655         DTRACE_CLASS_UNKNOWN
656 };

658 const dtrace_attribute_t _dtrace_typattr = {
659         DTRACE_STABILITY_PRIVATE,
660         DTRACE_STABILITY_PRIVATE,
661         DTRACE_CLASS_UNKNOWN
662 };

664 const dtrace_attribute_t _dtrace_prvattr = {
665         DTRACE_STABILITY_PRIVATE,
666         DTRACE_STABILITY_PRIVATE,
667         DTRACE_CLASS_UNKNOWN
668 };

670 const dtrace_pattr_t _dtrace_prvdesc = {
671 { DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_COMMON },
672 { DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_COMMON },
673 { DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_COMMON },
674 { DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_COMMON },
675 { DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_COMMON },
676 };

678 const char *_dtrace_defcpp = "/usr/ccs/lib/cpp"; /* default cpp(1) to invoke */
679 const char *_dtrace_defld = "/usr/ccs/bin/ld";   /* default ld(1) to invoke */

681 const char *_dtrace_libdir = "/usr/lib/dtrace"; /* default library directory */
682 const char *_dtrace_provdir = "/dev/dtrace/provider"; /* provider directory */

684 int _dtrace_strbuckets = 211;   /* default number of hash buckets (prime) */
685 int _dtrace_intbuckets = 256;   /* default number of integer buckets (Pof2) */
686 uint_t _dtrace_strsize = 256;   /* default size of string intrinsic type */
687 uint_t _dtrace_stkindent = 14;  /* default whitespace indent for stack/ustack */
688 uint_t _dtrace_pidbuckets = 64; /* default number of pid hash buckets */
689 uint_t _dtrace_pidlrulim = 8;   /* default number of pid handles to cache */
690 size_t _dtrace_bufsize = 512;   /* default dt_buf_create() size */
691 int _dtrace_argmax = 32;        /* default maximum number of probe arguments */

693 int _dtrace_debug = 0;          /* debug messages enabled (off) */
694 const char *const _dtrace_version = DT_VERS_STRING; /* API version string */
695 int _dtrace_rdvers = RD_VERSION; /* rtld_db feature version */

697 typedef struct dt_fdlist {
698         int *df_fds;            /* array of provider driver file descriptors */
699         uint_t df_ents;         /* number of valid elements in df_fds[] */
700         uint_t df_size;         /* size of df_fds[] */
701 } dt_fdlist_t;

703 #pragma init(_dtrace_init)
704 void
705 _dtrace_init(void)
706 {
707         _dtrace_debug = getenv("DTRACE_DEBUG") != NULL;

709         for (; _dtrace_rdvers > 0; _dtrace_rdvers--) {
710                 if (rd_init(_dtrace_rdvers) == RD_OK)
711                         break;
712         }
713 }
```



```
715 static dtrace_hdl_t *
716 set_open_errno(dtrace_hdl_t *dtp, int *errp, int err)
717 {
718         if (dtp != NULL)
719                 dtrace_close(dtp);
720         if (errp != NULL)
721                 *errp = err;
722         return (NULL);
723 }

725 static void
726 dt_provmod_open(dt_provmod_t **provmod, dt_fdlist_t *dfp)
727 {
728         dt_provmod_t *prov;
729         char path[PATH_MAX];
730         struct dirent *dp, *ep;
731         DIR *dirp;
732         int fd;

734         if ((dirp = opendir(_dtrace_provdir)) == NULL)
735                 return; /* failed to open directory; just skip it */

737         ep = alloca(sizeof (struct dirent) + PATH_MAX + 1);
738         bzero(ep, sizeof (struct dirent) + PATH_MAX + 1);

740         while (readdir_r(dirp, ep, &dp) == 0 && dp != NULL) {
741                 if (dp->d_name[0] == '.')
742                         continue; /* skip "." and ".." */

744                 if (dfp->df_ents == dfp->df_size) {
745                         uint_t size = dfp->df_size ? dfp->df_size * 2 : 16;
746                         int *fds = realloc(dfp->df_fds, size * sizeof (int));

748                         if (fds == NULL)
749                                 break; /* skip the rest of this directory */

751                         dfp->df_fds = fds;
752                         dfp->df_size = size;
753                 }

755                 (void) snprintf(path, sizeof (path), "%s/%s",
756                     _dtrace_provdir, dp->d_name);

758                 if ((fd = open(path, O_RDONLY)) == -1)
759                         continue; /* failed to open driver; just skip it */

761                 if (((prov = malloc(sizeof (dt_provmod_t))) == NULL) ||
762                     (prov->dp_name = malloc(strlen(dp->d_name) + 1)) == NULL) {
763                         free(prov);
764                         (void) close(fd);
765                         break;
766                 }

768                 (void) strcpy(prov->dp_name, dp->d_name);
769                 prov->dp_next = *provmod;
770                 *provmod = prov;

772                 dt_dprintf("opened provider %s\n", dp->d_name);
773                 dfp->df_fds[dfp->df_ents++] = fd;
774         }

776         (void) closedir(dirp);
777 }

779 static void
780 dt_provmod_destroy(dt_provmod_t **provmod)
```

```
781 {
782         dt_provmod_t *next, *current;

784         for (current = *provmod; current != NULL; current = next) {
785                 next = current->dp_next;
786                 free(current->dp_name);
787                 free(current);
788         }

790         *provmod = NULL;
791 }

793 static const char *
794 dt_get_sysinfo(int cmd, char *buf, size_t len)
795 {
796         ssize_t rv = sysinfo(cmd, buf, len);
797         char *p = buf;

799         if (rv < 0 || rv > len)
800                 (void) snprintf(buf, len, "%s", "Unknown");

802         while ((p = strchr(p, '.')) != NULL)
803                 *p++ = '_';

805         return (buf);
806 }

808 static dtrace_hdl_t *
809 dt_vopen(int version, int flags, int *errp,
810     const dtrace_vector_t *vector, void *arg)
811 {
812         dtrace_hdl_t *dtp = NULL;
813         int dtfd = -1, ftfd = -1, fterr = 0;
814         dtrace_prog_t *pgp;
815         dt_module_t *dmp;
816         dt_provmod_t *provmod = NULL;
817         int i, err;
818         struct rlimit rl;

820         const dt_intrinsic_t *dinp;
821         const dt_typedef_t *dtyp;
822         const dt_ident_t *idp;

824         dtrace_typeinfo_t dtt;
825         ctf_funcinfo_t ctc;
826         ctf_arinfo_t ctr;

828         dt_fdlist_t df = { NULL, 0, 0 };

830         char isadef[32], utsdef[32];
831         char s1[64], s2[64];

833         if (version <= 0)
834                 return (set_open_errno(dtp, errp, EINVAL));

836         if (version > DTRACE_VERSION)
837                 return (set_open_errno(dtp, errp, EDT_VERSION));

839         if (version < DTRACE_VERSION) {
840                 /*
841                  * Currently, increasing the library version number is used to
842                  * denote a binary incompatible change.  That is, a consumer
843                  * of the library cannot run on a version of the library with
844                  * a higher DTRACE_VERSION number than the consumer compiled
845                  * against.  Once the library API has been committed to,
846                  * backwards binary compatibility will be required; at that
```



```
847                  * time, this check should change to return EDT_OVERSION only
848                  * if the specified version number is less than the version
849                  * number at the time of interface commitment.
850                  */
851                 return (set_open_errno(dtp, errp, EDT_OVERSION));
852         }

854         if (flags & ~DTRACE_O_MASK)
855                 return (set_open_errno(dtp, errp, EINVAL));

857         if ((flags & DTRACE_O_LP64) && (flags & DTRACE_O_ILP32))
858                 return (set_open_errno(dtp, errp, EINVAL));

860         if (vector == NULL && arg != NULL)
861                 return (set_open_errno(dtp, errp, EINVAL));

863         if (elf_version(EV_CURRENT) == EV_NONE)
864                 return (set_open_errno(dtp, errp, EDT_ELFVERSION));

866         if (vector != NULL || (flags & DTRACE_O_NODEV))
867                 goto alloc; /* do not attempt to open dtrace device */

869         /*
870          * Before we get going, crank our limit on file descriptors up to the
871          * hard limit.  This is to allow for the fact that libproc keeps file
872          * descriptors to objects open for the lifetime of the proc handle;
873          * without raising our hard limit, we would have an acceptably small
874          * bound on the number of processes that we could concurrently
875          * instrument with the pid provider.
876          */
877         if (getrlimit(RLIMIT_NOFILE, &rl) == 0) {
878                 rl.rlim_cur = rl.rlim_max;
879                 (void) setrlimit(RLIMIT_NOFILE, &rl);
880         }

882         /*
883          * Get the device path of each of the providers.  We hold them open
884          * in the df.df_fds list until we open the DTrace driver itself,
885          * allowing us to see all of the probes provided on this system.  Once
886          * we have the DTrace driver open, we can safely close all the providers
887          * now that they have registered with the framework.
888          */
889         dt_provmod_open(&provmod, &df);

891         dtfd = open("/dev/dtrace/dtrace", O_RDWR);
892         err = errno; /* save errno from opening dtfd */

894         ftfd = open("/dev/dtrace/provider/fasttrap", O_RDWR);
895         fterr = ftfd == -1 ? errno : 0; /* save errno from open ftfd */

897         while (df.df_ents-- != 0)
898                 (void) close(df.df_fds[df.df_ents]);

900         free(df.df_fds);

902         /*
903          * If we failed to open the dtrace device, fail dtrace_open().
904          * We convert some kernel errnos to custom libdtrace errnos to
905          * improve the resulting message from the usual strerror().
906          */
907         if (dtfd == -1) {
908                 dt_provmod_destroy(&provmod);
909                 switch (err) {
910                 case ENOENT:
911                         err = EDT_NOENT;
912                         break;
```

```
913                 case EBUSY:
914                         err = EDT_BUSY;
915                         break;
916                 case EACCES:
917                         err = EDT_ACCESS;
918                         break;
919                 }
920                 return (set_open_errno(dtp, errp, err));
921         }

923         (void) fcntl(dtfd, F_SETFD, FD_CLOEXEC);
924         (void) fcntl(ftfd, F_SETFD, FD_CLOEXEC);

926 alloc:
927         if ((dtp = malloc(sizeof (dtrace_hdl_t))) == NULL)
928                 return (set_open_errno(dtp, errp, EDT_NOMEM));

930         bzero(dtp, sizeof (dtrace_hdl_t));
931         dtp->dt_oflags = flags;
932         dtp->dt_prcmode = DT_PROC_STOP_PREINIT;
933         dtp->dt_linkmode = DT_LINK_KERNEL;
934         dtp->dt_linktype = DT_LTYP_ELF;
935         dtp->dt_xlatemode = DT_XL_STATIC;
936         dtp->dt_stdcmode = DT_STDC_XA;
937         dtp->dt_version = version;
938         dtp->dt_fd = dtfd;
939         dtp->dt_ftfd = ftfd;
940         dtp->dt_fterr = fterr;
941         dtp->dt_cdefs_fd = -1;
942         dtp->dt_ddefs_fd = -1;
943         dtp->dt_stdout_fd = -1;
944         dtp->dt_modbuckets = _dtrace_strbuckets;
945         dtp->dt_mods = calloc(dtp->dt_modbuckets, sizeof (dt_module_t *));
946         dtp->dt_provbuckets = _dtrace_strbuckets;
947         dtp->dt_provs = calloc(dtp->dt_provbuckets, sizeof (dt_provider_t *));
948         dt_proc_init(dtp);
949         dtp->dt_vmax = DT_VERS_LATEST;
950         dtp->dt_cpp_path = strdup(_dtrace_defcpp);
951         dtp->dt_cpp_argv = malloc(sizeof (char *));
952         dtp->dt_cpp_argc = 1;
953         dtp->dt_cpp_args = 1;
954         dtp->dt_ld_path = strdup(_dtrace_defld);
955         dtp->dt_provmod = provmod;
956         dtp->dt_vector = vector;
957         dtp->dt_varg = arg;
958         dt_dof_init(dtp);
959         (void) uname(&dtp->dt_uts);

961         if (dtp->dt_mods == NULL || dtp->dt_provs == NULL ||
962             dtp->dt_procs == NULL || dtp->dt_proc_env == NULL ||
963             dtp->dt_ld_path == NULL || dtp->dt_cpp_path == NULL ||
964             dtp->dt_cpp_argv == NULL)
965                 return (set_open_errno(dtp, errp, EDT_NOMEM));

967         for (i = 0; i < DTRACEOPT_MAX; i++)
968                 dtp->dt_options[i] = DTRACEOPT_UNSET;

970         dtp->dt_cpp_argv[0] = (char *)strbasename(dtp->dt_cpp_path);

972         (void) snprintf(isadef, sizeof (isadef), "-D__SUNW_D_%u",
973             (uint_t)(sizeof (void *) * NBBY));

975         (void) snprintf(utsdef, sizeof (utsdef), "-D__%s_%s",
976             dt_get_sysinfo(SI_SYSNAME, s1, sizeof (s1)),
977             dt_get_sysinfo(SI_RELEASE, s2, sizeof (s2)));
```



```
979         if (dt_cpp_add_arg(dtp, "-D__sun") == NULL ||
980             dt_cpp_add_arg(dtp, "-D__unix") == NULL ||
981             dt_cpp_add_arg(dtp, "-D__SVR4") == NULL ||
982             dt_cpp_add_arg(dtp, "-D__SUNW_D=1") == NULL ||
983             dt_cpp_add_arg(dtp, isadef) == NULL ||
984             dt_cpp_add_arg(dtp, utsdef) == NULL)
985                 return (set_open_errno(dtp, errp, EDT_NOMEM));

987         if (flags & DTRACE_O_NODEV)
988                 bcopy(&_dtrace_conf, &dtp->dt_conf, sizeof (_dtrace_conf));
989         else if (dt_ioctl(dtp, DTRACEIOC_CONF, &dtp->dt_conf) != 0)
990                 return (set_open_errno(dtp, errp, errno));

992         if (flags & DTRACE_O_LP64)
993                 dtp->dt_conf.dtc_ctfmodel = CTF_MODEL_LP64;
994         else if (flags & DTRACE_O_ILP32)
995                 dtp->dt_conf.dtc_ctfmodel = CTF_MODEL_ILP32;

997 #ifdef __sparc
998         /*
999          * On SPARC systems, __sparc is always defined for <sys/isa_defs.h>
1000         * and __sparcv9 is defined if we are doing a 64-bit compile.
1001         */
1002        if (dt_cpp_add_arg(dtp, "-D__sparc") == NULL)
1003                return (set_open_errno(dtp, errp, EDT_NOMEM));

1005        if (dtp->dt_conf.dtc_ctfmodel == CTF_MODEL_LP64 &&
1006            dt_cpp_add_arg(dtp, "-D__sparcv9") == NULL)
1007                return (set_open_errno(dtp, errp, EDT_NOMEM));
1008 #endif

1010 #ifdef __x86
1011        /*
1012         * On x86 systems, __i386 is defined for <sys/isa_defs.h> for 32-bit
1013         * compiles and __amd64 is defined for 64-bit compiles.  Unlike SPARC,
1014         * they are defined exclusive of one another (see PSARC 2004/619).
1015         */
1016        if (dtp->dt_conf.dtc_ctfmodel == CTF_MODEL_LP64) {
1017                if (dt_cpp_add_arg(dtp, "-D__amd64") == NULL)
1018                        return (set_open_errno(dtp, errp, EDT_NOMEM));
1019        } else {
1020                if (dt_cpp_add_arg(dtp, "-D__i386") == NULL)
1021                        return (set_open_errno(dtp, errp, EDT_NOMEM));
1022        }
1023 #endif

1025        if (dtp->dt_conf.dtc_difversion < DIF_VERSION)
1026                return (set_open_errno(dtp, errp, EDT_DIFVERS));

1028        if (dtp->dt_conf.dtc_ctfmodel == CTF_MODEL_ILP32)
1029                bcopy(_dtrace_ints_32, dtp->dt_ints, sizeof (_dtrace_ints_32));
1030        else
1031                bcopy(_dtrace_ints_64, dtp->dt_ints, sizeof (_dtrace_ints_64));

1033        dtp->dt_macros = dt_idhash_create("macro", NULL, 0, UINT_MAX);
1034        dtp->dt_aggs = dt_idhash_create("aggregation", NULL,
1035            DTRACE_AGGVARIDNONE + 1, UINT_MAX);

1037        dtp->dt_globals = dt_idhash_create("global", _dtrace_globals,
1038            DIF_VAR_OTHER_UBASE, DIF_VAR_OTHER_MAX);

1040        dtp->dt_tls = dt_idhash_create("thread local", NULL,
1041            DIF_VAR_OTHER_UBASE, DIF_VAR_OTHER_MAX);

1043        if (dtp->dt_macros == NULL || dtp->dt_aggs == NULL ||
1044            dtp->dt_globals == NULL || dtp->dt_tls == NULL)
```

```
1045                return (set_open_errno(dtp, errp, EDT_NOMEM));

1047        /*
1048         * Populate the dt_macros identifier hash table by hand: we can't use
1049         * the dt_idhash_populate() mechanism because we're not yet compiling
1050         * and dtrace_update() needs to immediately reference these idents.
1051         */
1052        for (idp = _dtrace_macros; idp->di_name != NULL; idp++) {
1053                if (dt_idhash_insert(dtp->dt_macros, idp->di_name,
1054                    idp->di_kind, idp->di_flags, idp->di_id, idp->di_attr,
1055                    idp->di_vers, idp->di_ops ? idp->di_ops : &dt_idops_thaw,
1056                    idp->di_iarg, 0) == NULL)
1057                        return (set_open_errno(dtp, errp, EDT_NOMEM));
1058        }

1060        /*
1061         * Update the module list using /system/object and load the values for
1062         * the macro variable definitions according to the current process.
1063         */
1064        dtrace_update(dtp);

1066        /*
1067         * Select the intrinsics and typedefs we want based on the data model.
1068         * The intrinsics are under "C".  The typedefs are added under "D".
1069         */
1070        if (dtp->dt_conf.dtc_ctfmodel == CTF_MODEL_ILP32) {
1071                dinp = _dtrace_intrinsics_32;
1072                dtyp = _dtrace_typedefs_32;
1073        } else {
1074                dinp = _dtrace_intrinsics_64;
1075                dtyp = _dtrace_typedefs_64;
1076        }

1078        /*
1079         * Create a dynamic CTF container under the "C" scope for intrinsic
1080         * types and types defined in ANSI-C header files that are included.
1081         */
1082        if ((dmp = dtp->dt_cdefs = dt_module_create(dtp, "C")) == NULL)
1083                return (set_open_errno(dtp, errp, EDT_NOMEM));

1085        if ((dmp->dm_ctfp = ctf_create(&dtp->dt_ctferr)) == NULL)
1086                return (set_open_errno(dtp, errp, EDT_CTF));

1088        dt_dprintf("created CTF container for %s (%p)\n",
1089            dmp->dm_name, (void *)dmp->dm_ctfp);

1091        (void) ctf_setmodel(dmp->dm_ctfp, dtp->dt_conf.dtc_ctfmodel);
1092        ctf_setspecific(dmp->dm_ctfp, dmp);

1094        dmp->dm_flags = DT_DM_LOADED; /* fake up loaded bit */
1095        dmp->dm_modid = -1; /* no module ID */

1097        /*
1098         * Fill the dynamic "C" CTF container with all of the intrinsic
1099         * integer and floating-point types appropriate for this data model.
1100         */
1101        for (; dinp->din_name != NULL; dinp++) {
1102                if (dinp->din_kind == CTF_K_INTEGER) {
1103                        err = ctf_add_integer(dmp->dm_ctfp, CTF_ADD_ROOT,
1104                            dinp->din_name, &dinp->din_data);
1105                } else {
1106                        err = ctf_add_float(dmp->dm_ctfp, CTF_ADD_ROOT,
1107                            dinp->din_name, &dinp->din_data);
1108                }

1110                if (err == CTF_ERR) {
```



```
1111                        dt_dprintf("failed to add %s to C container: %s\n",
1112                            dinp->din_name, ctf_errmsg(
1113                            ctf_errno(dmp->dm_ctfp)));
1114                        return (set_open_errno(dtp, errp, EDT_CTF));
1115                }
1116        }

1118        if (ctf_update(dmp->dm_ctfp) != 0) {
1119                dt_dprintf("failed to update C container: %s\n",
1120                    ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
1121                return (set_open_errno(dtp, errp, EDT_CTF));
1122        }

1124        /*
1125         * Add intrinsic pointer types that are needed to initialize printf
1126         * format dictionary types (see table in dt_printf.c).
1127         */
1128        (void) ctf_add_pointer(dmp->dm_ctfp, CTF_ADD_ROOT,
1129            ctf_lookup_by_name(dmp->dm_ctfp, "void"));

1131        (void) ctf_add_pointer(dmp->dm_ctfp, CTF_ADD_ROOT,
1132            ctf_lookup_by_name(dmp->dm_ctfp, "char"));

1134        (void) ctf_add_pointer(dmp->dm_ctfp, CTF_ADD_ROOT,
1135            ctf_lookup_by_name(dmp->dm_ctfp, "int"));

1137        if (ctf_update(dmp->dm_ctfp) != 0) {
1138                dt_dprintf("failed to update C container: %s\n",
1139                    ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
1140                return (set_open_errno(dtp, errp, EDT_CTF));
1141        }

1143        /*
1144         * Create a dynamic CTF container under the "D" scope for types that
1145         * are defined by the D program itself or on-the-fly by the D compiler.
1146         * The "D" CTF container is a child of the "C" CTF container.
1147         */
1148        if ((dmp = dtp->dt_ddefs = dt_module_create(dtp, "D")) == NULL)
1149                return (set_open_errno(dtp, errp, EDT_NOMEM));

1151        if ((dmp->dm_ctfp = ctf_create(&dtp->dt_ctferr)) == NULL)
1152                return (set_open_errno(dtp, errp, EDT_CTF));

1154        dt_dprintf("created CTF container for %s (%p)\n",
1155            dmp->dm_name, (void *)dmp->dm_ctfp);

1157        (void) ctf_setmodel(dmp->dm_ctfp, dtp->dt_conf.dtc_ctfmodel);
1158        ctf_setspecific(dmp->dm_ctfp, dmp);

1160        dmp->dm_flags = DT_DM_LOADED; /* fake up loaded bit */
1161        dmp->dm_modid = -1; /* no module ID */

1163        if (ctf_import(dmp->dm_ctfp, dtp->dt_cdefs->dm_ctfp) == CTF_ERR) {
1164                dt_dprintf("failed to import D parent container: %s\n",
1165                    ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
1166                return (set_open_errno(dtp, errp, EDT_CTF));
1167        }

1169        /*
1170         * Fill the dynamic "D" CTF container with all of the built-in typedefs
1171         * that we need to use for our D variable and function definitions.
1172         * This ensures that basic inttypes.h names are always available to us.
1173         */
1174        for (; dtyp->dty_src != NULL; dtyp++) {
1175                if (ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1176                    dtyp->dty_dst, ctf_lookup_by_name(dmp->dm_ctfp,
```

```
1177                    dtyp->dty_src)) == CTF_ERR) {
1178                        dt_dprintf("failed to add typedef %s %s to D "
1179                            "container: %s", dtyp->dty_src, dtyp->dty_dst,
1180                            ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
1181                        return (set_open_errno(dtp, errp, EDT_CTF));
1182                }
1183        }

1185        /*
1186         * Insert a CTF ID corresponding to a pointer to a type of kind
1187         * CTF_K_FUNCTION we can use in the compiler for function pointers.
1188         * CTF treats all function pointers as "int (*)()" so we only need one.
1189         */
1190        ctc.ctc_return = ctf_lookup_by_name(dmp->dm_ctfp, "int");
1191        ctc.ctc_argc = 0;
1192        ctc.ctc_flags = 0;

1194        dtp->dt_type_func = ctf_add_function(dmp->dm_ctfp,
1195            CTF_ADD_ROOT, &ctc, NULL);

1197        dtp->dt_type_fptr = ctf_add_pointer(dmp->dm_ctfp,
1198            CTF_ADD_ROOT, dtp->dt_type_func);

1200        /*
1201         * We also insert CTF definitions for the special D intrinsic types
1202         * string and <DYN> into the D container.  The string type is added
1203         * as a typedef of char[n].  The <DYN> type is an alias for void.
1204         * We compare types to these special CTF ids throughout the compiler.
1205         */
1206        ctr.ctr_contents = ctf_lookup_by_name(dmp->dm_ctfp, "char");
1207        ctr.ctr_index = ctf_lookup_by_name(dmp->dm_ctfp, "long");
1208        ctr.ctr_nelems = _dtrace_strsize;

1210        dtp->dt_type_str = ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1211            "string", ctf_add_array(dmp->dm_ctfp, CTF_ADD_ROOT, &ctr));

1213        dtp->dt_type_dyn = ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1214            "<DYN>", ctf_lookup_by_name(dmp->dm_ctfp, "void"));

1216        dtp->dt_type_stack = ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1217            "stack", ctf_lookup_by_name(dmp->dm_ctfp, "void"));

1219        dtp->dt_type_symaddr = ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1220            "_symaddr", ctf_lookup_by_name(dmp->dm_ctfp, "void"));

1222        dtp->dt_type_usymaddr = ctf_add_typedef(dmp->dm_ctfp, CTF_ADD_ROOT,
1223            "_usymaddr", ctf_lookup_by_name(dmp->dm_ctfp, "void"));

1225        if (dtp->dt_type_func == CTF_ERR || dtp->dt_type_fptr == CTF_ERR ||
1226            dtp->dt_type_str == CTF_ERR || dtp->dt_type_dyn == CTF_ERR ||
1227            dtp->dt_type_stack == CTF_ERR || dtp->dt_type_symaddr == CTF_ERR ||
1228            dtp->dt_type_usymaddr == CTF_ERR) {
1229                dt_dprintf("failed to add intrinsic to D container: %s\n",
1230                    ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
1231                return (set_open_errno(dtp, errp, EDT_CTF));
1232        }

1234        if (ctf_update(dmp->dm_ctfp) != 0) {
1235                dt_dprintf("failed update D container: %s\n",
1236                    ctf_errmsg(ctf_errno(dmp->dm_ctfp)));
1237                return (set_open_errno(dtp, errp, EDT_CTF));
1238        }

1240        /*
1241         * Initialize the integer description table used to convert integer
1242         * constants to the appropriate types.  Refer to the comments above
```



```
1243         * dt_node_int() for a complete description of how this table is used.
1244         */
1245        for (i = 0; i < sizeof (dtp->dt_ints) / sizeof (dtp->dt_ints[0]); i++) {
1246                if (dtrace_lookup_by_type(dtp, DTRACE_OBJ_EVERY,
1247                    dtp->dt_ints[i].did_name, &dtt) != 0) {
1248                        dt_dprintf("failed to lookup integer type %s: %s\n",
1249                            dtp->dt_ints[i].did_name,
1250                            dtrace_errmsg(dtp, dtrace_errno(dtp)));
1251                        return (set_open_errno(dtp, errp, dtp->dt_errno));
1252                }
1253                dtp->dt_ints[i].did_ctfp = dtt.dtt_ctfp;
1254                dtp->dt_ints[i].did_type = dtt.dtt_type;
1255        }

1257        /*
1258         * Now that we've created the "C" and "D" containers, move them to the
1259         * start of the module list so that these types and symbols are found
1260         * first (for stability) when iterating through the module list.
1261         */
1262        dt_list_delete(&dtp->dt_modlist, dtp->dt_ddefs);
1263        dt_list_prepend(&dtp->dt_modlist, dtp->dt_ddefs);

1265        dt_list_delete(&dtp->dt_modlist, dtp->dt_cdefs);
1266        dt_list_prepend(&dtp->dt_modlist, dtp->dt_cdefs);

1268        if (dt_pfdict_create(dtp) == -1)
1269                return (set_open_errno(dtp, errp, dtp->dt_errno));

1271        /*
1272         * If we are opening libdtrace DTRACE_O_NODEV enable C_ZDEFS by default
1273         * because without /dev/dtrace open, we will not be able to load the
1274         * names and attributes of any providers or probes from the kernel.
1275         */
1276        if (flags & DTRACE_O_NODEV)
1277                dtp->dt_cflags |= DTRACE_C_ZDEFS;

1279        /*
1280         * Load hard-wired inlines into the definition cache by calling the
1281         * compiler on the raw definition string defined above.
1282         */
1283        if ((pgp = dtrace_program_strcompile(dtp, _dtrace_hardwire,
1284            DTRACE_PROBESPEC_NONE, DTRACE_C_EMPTY, 0, NULL)) == NULL) {
1285                dt_dprintf("failed to load hard-wired definitions: %s\n",
1286                    dtrace_errmsg(dtp, dtrace_errno(dtp)));
1287                return (set_open_errno(dtp, errp, EDT_HARDWIRE));
1288        }

1290        dt_program_destroy(dtp, pgp);

1292        /*
1293         * Set up the default DTrace library path.  Once set, the next call to
1294         * dt_compile() will compile all the libraries.  We intentionally defer
1295         * library processing to improve overhead for clients that don't ever
1296         * compile, and to provide better error reporting (because the full
1297         * reporting of compiler errors requires dtrace_open() to succeed).
1298         */
1299        if (dtrace_setopt(dtp, "libdir", _dtrace_libdir) != 0)
1300                return (set_open_errno(dtp, errp, dtp->dt_errno));

1302        return (dtp);
1303 }

1305 dtrace_hdl_t *
1306 dtrace_open(int version, int flags, int *errp)
1307 {
1308        return (dt_vopen(version, flags, errp, NULL, NULL));
```

```
1309 }
1311 dtrace_hdl_t *
1312 dtrace_vopen(int version, int flags, int *errp,
1313     const dtrace_vector_t *vector, void *arg)
1314 {
1315         return (dt_vopen(version, flags, errp, vector, arg));
1316 }
1318 void
1319 dtrace_close(dtrace_hdl_t *dtp)
1320 {
1321         dt_ident_t *idp, *ndp;
1322         dt_module_t *dmp;
1323         dt_provider_t *pvp;
1324         dtrace_prog_t *pgp;
1325         dt_xlator_t *dxp;
1326         dt_dirpath_t *dirp;
1327         int i;
1329         if (dtp->dt_procs != NULL)
1330                 dt_proc_fini(dtp);
1332         while ((pgp = dt_list_next(&dtp->dt_programs)) != NULL)
1333                 dt_program_destroy(dtp, pgp);
1335         while ((dxp = dt_list_next(&dtp->dt_xlators)) != NULL)
1336                 dt_xlator_destroy(dtp, dxp);
1338         dt_free(dtp, dtp->dt_xlatormap);
1340         for (idp = dtp->dt_externs; idp != NULL; idp = ndp) {
1341                 ndp = idp->di_next;
1342                 dt_ident_destroy(idp);
1343         }
1345         if (dtp->dt_macros != NULL)
1346                 dt_idhash_destroy(dtp->dt_macros);
1347         if (dtp->dt_aggs != NULL)
1348                 dt_idhash_destroy(dtp->dt_aggs);
1349         if (dtp->dt_globals != NULL)
1350                 dt_idhash_destroy(dtp->dt_globals);
1351         if (dtp->dt_tls != NULL)
1352                 dt_idhash_destroy(dtp->dt_tls);
1354         while ((dmp = dt_list_next(&dtp->dt_modlist)) != NULL)
1355                 dt_module_destroy(dtp, dmp);
1357         while ((pvp = dt_list_next(&dtp->dt_provlist)) != NULL)
1358                 dt_provider_destroy(dtp, pvp);
1360         if (dtp->dt_fd != -1)
1361                 (void) close(dtp->dt_fd);
1362         if (dtp->dt_ftfd != -1)
1363                 (void) close(dtp->dt_ftfd);
1364         if (dtp->dt_cdefs_fd != -1)
1365                 (void) close(dtp->dt_cdefs_fd);
1366         if (dtp->dt_ddefs_fd != -1)
1367                 (void) close(dtp->dt_ddefs_fd);
1368         if (dtp->dt_stdout_fd != -1)
1369                 (void) close(dtp->dt_stdout_fd);
1371         dt_epid_destroy(dtp);
1372         dt_aggid_destroy(dtp);
1373         dt_format_destroy(dtp);
1374         dt_strdata_destroy(dtp);
```



```
1375         dt_buffered_destroy(dtp);
1376         dt_aggregate_destroy(dtp);
1377         dt_pfdict_destroy(dtp);
1378         dt_provmod_destroy(&dtp->dt_provmod);
1379         dt_dof_fini(dtp);
1381         for (i = 1; i < dtp->dt_cpp_argc; i++)
1382                 free(dtp->dt_cpp_argv[i]);
1384         while ((dirp = dt_list_next(&dtp->dt_lib_path)) != NULL) {
1385                 dt_list_delete(&dtp->dt_lib_path, dirp);
1386                 free(dirp->dir_path);
1387                 free(dirp);
1388         }
1390         free(dtp->dt_cpp_argv);
1391         free(dtp->dt_cpp_path);
1392         free(dtp->dt_ld_path);
1394         free(dtp->dt_mods);
1395         free(dtp->dt_provs);
1396         free(dtp);
1397 }
1399 int
1400 dtrace_provider_modules(dtrace_hdl_t *dtp, const char **mods, int nmods)
1401 {
1402         dt_provmod_t *prov;
1403         int i = 0;
1405         for (prov = dtp->dt_provmod; prov != NULL; prov = prov->dp_next, i++) {
1406                 if (i < nmods)
1407                         mods[i] = prov->dp_name;
1408         }
1410         return (i);
1411 }
1413 int
1414 dtrace_ctlfd(dtrace_hdl_t *dtp)
1415 {
1416         return (dtp->dt_fd);
1417 }
```

```
**********************************************************
    7992 Tue Jan 14 16:50:00 2014
new/usr/src/lib/libdtrace/common/io.d.in
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 */
#pragma ident   "%Z%%M% %I%     %E% SMI"

#pragma D depends_on module unix
#pragma D depends_on provider io

inline int B_BUSY = @B_BUSY@;
#pragma D binding "1.0" B_BUSY
inline int B_DONE = @B_DONE@;
#pragma D binding "1.0" B_DONE
inline int B_ERROR = @B_ERROR@;
#pragma D binding "1.0" B_ERROR
inline int B_PAGEIO = @B_PAGEIO@;
#pragma D binding "1.0" B_PAGEIO
inline int B_PHYS = @B_PHYS@;
#pragma D binding "1.0" B_PHYS
inline int B_READ = @B_READ@;
#pragma D binding "1.0" B_READ
inline int B_WRITE = @B_WRITE@;
#pragma D binding "1.0" B_WRITE
inline int B_ASYNC = @B_ASYNC@;
#pragma D binding "1.0" B_ASYNC

typedef struct bufinfo {
        int b_flags;                    /* buffer status */
        size_t b_bcount;                /* number of bytes */
        caddr_t b_addr;                 /* buffer address */
        uint64_t b_lblkno;              /* block # on device */
        uint64_t b_blkno;               /* expanded block # on device */
        size_t b_resid;                 /* # of bytes not transferred */
```



```
        size_t b_bufsize;               /* size of allocated buffer */
        caddr_t b_iodone;               /* I/O completion routine */
        int b_error;                    /* expanded error field */
        dev_t b_edev;                   /* extended device */
} bufinfo_t;
______unchanged_portion_omitted_

inline fileinfo_t fds[int fd] = xlate <fileinfo_t> (getf(fd));
inline fileinfo_t fds[int fd] = xlate <fileinfo_t> (
    fd >= 0 && fd < curthread->t_procp->p_user.u_finfo.fi_nfiles ?
    curthread->t_procp->p_user.u_finfo.fi_list[fd].uf_file : NULL);

#pragma D attributes Stable/Stable/Common fds
#pragma D binding "1.1" fds

#pragma D binding "1.2" translator
translator fileinfo_t < struct vnode *V > {
        fi_name = V->v_path == NULL ? "<unknown>" :
            basename(cleanpath(V->v_path));
        fi_dirname = V->v_path == NULL ? "<unknown>" :
            dirname(cleanpath(V->v_path));
        fi_pathname = V->v_path == NULL ? "<unknown>" : cleanpath(V->v_path);
        fi_fs = stringof(V->v_op->vnop_name);
        fi_mount = V->v_vfsp->vfs_vnodecovered == NULL ? "/" :
            V->v_vfsp->vfs_vnodecovered->v_path == NULL ? "<unknown>" :
            cleanpath(V->v_vfsp->vfs_vnodecovered->v_path);
};
______unchanged_portion_omitted_
```

```
**********************************************************
  119866 Tue Jan 14 16:50:00 2014
new/usr/src/pkg/manifests/system-dtrace-tests.mf
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

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

#
# Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright (c) 2012 by Delphix. All rights reserved.
#

set name=pkg.fmri value=pkg:/system/dtrace/tests@$(PKGVERS)
set name=pkg.description value="DTrace Test Suite Internal Distribution"
set name=pkg.summary value="DTrace Test Suite"
set name=info.classification \
    value=org.opensolaris.category.2008:Development/System
set name=variant.arch value=$(ARCH)
dir path=opt/SUNWdtrt group=sys
dir path=opt/SUNWdtrt/bin
dir path=opt/SUNWdtrt/bin/$(ARCH32)
dir path=opt/SUNWdtrt/bin/$(ARCH64)
dir path=opt/SUNWdtrt/lib
dir path=opt/SUNWdtrt/lib/java
dir path=opt/SUNWdtrt/tst
dir path=opt/SUNWdtrt/tst/$(ARCH)
dir path=opt/SUNWdtrt/tst/$(ARCH)/arrays
$(i386_ONLY)dir path=opt/SUNWdtrt/tst/$(ARCH)/funcs
dir path=opt/SUNWdtrt/tst/$(ARCH)/pid
$(sparc_ONLY)dir path=opt/SUNWdtrt/tst/$(ARCH)/usdt
dir path=opt/SUNWdtrt/tst/$(ARCH)/ustack
dir path=opt/SUNWdtrt/tst/common
dir path=opt/SUNWdtrt/tst/common/aggs
dir path=opt/SUNWdtrt/tst/common/arithmetic
dir path=opt/SUNWdtrt/tst/common/arrays
dir path=opt/SUNWdtrt/tst/common/assocs
dir path=opt/SUNWdtrt/tst/common/begin
dir path=opt/SUNWdtrt/tst/common/bitfields
dir path=opt/SUNWdtrt/tst/common/buffering
dir path=opt/SUNWdtrt/tst/common/builtinvar
dir path=opt/SUNWdtrt/tst/common/cg
dir path=opt/SUNWdtrt/tst/common/clauses
dir path=opt/SUNWdtrt/tst/common/cpc
```



```
dir path=opt/SUNWdtrt/tst/common/decls
dir path=opt/SUNWdtrt/tst/common/drops
dir path=opt/SUNWdtrt/tst/common/dtraceUtil
dir path=opt/SUNWdtrt/tst/common/end
dir path=opt/SUNWdtrt/tst/common/enum
dir path=opt/SUNWdtrt/tst/common/env
dir path=opt/SUNWdtrt/tst/common/error
dir path=opt/SUNWdtrt/tst/common/exit
dir path=opt/SUNWdtrt/tst/common/fbtprovider
dir path=opt/SUNWdtrt/tst/common/funcs
dir path=opt/SUNWdtrt/tst/common/grammar
dir path=opt/SUNWdtrt/tst/common/include
dir path=opt/SUNWdtrt/tst/common/inline
dir path=opt/SUNWdtrt/tst/common/io
dir path=opt/SUNWdtrt/tst/common/ip
dir path=opt/SUNWdtrt/tst/common/java_api
dir path=opt/SUNWdtrt/tst/common/lexer
dir path=opt/SUNWdtrt/tst/common/llquantize
dir path=opt/SUNWdtrt/tst/common/mdb
dir path=opt/SUNWdtrt/tst/common/mib
dir path=opt/SUNWdtrt/tst/common/misc
dir path=opt/SUNWdtrt/tst/common/multiaggs
dir path=opt/SUNWdtrt/tst/common/nfs
dir path=opt/SUNWdtrt/tst/common/offsetof
dir path=opt/SUNWdtrt/tst/common/operators
dir path=opt/SUNWdtrt/tst/common/pid
dir path=opt/SUNWdtrt/tst/common/plockstat
dir path=opt/SUNWdtrt/tst/common/pointers
dir path=opt/SUNWdtrt/tst/common/pragma
dir path=opt/SUNWdtrt/tst/common/predicates
dir path=opt/SUNWdtrt/tst/common/preprocessor
dir path=opt/SUNWdtrt/tst/common/print
dir path=opt/SUNWdtrt/tst/common/printa
dir path=opt/SUNWdtrt/tst/common/printf
dir path=opt/SUNWdtrt/tst/common/privs
dir path=opt/SUNWdtrt/tst/common/probes
dir path=opt/SUNWdtrt/tst/common/proc
dir path=opt/SUNWdtrt/tst/common/profile-n
dir path=opt/SUNWdtrt/tst/common/providers
dir path=opt/SUNWdtrt/tst/common/raise
dir path=opt/SUNWdtrt/tst/common/rates
dir path=opt/SUNWdtrt/tst/common/safety
dir path=opt/SUNWdtrt/tst/common/scalars
dir path=opt/SUNWdtrt/tst/common/sched
dir path=opt/SUNWdtrt/tst/common/scripting
dir path=opt/SUNWdtrt/tst/common/sdt
dir path=opt/SUNWdtrt/tst/common/sizeof
dir path=opt/SUNWdtrt/tst/common/speculation
dir path=opt/SUNWdtrt/tst/common/stability
dir path=opt/SUNWdtrt/tst/common/stack
dir path=opt/SUNWdtrt/tst/common/stackdepth
dir path=opt/SUNWdtrt/tst/common/stop
dir path=opt/SUNWdtrt/tst/common/strlen
dir path=opt/SUNWdtrt/tst/common/struct
dir path=opt/SUNWdtrt/tst/common/syscall
dir path=opt/SUNWdtrt/tst/common/sysevent
dir path=opt/SUNWdtrt/tst/common/tick-n
dir path=opt/SUNWdtrt/tst/common/trace
dir path=opt/SUNWdtrt/tst/common/tracemem
dir path=opt/SUNWdtrt/tst/common/translators
dir path=opt/SUNWdtrt/tst/common/typedef
dir path=opt/SUNWdtrt/tst/common/types
dir path=opt/SUNWdtrt/tst/common/union
dir path=opt/SUNWdtrt/tst/common/usdt
dir path=opt/SUNWdtrt/tst/common/ustack
dir path=opt/SUNWdtrt/tst/common/vars
```

```
dir path=opt/SUNWdtrt/tst/common/version
$(i386_ONLY)dir path=opt/SUNWdtrt/tst/i86xpv
$(i386_ONLY)dir path=opt/SUNWdtrt/tst/i86xpv/xdt
file path=opt/SUNWdtrt/README mode=0444
file path=opt/SUNWdtrt/bin/$(ARCH32)/chkargs mode=0555
file path=opt/SUNWdtrt/bin/$(ARCH64)/chkargs mode=0555
file path=opt/SUNWdtrt/bin/baddof mode=0555
file path=opt/SUNWdtrt/bin/badioctl mode=0555
file path=opt/SUNWdtrt/bin/chkargs mode=0555
file path=opt/SUNWdtrt/bin/dstyle mode=0555
file path=opt/SUNWdtrt/bin/dtest mode=0555
file path=opt/SUNWdtrt/bin/dtfailures mode=0555
file path=opt/SUNWdtrt/bin/exception.lst mode=0444
file path=opt/SUNWdtrt/bin/jdtrace mode=0555
file path=opt/SUNWdtrt/lib/java/jdtrace.jar
file path=opt/SUNWdtrt/tst/$(ARCH)/arrays/tst.uregsarray.d mode=0444
$(i386_ONLY)file path=opt/SUNWdtrt/tst/$(ARCH)/funcs/tst.badcopyin.d mode=0444
$(i386_ONLY)file path=opt/SUNWdtrt/tst/$(ARCH)/funcs/tst.badcopyinstr.d \
    mode=0444
$(i386_ONLY)file path=opt/SUNWdtrt/tst/$(ARCH)/funcs/tst.badcopyout.d \
    mode=0444
$(i386_ONLY)file path=opt/SUNWdtrt/tst/$(ARCH)/funcs/tst.badcopyoutstr.d \
    mode=0444
$(sparc_ONLY)file \
    path=opt/SUNWdtrt/tst/$(ARCH)/pid/err.D_PROC_ALIGN.misaligned.d mode=0444
$(sparc_ONLY)file \
    path=opt/SUNWdtrt/tst/$(ARCH)/pid/err.D_PROC_ALIGN.misaligned.exe \
    mode=0555
$(i386_ONLY)file path=opt/SUNWdtrt/tst/$(ARCH)/pid/tst.badinstr.d mode=0444
$(i386_ONLY)file path=opt/SUNWdtrt/tst/$(ARCH)/pid/tst.badinstr.exe mode=0555
$(sparc_ONLY)file path=opt/SUNWdtrt/tst/$(ARCH)/pid/tst.br.d mode=0444
$(sparc_ONLY)file path=opt/SUNWdtrt/tst/$(ARCH)/pid/tst.br.d.out mode=0444
$(sparc_ONLY)file path=opt/SUNWdtrt/tst/$(ARCH)/pid/tst.br.exe mode=0555
file path=opt/SUNWdtrt/tst/$(ARCH)/pid/tst.branch.d mode=0444
file path=opt/SUNWdtrt/tst/$(ARCH)/pid/tst.branch.exe mode=0555
file path=opt/SUNWdtrt/tst/$(ARCH)/pid/tst.embedded.d mode=0444
file path=opt/SUNWdtrt/tst/$(ARCH)/pid/tst.embedded.exe mode=0555
$(i386_ONLY)file path=opt/SUNWdtrt/tst/$(ARCH)/pid/tst.ret.d mode=0444
$(i386_ONLY)file path=opt/SUNWdtrt/tst/$(ARCH)/pid/tst.ret.exe mode=0555
$(i386_ONLY)file path=opt/SUNWdtrt/tst/$(ARCH)/pid/tst.retlist.exe mode=0555
$(i386_ONLY)file path=opt/SUNWdtrt/tst/$(ARCH)/pid/tst.retlist.ksh mode=0444
$(sparc_ONLY)file path=opt/SUNWdtrt/tst/$(ARCH)/usdt/tst.tailcall.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/$(ARCH)/ustack/tst.annotated.d mode=0444
file path=opt/SUNWdtrt/tst/$(ARCH)/ustack/tst.annotated.d.out mode=0444
file path=opt/SUNWdtrt/tst/$(ARCH)/ustack/tst.annotated.exe mode=0555
file path=opt/SUNWdtrt/tst/$(ARCH)/ustack/tst.circstack.d mode=0444
file path=opt/SUNWdtrt/tst/$(ARCH)/ustack/tst.circstack.exe mode=0555
file path=opt/SUNWdtrt/tst/$(ARCH)/ustack/tst.helper.d mode=0444
file path=opt/SUNWdtrt/tst/$(ARCH)/ustack/tst.helper.d.out mode=0444
file path=opt/SUNWdtrt/tst/$(ARCH)/ustack/tst.helper.exe mode=0555
$(sparc_ONLY)file path=opt/SUNWdtrt/tst/$(ARCH)/ustack/tst.trapstat.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_AGG_FUNC.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_AGG_MDIM.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_AGG_NULL.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_AGG_REDEF.redef.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_AGG_SCALAR.avgtoofew.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_AGG_SCALAR.maxnoarg.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_AGG_SCALAR.mintoofew.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_AGG_SCALAR.quantizetoofew.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_AGG_SCALAR.stddevtoofew.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_AGG_SCALAR.sumtoofew.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_CLEAR_AGGARG.bad.d mode=0444
```



```
file path=opt/SUNWdtrt/tst/common/aggs/err.D_CLEAR_PROTO.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_FUNC_IDENT.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_FUNC_UNDEF.badaggfunc.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_IDENT_UNDEF.badexpr.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_IDENT_UNDEF.badkey3.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_IDENT_UNDEF.noeffect.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_KEY_TYPE.badkey1.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_KEY_TYPE.badkey2.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_KEY_TYPE.badkey4.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_LQUANT_BASETYPE.lqbad1.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_LQUANT_BASETYPE.lqshort.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_LQUANT_BASEVAL.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_LQUANT_LIMTYPE.lqbad1.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_LQUANT_LIMVAL.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_LQUANT_MATCHBASE.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_LQUANT_MATCHBASE.order.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_LQUANT_MATCHLIM.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_LQUANT_MATCHLIM.order.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_LQUANT_MATCHSTEP.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_LQUANT_MISMATCH.lqbadarg.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_LQUANT_STEPLARGE.lqtoofew.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_LQUANT_STEPSMALL.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_LQUANT_STEPTYPE.lqbadinc.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_LQUANT_STEPVAL.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_NORMALIZE_AGGARG.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_NORMALIZE_PROTO.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_NORMALIZE_SCALAR.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_PROTO_ARG.lquantizetoofew.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_PROTO_LEN.avgnoarg.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_PROTO_LEN.avgtoomany.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_PROTO_LEN.counttoomany.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_PROTO_LEN.lquantizenoarg.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_PROTO_LEN.lquantizetoomany.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_PROTO_LEN.maxnoarg.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_PROTO_LEN.maxtoomany.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_PROTO_LEN.minnoarg.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_PROTO_LEN.mintoomany.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_PROTO_LEN.quantizenoarg.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_PROTO_LEN.quantizetoomany.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_PROTO_LEN.stddevnoarg.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_PROTO_LEN.stddevtoomany.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_PROTO_LEN.sumnoarg.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_PROTO_LEN.sumtoomany.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_TRUNC_AGGARG.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_TRUNC_PROTO.badmany.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_TRUNC_PROTO.badnone.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/err.D_TRUNC_SCALAR.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.allquant.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.allquant.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.avg.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.avg.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.avg_neg.d mode=0444
file path=opt/SUNWdtrt/tst/common/aggs/tst.avg_neg.d.out mode=0444
```

```
256 file path=opt/SUNWdtrt/tst/common/aggs/tst.clear.d mode=0444
257 file path=opt/SUNWdtrt/tst/common/aggs/tst.clear.d.out mode=0444
258 file path=opt/SUNWdtrt/tst/common/aggs/tst.clearavg.d mode=0444
259 file path=opt/SUNWdtrt/tst/common/aggs/tst.clearavg.d.out mode=0444
260 file path=opt/SUNWdtrt/tst/common/aggs/tst.clearavg2.d mode=0444
261 file path=opt/SUNWdtrt/tst/common/aggs/tst.clearavg2.d.out mode=0444
262 file path=opt/SUNWdtrt/tst/common/aggs/tst.cleardenormalize.d mode=0444
263 file path=opt/SUNWdtrt/tst/common/aggs/tst.cleardenormalize.d.out mode=0444
264 file path=opt/SUNWdtrt/tst/common/aggs/tst.clearlquantize.d mode=0444
265 file path=opt/SUNWdtrt/tst/common/aggs/tst.clearlquantize.d.out mode=0444
266 file path=opt/SUNWdtrt/tst/common/aggs/tst.clearnormalize.d mode=0444
267 file path=opt/SUNWdtrt/tst/common/aggs/tst.clearnormalize.d.out mode=0444
268 file path=opt/SUNWdtrt/tst/common/aggs/tst.clearstddev.d mode=0444
269 file path=opt/SUNWdtrt/tst/common/aggs/tst.clearstddev.d.out mode=0444
270 file path=opt/SUNWdtrt/tst/common/aggs/tst.count.d mode=0444
271 file path=opt/SUNWdtrt/tst/common/aggs/tst.count.d.out mode=0444
272 file path=opt/SUNWdtrt/tst/common/aggs/tst.count2.d mode=0444
273 file path=opt/SUNWdtrt/tst/common/aggs/tst.count2.d.out mode=0444
274 file path=opt/SUNWdtrt/tst/common/aggs/tst.count3.d mode=0444
275 file path=opt/SUNWdtrt/tst/common/aggs/tst.denormalize.d mode=0444
276 file path=opt/SUNWdtrt/tst/common/aggs/tst.denormalize.d.out mode=0444
277 file path=opt/SUNWdtrt/tst/common/aggs/tst.denormalizeonly.d mode=0444
278 file path=opt/SUNWdtrt/tst/common/aggs/tst.denormalizeonly.d.out mode=0444
279 file path=opt/SUNWdtrt/tst/common/aggs/tst.fmtnormalize.d mode=0444
280 file path=opt/SUNWdtrt/tst/common/aggs/tst.fmtnormalize.d.out mode=0444
281 file path=opt/SUNWdtrt/tst/common/aggs/tst.forms.d mode=0444
282 file path=opt/SUNWdtrt/tst/common/aggs/tst.forms.d.out mode=0444
283 file path=opt/SUNWdtrt/tst/common/aggs/tst.goodkey.d mode=0444
284 file path=opt/SUNWdtrt/tst/common/aggs/tst.keysort.d mode=0444
285 file path=opt/SUNWdtrt/tst/common/aggs/tst.keysort.d.out mode=0444
286 file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantize.d mode=0444
287 file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantize.d.out mode=0444
288 file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantnormal.d mode=0444
289 file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantnormal.d.out mode=0444
290 file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantrange.d mode=0444
291 file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantrange.d.out mode=0444
292 file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantround.d mode=0444
293 file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantround.d.out mode=0444
294 file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantzero.d mode=0444
295 file path=opt/SUNWdtrt/tst/common/aggs/tst.lquantzero.d.out mode=0444
296 file path=opt/SUNWdtrt/tst/common/aggs/tst.max.d mode=0444
297 file path=opt/SUNWdtrt/tst/common/aggs/tst.max.d.out mode=0444
298 file path=opt/SUNWdtrt/tst/common/aggs/tst.max_neg.d mode=0444
299 file path=opt/SUNWdtrt/tst/common/aggs/tst.max_neg.d.out mode=0444
300 file path=opt/SUNWdtrt/tst/common/aggs/tst.min.d mode=0444
301 file path=opt/SUNWdtrt/tst/common/aggs/tst.min.d.out mode=0444
302 file path=opt/SUNWdtrt/tst/common/aggs/tst.min_neg.d mode=0444
303 file path=opt/SUNWdtrt/tst/common/aggs/tst.min_neg.d.out mode=0444
304 file path=opt/SUNWdtrt/tst/common/aggs/tst.multiaggs1.d mode=0444
305 file path=opt/SUNWdtrt/tst/common/aggs/tst.multiaggs2.d mode=0444
306 file path=opt/SUNWdtrt/tst/common/aggs/tst.multiaggs2.d.out mode=0444
307 file path=opt/SUNWdtrt/tst/common/aggs/tst.multiaggs3.d mode=0444
308 file path=opt/SUNWdtrt/tst/common/aggs/tst.multiaggs3.d.out mode=0444
309 file path=opt/SUNWdtrt/tst/common/aggs/tst.multinormalize.d mode=0444
310 file path=opt/SUNWdtrt/tst/common/aggs/tst.multinormalize.d.out mode=0444
311 file path=opt/SUNWdtrt/tst/common/aggs/tst.neglquant.d mode=0444
312 file path=opt/SUNWdtrt/tst/common/aggs/tst.neglquant.d.out mode=0444
313 file path=opt/SUNWdtrt/tst/common/aggs/tst.negorder.d mode=0444
314 file path=opt/SUNWdtrt/tst/common/aggs/tst.negorder.d.out mode=0444
315 file path=opt/SUNWdtrt/tst/common/aggs/tst.negquant.d mode=0444
316 file path=opt/SUNWdtrt/tst/common/aggs/tst.negquant.d.out mode=0444
317 file path=opt/SUNWdtrt/tst/common/aggs/tst.negtrunc.d mode=0444
318 file path=opt/SUNWdtrt/tst/common/aggs/tst.negtrunc.d.out mode=0444
319 file path=opt/SUNWdtrt/tst/common/aggs/tst.negtruncquant.d mode=0444
320 file path=opt/SUNWdtrt/tst/common/aggs/tst.negtruncquant.d.out mode=0444
321 file path=opt/SUNWdtrt/tst/common/aggs/tst.normalize.d mode=0444
```



```
322 file path=opt/SUNWdtrt/tst/common/aggs/tst.normalize.d.out mode=0444
323 file path=opt/SUNWdtrt/tst/common/aggs/tst.order.d mode=0444
324 file path=opt/SUNWdtrt/tst/common/aggs/tst.order.d.out mode=0444
325 file path=opt/SUNWdtrt/tst/common/aggs/tst.quantize.d mode=0444
326 file path=opt/SUNWdtrt/tst/common/aggs/tst.quantize.d.out mode=0444
327 file path=opt/SUNWdtrt/tst/common/aggs/tst.quantmany.d mode=0444
328 file path=opt/SUNWdtrt/tst/common/aggs/tst.quantmany.d.out mode=0444
329 file path=opt/SUNWdtrt/tst/common/aggs/tst.quantround.d mode=0444
330 file path=opt/SUNWdtrt/tst/common/aggs/tst.quantround.d.out mode=0444
331 file path=opt/SUNWdtrt/tst/common/aggs/tst.quantzero.d mode=0444
332 file path=opt/SUNWdtrt/tst/common/aggs/tst.quantzero.d.out mode=0444
333 file path=opt/SUNWdtrt/tst/common/aggs/tst.signature.d mode=0444
334 file path=opt/SUNWdtrt/tst/common/aggs/tst.signedkeys.d mode=0444
335 file path=opt/SUNWdtrt/tst/common/aggs/tst.signedkeys.d.out mode=0444
336 file path=opt/SUNWdtrt/tst/common/aggs/tst.signedkeyspos.d mode=0444
337 file path=opt/SUNWdtrt/tst/common/aggs/tst.signedkeyspos.d.out mode=0444
338 file path=opt/SUNWdtrt/tst/common/aggs/tst.sizedkeys.d mode=0444
339 file path=opt/SUNWdtrt/tst/common/aggs/tst.sizedkeys.d.out mode=0444
340 file path=opt/SUNWdtrt/tst/common/aggs/tst.stddev.d mode=0444
341 file path=opt/SUNWdtrt/tst/common/aggs/tst.stddev.d.out mode=0444
342 file path=opt/SUNWdtrt/tst/common/aggs/tst.subr.d mode=0444
343 file path=opt/SUNWdtrt/tst/common/aggs/tst.sum.d mode=0444
344 file path=opt/SUNWdtrt/tst/common/aggs/tst.sum.d.out mode=0444
345 file path=opt/SUNWdtrt/tst/common/aggs/tst.trunc.d mode=0444
346 file path=opt/SUNWdtrt/tst/common/aggs/tst.trunc.d.out mode=0444
347 file path=opt/SUNWdtrt/tst/common/aggs/tst.trunc0.d mode=0444
348 file path=opt/SUNWdtrt/tst/common/aggs/tst.trunc0.d.out mode=0444
349 file path=opt/SUNWdtrt/tst/common/aggs/tst.truncquant.d mode=0444
350 file path=opt/SUNWdtrt/tst/common/aggs/tst.truncquant.d.out mode=0444
351 file path=opt/SUNWdtrt/tst/common/aggs/tst.valsortkeypos.d mode=0444
352 file path=opt/SUNWdtrt/tst/common/aggs/tst.valsortkeypos.d.out mode=0444
353 file path=opt/SUNWdtrt/tst/common/arithmetic/err.D_DIV_ZERO.divby0.d mode=0444
354 file path=opt/SUNWdtrt/tst/common/arithmetic/err.D_DIV_ZERO.divby0_1.d \
355     mode=0444
356 file path=opt/SUNWdtrt/tst/common/arithmetic/err.D_DIV_ZERO.divby0_2.d \
357     mode=0444
358 file path=opt/SUNWdtrt/tst/common/arithmetic/err.D_DIV_ZERO.modby0.d mode=0444
359 file path=opt/SUNWdtrt/tst/common/arithmetic/err.D_SYNTAX.addmin.d mode=0444
360 file path=opt/SUNWdtrt/tst/common/arithmetic/err.D_SYNTAX.divmin.d mode=0444
361 file path=opt/SUNWdtrt/tst/common/arithmetic/err.D_SYNTAX.muladd.d mode=0444
362 file path=opt/SUNWdtrt/tst/common/arithmetic/err.D_SYNTAX.muldiv.d mode=0444
363 file path=opt/SUNWdtrt/tst/common/arithmetic/tst.basics.d mode=0444
364 file path=opt/SUNWdtrt/tst/common/arithmetic/tst.basics.d.out mode=0444
365 file path=opt/SUNWdtrt/tst/common/arithmetic/tst.compcast.d mode=0444
366 file path=opt/SUNWdtrt/tst/common/arithmetic/tst.compcast.d.out mode=0444
367 file path=opt/SUNWdtrt/tst/common/arithmetic/tst.compnarrowassign.d mode=0444
368 file path=opt/SUNWdtrt/tst/common/arithmetic/tst.compnarrowassign.d.out \
369     mode=0444
370 file path=opt/SUNWdtrt/tst/common/arithmetic/tst.execcast.d mode=0444
371 file path=opt/SUNWdtrt/tst/common/arithmetic/tst.execcast.d.out mode=0444
372 file path=opt/SUNWdtrt/tst/common/arrays/err.D_ARR_BADREF.bad.d mode=0444
373 file path=opt/SUNWdtrt/tst/common/arrays/err.D_DECL_ARRBIG.toobig.d mode=0444
374 file path=opt/SUNWdtrt/tst/common/arrays/err.D_DECL_ARRNULL.bad.d mode=0444
375 file path=opt/SUNWdtrt/tst/common/arrays/err.D_DECL_ARRSUB.bad.d mode=0444
376 file path=opt/SUNWdtrt/tst/common/arrays/err.D_DECL_PROTO_TYPE.badtuple.d \
377     mode=0444
378 file path=opt/SUNWdtrt/tst/common/arrays/err.D_IDENT_UNDEF.badureg.d mode=0444
379 file path=opt/SUNWdtrt/tst/common/arrays/tst.basic1.d mode=0444
380 file path=opt/SUNWdtrt/tst/common/arrays/tst.basic2.d mode=0444
381 file path=opt/SUNWdtrt/tst/common/arrays/tst.basic3.d mode=0444
382 file path=opt/SUNWdtrt/tst/common/arrays/tst.basic4.d mode=0444
383 file path=opt/SUNWdtrt/tst/common/arrays/tst.basic5.d mode=0444
384 file path=opt/SUNWdtrt/tst/common/arrays/tst.basic6.d mode=0444
385 file path=opt/SUNWdtrt/tst/common/arrays/tst.uregsarray.d mode=0444
386 file path=opt/SUNWdtrt/tst/common/assocs/err.D_OP_INCOMPAT.dupgtype.d \
387     mode=0444
```

```
388 file path=opt/SUNWdtrt/tst/common/assocs/err.D_OP_INCOMPAT.dupttype.d \
389     mode=0444
390 file path=opt/SUNWdtrt/tst/common/assocs/err.D_OP_INCOMPAT.this.d mode=0444
391 file path=opt/SUNWdtrt/tst/common/assocs/err.D_PROTO_ARG.badsig.d mode=0444
392 file path=opt/SUNWdtrt/tst/common/assocs/err.D_PROTO_LEN.toofew.d mode=0444
393 file path=opt/SUNWdtrt/tst/common/assocs/err.D_PROTO_LEN.toomany.d mode=0444
394 file path=opt/SUNWdtrt/tst/common/assocs/err.D_SYNTAX.errassign.d mode=0444
395 file path=opt/SUNWdtrt/tst/common/assocs/err.tupoflow.d mode=0444
396 file path=opt/SUNWdtrt/tst/common/assocs/tst.cpyarray.d mode=0444
397 file path=opt/SUNWdtrt/tst/common/assocs/tst.diffprofile.d mode=0444
398 file path=opt/SUNWdtrt/tst/common/assocs/tst.initialize.d mode=0444
399 file path=opt/SUNWdtrt/tst/common/assocs/tst.invalidref.d mode=0444
400 file path=opt/SUNWdtrt/tst/common/assocs/tst.misc.d mode=0444
401 file path=opt/SUNWdtrt/tst/common/assocs/tst.orthogonality.d mode=0444
402 file path=opt/SUNWdtrt/tst/common/assocs/tst.this.d mode=0444
403 file path=opt/SUNWdtrt/tst/common/assocs/tst.valassign.d.out mode=0444
404 file path=opt/SUNWdtrt/tst/common/begin/err.D_PDESC_ZERO.begin.d mode=0444
405 file path=opt/SUNWdtrt/tst/common/begin/err.D_PDESC_ZERO.tick.d mode=0444
406 file path=opt/SUNWdtrt/tst/common/begin/tst.begin.d mode=0444
407 file path=opt/SUNWdtrt/tst/common/begin/tst.begin.d.out mode=0444
408 file path=opt/SUNWdtrt/tst/common/begin/tst.multibegin.d mode=0444
409 file path=opt/SUNWdtrt/tst/common/begin/tst.multibegin.d.out mode=0444
410 file \
411     path=opt/SUNWdtrt/tst/common/bitfields/err.D_ADDROF_BITFIELD.BitfieldAddress
412     mode=0444
413 file path=opt/SUNWdtrt/tst/common/bitfields/err.D_DECL_BFCONST.NegBitField.d \
414     mode=0444
415 file path=opt/SUNWdtrt/tst/common/bitfields/err.D_DECL_BFCONST.ZeroBitField.d \
416     mode=0444
417 file path=opt/SUNWdtrt/tst/common/bitfields/err.D_DECL_BFSIZE.ExceedBaseType.d \
418     mode=0444
419 file path=opt/SUNWdtrt/tst/common/bitfields/err.D_DECL_BFSIZE.GreaterThan64.d \
420     mode=0444
421 file path=opt/SUNWdtrt/tst/common/bitfields/err.D_DECL_BFTYPE.badtype.d \
422     mode=0444
423 file path=opt/SUNWdtrt/tst/common/bitfields/err.D_OFFSETOF_BITFIELD.d \
424     mode=0444
425 file \
426     path=opt/SUNWdtrt/tst/common/bitfields/err.D_SIZEOF_BITFIELD.SizeofBitfield.
427     mode=0444
428 file path=opt/SUNWdtrt/tst/common/bitfields/tst.BitFieldPromotion.d mode=0444
429 file path=opt/SUNWdtrt/tst/common/bitfields/tst.SizeofBitField.d mode=0444
430 file path=opt/SUNWdtrt/tst/common/buffering/err.end.d mode=0444
431 file path=opt/SUNWdtrt/tst/common/buffering/err.resize1.d mode=0444
432 file path=opt/SUNWdtrt/tst/common/buffering/err.resize2.d mode=0444
433 file path=opt/SUNWdtrt/tst/common/buffering/err.resize3.d mode=0444
434 file path=opt/SUNWdtrt/tst/common/buffering/err.zerobuf.d mode=0444
435 file path=opt/SUNWdtrt/tst/common/buffering/tst.alignring.d mode=0444
436 file path=opt/SUNWdtrt/tst/common/buffering/tst.cputime.ksh mode=0444
437 file path=opt/SUNWdtrt/tst/common/buffering/tst.dynvarsize.d mode=0444
438 file path=opt/SUNWdtrt/tst/common/buffering/tst.fill1.d mode=0444
439 file path=opt/SUNWdtrt/tst/common/buffering/tst.fill1.d.out mode=0444
440 file path=opt/SUNWdtrt/tst/common/buffering/tst.resize1.d mode=0444
441 file path=opt/SUNWdtrt/tst/common/buffering/tst.resize2.d mode=0444
442 file path=opt/SUNWdtrt/tst/common/buffering/tst.resize3.d mode=0444
443 file path=opt/SUNWdtrt/tst/common/buffering/tst.ring1.d mode=0444
444 file path=opt/SUNWdtrt/tst/common/buffering/tst.ring2.d mode=0444
445 file path=opt/SUNWdtrt/tst/common/buffering/tst.ring2.d.out mode=0444
446 file path=opt/SUNWdtrt/tst/common/buffering/tst.ring3.d mode=0444
447 file path=opt/SUNWdtrt/tst/common/buffering/tst.ring3.d.out mode=0444
448 file path=opt/SUNWdtrt/tst/common/buffering/tst.smallring.d mode=0444
449 file path=opt/SUNWdtrt/tst/common/buffering/tst.switch1.d mode=0444
450 file path=opt/SUNWdtrt/tst/common/buffering/tst.switch1.d.out mode=0444
451 file path=opt/SUNWdtrt/tst/common/builtinvar/err.D_XLATE_NOCONV.cpuusage.d \
452     mode=0444
453 file path=opt/SUNWdtrt/tst/common/builtinvar/err.D_XLATE_NOCONV.nice.d \
```



```
454     mode=0444
455 file path=opt/SUNWdtrt/tst/common/builtinvar/err.D_XLATE_NOCONV.priority.d \
456     mode=0444
457 file path=opt/SUNWdtrt/tst/common/builtinvar/err.D_XLATE_NOCONV.prsize.d \
458     mode=0444
459 file path=opt/SUNWdtrt/tst/common/builtinvar/err.D_XLATE_NOCONV.rssize.d \
460     mode=0444
461 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.arg0.d mode=0444
462 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.arg0clause.d mode=0444
463 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.arg1.d mode=0444
464 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.arg1to8.d mode=0444
465 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.arg1to8clause.d mode=0444
466 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.caller.d mode=0444
467 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.caller1.d mode=0444
468 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.epid.d mode=0444
469 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.epid1.d mode=0444
470 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.errno.d mode=0444
471 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.errno1.d mode=0444
472 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.execname.d mode=0444
473 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.hpriority.d mode=0444
474 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.id.d mode=0444
475 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.id1.d mode=0444
476 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.ipl.d mode=0444
477 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.ipl1.d mode=0444
478 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.lwpsinfo.d mode=0444
479 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.lwpsinfo1.d mode=0444
480 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.pid.d mode=0444
481 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.pid1.d mode=0444
482 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.psinfo.d mode=0444
483 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.psinfo1.d mode=0444
484 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.tid.d mode=0444
485 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.tid1.d mode=0444
486 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.timestamp.d mode=0444
487 file path=opt/SUNWdtrt/tst/common/builtinvar/tst.vtimestamp.d mode=0444
488 file path=opt/SUNWdtrt/tst/common/cg/err.D_NOREG.noreg.d mode=0444
489 file path=opt/SUNWdtrt/tst/common/cg/err.baddif.d mode=0444
490 file path=opt/SUNWdtrt/tst/common/clauses/err.D_IDENT_UNDEF.aggfun.d mode=0444
491 file path=opt/SUNWdtrt/tst/common/clauses/err.D_IDENT_UNDEF.aggtup.d mode=0444
492 file path=opt/SUNWdtrt/tst/common/clauses/err.D_IDENT_UNDEF.arrtup.d mode=0444
493 file path=opt/SUNWdtrt/tst/common/clauses/err.D_IDENT_UNDEF.body.d mode=0444
494 file path=opt/SUNWdtrt/tst/common/clauses/err.D_IDENT_UNDEF.both.d mode=0444
495 file path=opt/SUNWdtrt/tst/common/clauses/err.D_IDENT_UNDEF.pred.d mode=0444
496 file path=opt/SUNWdtrt/tst/common/clauses/tst.nopred.d mode=0444
497 file path=opt/SUNWdtrt/tst/common/clauses/tst.pred.d mode=0444
498 file path=opt/SUNWdtrt/tst/common/clauses/tst.predfirst.d mode=0444
499 file path=opt/SUNWdtrt/tst/common/clauses/tst.predlast.d mode=0444
500 file path=opt/SUNWdtrt/tst/common/cpc/err.D_PDESC_ZERO.lowfrequency.d \
501     mode=0444
502 file path=opt/SUNWdtrt/tst/common/cpc/err.D_PDESC_ZERO.malformedoverflow.d \
503     mode=0444
504 file path=opt/SUNWdtrt/tst/common/cpc/err.D_PDESC_ZERO.nonexistentevent.d \
505     mode=0444
506 file path=opt/SUNWdtrt/tst/common/cpc/err.cpcvscpustatpart1.ksh mode=0444
507 file path=opt/SUNWdtrt/tst/common/cpc/err.cpcvscpustatpart2.ksh mode=0444
508 file path=opt/SUNWdtrt/tst/common/cpc/err.cputrackfailtostart.ksh mode=0444
509 file path=opt/SUNWdtrt/tst/common/cpc/err.cputrackterminates.ksh mode=0444
510 file path=opt/SUNWdtrt/tst/common/cpc/err.toomanyenablings.d mode=0444
511 file path=opt/SUNWdtrt/tst/common/cpc/tst.allcpus.ksh mode=0444
512 file path=opt/SUNWdtrt/tst/common/cpc/tst.genericevent.d mode=0444
513 file path=opt/SUNWdtrt/tst/common/cpc/tst.platformevent.ksh mode=0444
514 file path=opt/SUNWdtrt/tst/common/decls/err.D_DECL_LOCASSC.NonLocalAssoc.d \
515     mode=0444
516 file path=opt/SUNWdtrt/tst/common/decls/err.D_DECL_LONGINT.LongStruct.d \
517     mode=0444
518 file path=opt/SUNWdtrt/tst/common/decls/err.D_DECL_PARMCLASS.BadStorageClass.d \
519     mode=0444
```

```
file path=opt/SUNWdtrt/tst/common/decls/err.D_DECL_PROTO_NAME.VoidName.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/decls/err.D_DECL_PROTO_TYPE.Dyn.d mode=0444
file path=opt/SUNWdtrt/tst/common/decls/err.D_DECL_PROTO_VARARGS.VarLenArgs.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/decls/err.D_DECL_PROTO_VOID.NonSoleVoid.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/decls/err.D_DECL_SIGNINT.UnsignedStruct.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/decls/err.D_DECL_VOIDATTR.ShortVoidDecl.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/decls/tst.arrays.d mode=0444
file path=opt/SUNWdtrt/tst/common/decls/tst.basics.d mode=0444
file path=opt/SUNWdtrt/tst/common/decls/tst.funcs.d mode=0444
file path=opt/SUNWdtrt/tst/common/decls/tst.pointers.d mode=0444
file path=opt/SUNWdtrt/tst/common/decls/tst.varargsfuncs.d mode=0444
file path=opt/SUNWdtrt/tst/common/drops/drp.DTRACEDROP_AGGREGATION.d mode=0444
file path=opt/SUNWdtrt/tst/common/drops/drp.DTRACEDROP_DBLERROR.d mode=0444
file path=opt/SUNWdtrt/tst/common/drops/drp.DTRACEDROP_DYNAMIC.d mode=0444
file path=opt/SUNWdtrt/tst/common/drops/drp.DTRACEDROP_PRINCIPAL.d mode=0444
file path=opt/SUNWdtrt/tst/common/drops/drp.DTRACEDROP_PRINCIPAL.end.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/drops/drp.DTRACEDROP_SPEC.d mode=0444
file path=opt/SUNWdtrt/tst/common/drops/drp.DTRACEDROP_SPECUNAVAIL.d mode=0444
file path=opt/SUNWdtrt/tst/common/drops/drp.DTRACEDROP_STKSTROVERFLOW.d \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/dtraceUtil/err.D_PDESC_ZERO.InvalidDescription1
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.AddSearchPath.d.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.BufsizeGiga.d.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.BufsizeKilo.d.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.BufsizeMega.d.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.BufsizeTera.d.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.DataModel32.d.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.DataModel64.d.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.DefineNameWithCPP.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.DefineNameWithCPP.d.ksh.out \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.DestructWithFunction.d.ksh \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.DestructWithFunction.d.ksh.out \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.DestructWithID.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.DestructWithID.d.ksh.out \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.DestructWithModule.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.DestructWithModule.d.ksh.out \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.DestructWithName.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.DestructWithName.d.ksh.out \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.DestructWithProvider.d.ksh \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.DestructWithProvider.d.ksh.out \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.DestructWithoutW.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ELFGenerationOut.d.ksh \
    mode=0444
```



```
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ELFGenerationWithO.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ExitStatus1.d.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ExitStatus2.d.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ExtraneousProbeIds.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidFuncName1.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidFuncName2.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidId1.d.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidId2.d.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidId3.d.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidModule1.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidModule2.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidModule3.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidModule4.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidProbeIdentifier.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidProvider1.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidProvider2.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidProvider3.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidProvider4.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceFunc1.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceFunc2.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceFunc3.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceFunc4.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceFunc5.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceFunc6.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceFunc7.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceFunc8.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceFunc9.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceID1.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceID2.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceID3.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceID4.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceID5.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceID6.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceID7.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceModule1.d.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceModule2.d.ksh \
```

```
652     mode=0444
653 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceModule3.d.ksh \
654     mode=0444
655 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceModule4.d.ksh \
656     mode=0444
657 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceModule5.d.ksh \
658     mode=0444
659 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceModule6.d.ksh \
660     mode=0444
661 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceModule7.d.ksh \
662     mode=0444
663 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceModule8.d.ksh \
664     mode=0444
665 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName1.d.ksh \
666     mode=0444
667 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName2.d.ksh \
668     mode=0444
669 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName3.d.ksh \
670     mode=0444
671 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName4.d.ksh \
672     mode=0444
673 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName5.d.ksh \
674     mode=0444
675 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName6.d.ksh \
676     mode=0444
677 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName7.d.ksh \
678     mode=0444
679 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName8.d.ksh \
680     mode=0444
681 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceName9.d.ksh \
682     mode=0444
683 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceProvider1.d.ksh \
684     mode=0444
685 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceProvider2.d.ksh \
686     mode=0444
687 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceProvider3.d.ksh \
688     mode=0444
689 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceProvider4.d.ksh \
690     mode=0444
691 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.InvalidTraceProvider5.d.ksh \
692     mode=0444
693 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.MultipleInvalidProbeId.d.ksh \
694     mode=0444
695 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.PreprocessorStatement.d.ksh \
696     mode=0444
697 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.QuietMode.d.ksh mode=0444
698 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.QuietMode.d.ksh.out mode=0444
699 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.TestCompile.d.ksh mode=0444
700 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.TestCompile.d.ksh.out \
701     mode=0444
702 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.UnDefineNameWithCPP.d.ksh \
703     mode=0444
704 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ZeroFunctionProbes.d.ksh \
705     mode=0444
706 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ZeroFunctionProbes.d.ksh.out \
707     mode=0444
708 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ZeroModuleProbes.d.ksh \
709     mode=0444
710 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ZeroModuleProbes.d.ksh.out \
711     mode=0444
712 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ZeroNameProbes.d.ksh \
713     mode=0444
714 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ZeroNameProbes.d.ksh.out \
715     mode=0444
716 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ZeroProbeIdentfier.d.ksh \
717     mode=0444
```



```
718 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ZeroProbesWithoutZ.d.ksh \
719     mode=0444
720 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ZeroProviderProbes.d.ksh \
721     mode=0444
722 file path=opt/SUNWdtrt/tst/common/dtraceUtil/tst.ZeroProviderProbes.d.ksh.out \
723     mode=0444
724 file path=opt/SUNWdtrt/tst/common/end/err.D_IDENT_UNDEF.timespent.d mode=0444
725 file path=opt/SUNWdtrt/tst/common/end/tst.end.d mode=0444
726 file path=opt/SUNWdtrt/tst/common/end/tst.endwithoutbegin.d mode=0444
727 file path=opt/SUNWdtrt/tst/common/end/tst.multibeginend.d mode=0444
728 file path=opt/SUNWdtrt/tst/common/end/tst.multiend.d mode=0444
729 file path=opt/SUNWdtrt/tst/common/enum/err.D_DECL_IDRED.EnumSameName.d \
730     mode=0444
731 file path=opt/SUNWdtrt/tst/common/enum/err.D_UNKNOWN.RepeatIdentifiers.d \
732     mode=0444
733 file path=opt/SUNWdtrt/tst/common/enum/tst.EnumEquality.d mode=0444
734 file path=opt/SUNWdtrt/tst/common/enum/tst.EnumSameValue.d mode=0444
735 file path=opt/SUNWdtrt/tst/common/enum/tst.EnumValAssign.d mode=0444
736 file path=opt/SUNWdtrt/tst/common/env/err.D_PRAGMA_OPTSET.setfromscript.d \
737     mode=0444
738 file path=opt/SUNWdtrt/tst/common/env/err.D_PRAGMA_OPTSET.unsetfromscript.d \
739     mode=0444
740 file path=opt/SUNWdtrt/tst/common/env/tst.ld_nolazyload.ksh mode=0444
741 file path=opt/SUNWdtrt/tst/common/env/tst.ld_nolazyload.ksh.out mode=0444
742 file path=opt/SUNWdtrt/tst/common/env/tst.setenv1.ksh mode=0444
743 file path=opt/SUNWdtrt/tst/common/env/tst.setenv1.ksh.out mode=0444
744 file path=opt/SUNWdtrt/tst/common/env/tst.setenv2.ksh mode=0444
745 file path=opt/SUNWdtrt/tst/common/env/tst.setenv2.ksh.out mode=0444
746 file path=opt/SUNWdtrt/tst/common/env/tst.unsetenv1.ksh mode=0444
747 file path=opt/SUNWdtrt/tst/common/env/tst.unsetenv1.ksh.out mode=0444
748 file path=opt/SUNWdtrt/tst/common/env/tst.unsetenv2.ksh mode=0444
749 file path=opt/SUNWdtrt/tst/common/env/tst.unsetenv2.ksh.out mode=0444
750 file path=opt/SUNWdtrt/tst/common/error/tst.DTRACEFLT_BADADDR.d mode=0444
751 file path=opt/SUNWdtrt/tst/common/error/tst.DTRACEFLT_DIVZERO.d mode=0444
752 file path=opt/SUNWdtrt/tst/common/error/tst.DTRACEFLT_UNKNOWN.d mode=0444
753 file path=opt/SUNWdtrt/tst/common/error/tst.error.d mode=0444
754 file path=opt/SUNWdtrt/tst/common/error/tst.errorend.d mode=0444
755 file path=opt/SUNWdtrt/tst/common/exit/err.D_PROTO_LEN.noarg.d mode=0444
756 file path=opt/SUNWdtrt/tst/common/exit/err.exitarg1.d mode=0444
757 file path=opt/SUNWdtrt/tst/common/exit/tst.basic1.d mode=0444
758 file path=opt/SUNWdtrt/tst/common/fbtprovider/err.D_PDESC_ZERO.notreturn.d \
759     mode=0444
760 file path=opt/SUNWdtrt/tst/common/fbtprovider/tst.basic.d mode=0444
761 file path=opt/SUNWdtrt/tst/common/fbtprovider/tst.functionentry.d mode=0444
762 file path=opt/SUNWdtrt/tst/common/fbtprovider/tst.functionreturnvalue.d \
763     mode=0444
764 file path=opt/SUNWdtrt/tst/common/fbtprovider/tst.ioctlargs.d mode=0444
765 file path=opt/SUNWdtrt/tst/common/fbtprovider/tst.offset.d mode=0444
766 file path=opt/SUNWdtrt/tst/common/fbtprovider/tst.offsetzero.d mode=0444
767 file path=opt/SUNWdtrt/tst/common/fbtprovider/tst.return.d mode=0444
768 file path=opt/SUNWdtrt/tst/common/fbtprovider/tst.return0.d mode=0444
769 file path=opt/SUNWdtrt/tst/common/fbtprovider/tst.tailcall.d mode=0444
770 file path=opt/SUNWdtrt/tst/common/funcs/err.D_FUNC_UNDEF.progenyofbad1.d \
771     mode=0444
772 file path=opt/SUNWdtrt/tst/common/funcs/err.D_OP_VFPTR.badop.d mode=0444
773 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_ARG.chillbadarg.d \
774     mode=0444
775 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_ARG.copyoutbadarg.d \
776     mode=0444
777 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_ARG.mobadarg.d mode=0444
778 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_ARG.raisebadarg.d \
779     mode=0444
780 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_ARG.tolower.d mode=0444
781 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_ARG.toupper.d mode=0444
782 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.allocanoarg.d \
783     mode=0444
```

```
784 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.badbreakpoint.d \
785     mode=0444
786 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.chilltoofew.d \
787     mode=0444
788 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.chilltoomany.d \
789     mode=0444
790 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.copyoutstrbadarg.d \
791     mode=0444
792 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.copyoutstrtoofew.d \
793     mode=0444
794 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.copyouttoofew.d \
795     mode=0444
796 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.copyouttoomany.d \
797     mode=0444
798 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.motoofew.d mode=0444
799 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.motoomany.d mode=0444
800 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.mtabadarg.d mode=0444
801 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.mtatoofew.d mode=0444
802 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.mtatoomany.d mode=0444
803 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.panicbadarg.d \
804     mode=0444
805 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.progenyofbad2.d \
806     mode=0444
807 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.stopbadarg.d mode=0444
808 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.tolower.d mode=0444
809 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.tolowertoomany.d \
810     mode=0444
811 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.toupper.d mode=0444
812 file path=opt/SUNWdtrt/tst/common/funcs/err.D_PROTO_LEN.touppertoomany.d \
813     mode=0444
814 file path=opt/SUNWdtrt/tst/common/funcs/err.D_STRINGOF_TYPE.badstringof.d \
815     mode=0444
816 file path=opt/SUNWdtrt/tst/common/funcs/err.D_VAR_UNDEF.badvar.d mode=0444
817 file path=opt/SUNWdtrt/tst/common/funcs/err.badalloca.d mode=0444
818 file path=opt/SUNWdtrt/tst/common/funcs/err.badalloca2.d mode=0444
819 file path=opt/SUNWdtrt/tst/common/funcs/err.badbcopy.d mode=0444
820 file path=opt/SUNWdtrt/tst/common/funcs/err.badbcopy1.d mode=0444
821 file path=opt/SUNWdtrt/tst/common/funcs/err.badbcopy2.d mode=0444
822 file path=opt/SUNWdtrt/tst/common/funcs/err.badbcopy3.d mode=0444
823 file path=opt/SUNWdtrt/tst/common/funcs/err.badbcopy4.d mode=0444
824 file path=opt/SUNWdtrt/tst/common/funcs/err.badbcopy5.d mode=0444
825 file path=opt/SUNWdtrt/tst/common/funcs/err.badbcopy6.d mode=0444
826 file path=opt/SUNWdtrt/tst/common/funcs/err.badchill.d mode=0444
827 file path=opt/SUNWdtrt/tst/common/funcs/err.chillbadarg.ksh mode=0444
828 file path=opt/SUNWdtrt/tst/common/funcs/err.copyout.d mode=0444
829 file path=opt/SUNWdtrt/tst/common/funcs/err.copyoutbadaddr.ksh mode=0444
830 file path=opt/SUNWdtrt/tst/common/funcs/err.copyoutstrbadaddr.ksh mode=0444
831 file path=opt/SUNWdtrt/tst/common/funcs/err.inet_ntoa6badaddr.d mode=0444
832 file path=opt/SUNWdtrt/tst/common/funcs/err.inet_ntoabadaddr.d mode=0444
833 file path=opt/SUNWdtrt/tst/common/funcs/err.inet_ntopbadaddr.d mode=0444
834 file path=opt/SUNWdtrt/tst/common/funcs/err.inet_ntopbadarg.d mode=0444
835 file path=opt/SUNWdtrt/tst/common/funcs/tst.badfreopen.ksh mode=0444
836 file path=opt/SUNWdtrt/tst/common/funcs/tst.basename.d mode=0444
837 file path=opt/SUNWdtrt/tst/common/funcs/tst.basename.d.out mode=0444
838 file path=opt/SUNWdtrt/tst/common/funcs/tst.bcopy.d mode=0444
839 file path=opt/SUNWdtrt/tst/common/funcs/tst.chill.ksh mode=0444
840 file path=opt/SUNWdtrt/tst/common/funcs/tst.cleanpath.d mode=0444
841 file path=opt/SUNWdtrt/tst/common/funcs/tst.cleanpath.d.out mode=0444
842 file path=opt/SUNWdtrt/tst/common/funcs/tst.copyin.d mode=0444
843 file path=opt/SUNWdtrt/tst/common/funcs/tst.copyinto.d mode=0444
844 file path=opt/SUNWdtrt/tst/common/funcs/tst.ddi_pathname.d mode=0444
845 file path=opt/SUNWdtrt/tst/common/funcs/tst.default.d mode=0444
846 file path=opt/SUNWdtrt/tst/common/funcs/tst.freopen.ksh mode=0444
847 file path=opt/SUNWdtrt/tst/common/funcs/tst.ftruncate.ksh mode=0444
848 file path=opt/SUNWdtrt/tst/common/funcs/tst.ftruncate.ksh.out mode=0444
849 file path=opt/SUNWdtrt/tst/common/funcs/tst.hton.d mode=0444
```



```
850 file path=opt/SUNWdtrt/tst/common/funcs/tst.index.d mode=0444
851 file path=opt/SUNWdtrt/tst/common/funcs/tst.index.d.out mode=0444
852 file path=opt/SUNWdtrt/tst/common/funcs/tst.inet_ntoa.d mode=0444
853 file path=opt/SUNWdtrt/tst/common/funcs/tst.inet_ntoa.d.out mode=0444
854 file path=opt/SUNWdtrt/tst/common/funcs/tst.inet_ntoa6.d mode=0444
855 file path=opt/SUNWdtrt/tst/common/funcs/tst.inet_ntoa6.d.out mode=0444
856 file path=opt/SUNWdtrt/tst/common/funcs/tst.inet_ntop.d mode=0444
857 file path=opt/SUNWdtrt/tst/common/funcs/tst.inet_ntop.d.out mode=0444
858 file path=opt/SUNWdtrt/tst/common/funcs/tst.lltostr.d mode=0444
859 file path=opt/SUNWdtrt/tst/common/funcs/tst.lltostr.d.out mode=0444
860 file path=opt/SUNWdtrt/tst/common/funcs/tst.lltostrbase.d mode=0444
861 file path=opt/SUNWdtrt/tst/common/funcs/tst.lltostrbase.d.out mode=0444
862 file path=opt/SUNWdtrt/tst/common/funcs/tst.mutex_owned.d mode=0444
863 file path=opt/SUNWdtrt/tst/common/funcs/tst.mutex_owner.d mode=0444
864 file path=opt/SUNWdtrt/tst/common/funcs/tst.mutex_type_adaptive.d mode=0444
865 file path=opt/SUNWdtrt/tst/common/funcs/tst.progenyof.d mode=0444
866 file path=opt/SUNWdtrt/tst/common/funcs/tst.rand.d mode=0444
867 file path=opt/SUNWdtrt/tst/common/funcs/tst.strchr.d mode=0444
868 file path=opt/SUNWdtrt/tst/common/funcs/tst.strchr.d.out mode=0444
869 file path=opt/SUNWdtrt/tst/common/funcs/tst.strjoin.d mode=0444
870 file path=opt/SUNWdtrt/tst/common/funcs/tst.strjoin.d.out mode=0444
871 file path=opt/SUNWdtrt/tst/common/funcs/tst.strstr.d mode=0444
872 file path=opt/SUNWdtrt/tst/common/funcs/tst.strstr.d.out mode=0444
873 file path=opt/SUNWdtrt/tst/common/funcs/tst.strtok.d mode=0444
874 file path=opt/SUNWdtrt/tst/common/funcs/tst.strtok.d.out mode=0444
875 file path=opt/SUNWdtrt/tst/common/funcs/tst.strtok_null.d mode=0444
876 file path=opt/SUNWdtrt/tst/common/funcs/tst.substr.d mode=0444
877 file path=opt/SUNWdtrt/tst/common/funcs/tst.substr.d.out mode=0444
878 file path=opt/SUNWdtrt/tst/common/funcs/tst.substrminate.d mode=0444
879 file path=opt/SUNWdtrt/tst/common/funcs/tst.substrminate.d.out mode=0444
880 file path=opt/SUNWdtrt/tst/common/funcs/tst.system.d mode=0444
881 file path=opt/SUNWdtrt/tst/common/funcs/tst.system.d.out mode=0444
882 file path=opt/SUNWdtrt/tst/common/funcs/tst.tolower.d mode=0444
883 file path=opt/SUNWdtrt/tst/common/funcs/tst.toupper.d mode=0444
884 file path=opt/SUNWdtrt/tst/common/grammar/err.D_ADDROF_LVAL.d mode=0444
885 file path=opt/SUNWdtrt/tst/common/grammar/err.D_EMPTY.empty.d mode=0444
886 file path=opt/SUNWdtrt/tst/common/grammar/tst.clauses.d mode=0444
887 file path=opt/SUNWdtrt/tst/common/grammar/tst.stmts.d mode=0444
888 file path=opt/SUNWdtrt/tst/common/include/tst.includefirst.ksh mode=0444
889 file path=opt/SUNWdtrt/tst/common/inline/err.D_DECL_IDRED.redef1.d mode=0444
890 file path=opt/SUNWdtrt/tst/common/inline/err.D_DECL_IDRED.redef2.d mode=0444
891 file path=opt/SUNWdtrt/tst/common/inline/err.D_IDENT_UNDEF.recur.d mode=0444
892 file path=opt/SUNWdtrt/tst/common/inline/err.D_OP_INCOMPAT.baddef1.d mode=0444
893 file path=opt/SUNWdtrt/tst/common/inline/err.D_OP_INCOMPAT.baddef2.d mode=0444
894 file path=opt/SUNWdtrt/tst/common/inline/err.D_OP_INCOMPAT.badxlate.d \
895     mode=0444
896 file path=opt/SUNWdtrt/tst/common/inline/tst.InlineDataAssign.d mode=0444
897 file path=opt/SUNWdtrt/tst/common/inline/tst.InlineExpression.d mode=0444
898 file path=opt/SUNWdtrt/tst/common/inline/tst.InlineKinds.d mode=0444
899 file path=opt/SUNWdtrt/tst/common/inline/tst.InlineKinds.d.out mode=0444
900 file path=opt/SUNWdtrt/tst/common/inline/tst.InlineTypedef.d mode=0444
901 file path=opt/SUNWdtrt/tst/common/inline/tst.InlineWritableAssign.d mode=0444
902 file path=opt/SUNWdtrt/tst/common/io/tst.fds.d mode=0444
903 file path=opt/SUNWdtrt/tst/common/io/tst.fds.d.out mode=0444
904 file path=opt/SUNWdtrt/tst/common/io/tst.fds.exe mode=0555
905 file path=opt/SUNWdtrt/tst/common/ip/get.ipv4remote.pl mode=0555
906 file path=opt/SUNWdtrt/tst/common/ip/get.ipv6remote.pl mode=0555
907 file path=opt/SUNWdtrt/tst/common/ip/tst.ipv4localicmp.ksh mode=0444
908 file path=opt/SUNWdtrt/tst/common/ip/tst.ipv4localicmp.ksh.out mode=0444
909 file path=opt/SUNWdtrt/tst/common/ip/tst.ipv4localtcp.ksh mode=0444
910 file path=opt/SUNWdtrt/tst/common/ip/tst.ipv4localtcp.ksh.out mode=0444
911 file path=opt/SUNWdtrt/tst/common/ip/tst.ipv4localudp.ksh mode=0444
912 file path=opt/SUNWdtrt/tst/common/ip/tst.ipv4localudp.ksh.out mode=0444
913 file path=opt/SUNWdtrt/tst/common/ip/tst.ipv4remoteicmp.ksh mode=0444
914 file path=opt/SUNWdtrt/tst/common/ip/tst.ipv4remoteicmp.ksh.out mode=0444
915 file path=opt/SUNWdtrt/tst/common/ip/tst.ipv4remotetcp.ksh mode=0444
```

```
file path=opt/SUNWdtrt/tst/common/ip/tst.ipv4remotetcp.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.ipv4remoteudp.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.ipv4remoteudp.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.ipv6localicmp.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.ipv6localicmp.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.ipv6remoteicmp.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.ipv6remoteicmp.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.localtcpstate.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.localtcpstate.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.remotetcpstate.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/ip/tst.remotetcpstate.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/test.jar
file path=opt/SUNWdtrt/tst/common/java_api/tst.Abort.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Abort.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Bean.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Bean.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Close.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Close.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Drop.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Drop.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Enable.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.Enable.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.FunctionLookup.exe mode=0555
file path=opt/SUNWdtrt/tst/common/java_api/tst.FunctionLookup.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.FunctionLookup.ksh.out \
    mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.GetAggregate.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.MaxConsumers.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.MaxConsumers.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.MultiAggPrinta.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.MultiAggPrinta.ksh.out \
    mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.ProbeData.exe mode=0555
file path=opt/SUNWdtrt/tst/common/java_api/tst.ProbeData.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.ProbeData.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.ProbeDescription.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.ProbeDescription.ksh.out \
    mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.StateMachine.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.StateMachine.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.StopLock.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.StopLock.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.printa.d mode=0444
file path=opt/SUNWdtrt/tst/common/java_api/tst.printa.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_CHR_NL.char.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_CHR_NULL.char.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_INT_DIGIT.InvalidDigit.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_INT_OFLOW.BigInt.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_STR_NL.string.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_SYNTAX.brace1.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_SYNTAX.brace2.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_SYNTAX.brack1.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_SYNTAX.brack2.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_SYNTAX.brack3.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_SYNTAX.paren1.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_SYNTAX.paren2.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/err.D_SYNTAX.paren3.d mode=0444
file path=opt/SUNWdtrt/tst/common/lexer/tst.D_MACRO_OFLOW.ParIntOvflow.d.ksh \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_FACTOREVEN.nodivide.d
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_FACTOREVEN.notfactor.d
    mode=0444
```



```
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_FACTORMATCH.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_FACTORNSTEPS.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_FACTORSMALL.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_FACTORTYPE.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_FACTORVAL.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_HIGHMATCH.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_HIGHTYPE.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_HIGHVAL.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_LOWMATCH.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_LOWTYPE.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_LOWVAL.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_MAGRANGE.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_MAGTOOBIG.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_NSTEPMATCH.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_NSTEPTYPE.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/err.D_LLQUANT_NSTEPVAL.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.bases.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.bases.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.basic.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.basic.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.negorder.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.negorder.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.negvalue.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.negvalue.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.normal.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.normal.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.range.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.range.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.steps.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.steps.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.trunc.d mode=0444
file path=opt/SUNWdtrt/tst/common/llquantize/tst.trunc.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/mdb/tst.dtracedcmd.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/mib/tst.icmp.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/mib/tst.tcp.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/mib/tst.udp.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/misc/err.D_PRAGMA_OPTSET.d mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.badopt.d mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.boolopt.d mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.boolopt.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.dynopt.d mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.dynopt.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.enablerace.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.haslam.d mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.include.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.macroglob.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.macroglob.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.roch.d mode=0444
file path=opt/SUNWdtrt/tst/common/misc/tst.schrock.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/err.D_PRINTA_AGGKEY.d mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/err.D_PRINTA_AGGPROTO.d mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.many.d mode=0444
file path=opt/SUNWdtrt/tst/common/multiaggs/tst.many.d.out mode=0444
```

```
1048 file path=opt/SUNWdtrt/tst/common/multiaggs/tst.same.d mode=0444
1049 file path=opt/SUNWdtrt/tst/common/multiaggs/tst.same.d.out mode=0444
1050 file path=opt/SUNWdtrt/tst/common/multiaggs/tst.sort.d mode=0444
1051 file path=opt/SUNWdtrt/tst/common/multiaggs/tst.sort.d.out mode=0444
1052 file path=opt/SUNWdtrt/tst/common/multiaggs/tst.sortpos.d mode=0444
1053 file path=opt/SUNWdtrt/tst/common/multiaggs/tst.sortpos.d.out mode=0444
1054 file path=opt/SUNWdtrt/tst/common/multiaggs/tst.tuplecompat.d mode=0444
1055 file path=opt/SUNWdtrt/tst/common/multiaggs/tst.tuplecompat.d.out mode=0444
1056 file path=opt/SUNWdtrt/tst/common/multiaggs/tst.zero.d mode=0444
1057 file path=opt/SUNWdtrt/tst/common/multiaggs/tst.zero.d.out mode=0444
1058 file path=opt/SUNWdtrt/tst/common/multiaggs/tst.zero2.d mode=0444
1059 file path=opt/SUNWdtrt/tst/common/multiaggs/tst.zero2.d.out mode=0444
1060 file path=opt/SUNWdtrt/tst/common/multiaggs/tst.zero3.d mode=0444
1061 file path=opt/SUNWdtrt/tst/common/multiaggs/tst.zero3.d.out mode=0444
1062 file path=opt/SUNWdtrt/tst/common/nfs/tst.call.d mode=0444
1063 file path=opt/SUNWdtrt/tst/common/nfs/tst.call.exe mode=0555
1064 file path=opt/SUNWdtrt/tst/common/nfs/tst.call3.d mode=0444
1065 file path=opt/SUNWdtrt/tst/common/nfs/tst.call3.exe mode=0555
1066 file path=opt/SUNWdtrt/tst/common/offsetof/err.D_OFFSETOF_BITFIELD.bitfield.d \
1067     mode=0444
1068 file path=opt/SUNWdtrt/tst/common/offsetof/err.D_OFFSETOF_TYPE.badtype.d \
1069     mode=0444
1070 file path=opt/SUNWdtrt/tst/common/offsetof/err.D_OFFSETOF_TYPE.notsou.d \
1071     mode=0444
1072 file path=opt/SUNWdtrt/tst/common/offsetof/err.D_UNKNOWN.OffsetofNULL.d \
1073     mode=0444
1074 file path=opt/SUNWdtrt/tst/common/offsetof/err.D_UNKNOWN.badmemb.d mode=0444
1075 file path=opt/SUNWdtrt/tst/common/offsetof/tst.OffsetofAlias.d mode=0444
1076 file path=opt/SUNWdtrt/tst/common/offsetof/tst.OffsetofArith.d mode=0444
1077 file path=opt/SUNWdtrt/tst/common/offsetof/tst.OffsetofUnion.d mode=0444
1078 file path=opt/SUNWdtrt/tst/common/offsetof/tst.struct.d mode=0444
1079 file path=opt/SUNWdtrt/tst/common/offsetof/tst.struct.d.out mode=0444
1080 file path=opt/SUNWdtrt/tst/common/offsetof/tst.union.d mode=0444
1081 file path=opt/SUNWdtrt/tst/common/offsetof/tst.union.d.out mode=0444
1082 file path=opt/SUNWdtrt/tst/common/operators/tst.ternary.d mode=0444
1083 file path=opt/SUNWdtrt/tst/common/operators/tst.ternary.d.out mode=0444
1084 file path=opt/SUNWdtrt/tst/common/pid/err.D_PDESC_ZERO.badlib.d mode=0444
1085 file path=opt/SUNWdtrt/tst/common/pid/err.D_PDESC_ZERO.badlib.exe mode=0555
1086 file path=opt/SUNWdtrt/tst/common/pid/err.D_PDESC_ZERO.badproc1.d mode=0444
1087 file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_BADPID.badproc2.d mode=0444
1088 file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_CREATEFAIL.many.d mode=0444
1089 file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_CREATEFAIL.many.exe mode=0555
1090 file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_FUNC.badfunc.d mode=0444
1091 file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_FUNC.badfunc.exe mode=0555
1092 file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_LIB.libdash.d mode=0444
1093 file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_LIB.libdash.exe mode=0555
1094 file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_NAME.alldash.d mode=0444
1095 file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_NAME.alldash.exe mode=0555
1096 file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_NAME.badname.d mode=0444
1097 file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_NAME.badname.exe mode=0555
1098 file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_NAME.globdash.d mode=0444
1099 file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_NAME.globdash.exe mode=0555
1100 file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_OFF.toobig.d mode=0444
1101 file path=opt/SUNWdtrt/tst/common/pid/err.D_PROC_OFF.toobig.exe mode=0555
1102 file path=opt/SUNWdtrt/tst/common/pid/tst.addprobes.ksh mode=0444
1103 file path=opt/SUNWdtrt/tst/common/pid/tst.args1.d mode=0444
1104 file path=opt/SUNWdtrt/tst/common/pid/tst.args1.exe mode=0555
1105 file path=opt/SUNWdtrt/tst/common/pid/tst.coverage.d mode=0444
1106 file path=opt/SUNWdtrt/tst/common/pid/tst.coverage.exe mode=0555
1107 file path=opt/SUNWdtrt/tst/common/pid/tst.emptystack.d mode=0444
1108 file path=opt/SUNWdtrt/tst/common/pid/tst.emptystack.d.out mode=0444
1109 file path=opt/SUNWdtrt/tst/common/pid/tst.emptystack.exe mode=0555
1110 file path=opt/SUNWdtrt/tst/common/pid/tst.float.d mode=0444
1111 file path=opt/SUNWdtrt/tst/common/pid/tst.float.exe mode=0555
1112 file path=opt/SUNWdtrt/tst/common/pid/tst.fork.d mode=0444
1113 file path=opt/SUNWdtrt/tst/common/pid/tst.fork.exe mode=0555
```



```
1114 file path=opt/SUNWdtrt/tst/common/pid/tst.gcc.d mode=0444
1115 file path=opt/SUNWdtrt/tst/common/pid/tst.gcc.exe mode=0555
1116 file path=opt/SUNWdtrt/tst/common/pid/tst.killonerror.ksh mode=0444
1117 file path=opt/SUNWdtrt/tst/common/pid/tst.main.ksh mode=0444
1118 file path=opt/SUNWdtrt/tst/common/pid/tst.manypids.ksh mode=0444
1119 file path=opt/SUNWdtrt/tst/common/pid/tst.newprobes.ksh mode=0444
1120 file path=opt/SUNWdtrt/tst/common/pid/tst.newprobes.ksh.out mode=0444
1121 file path=opt/SUNWdtrt/tst/common/pid/tst.probemod.ksh mode=0444
1122 file path=opt/SUNWdtrt/tst/common/pid/tst.provregex1.ksh mode=0444
1123 file path=opt/SUNWdtrt/tst/common/pid/tst.provregex2.ksh mode=0444
1124 file path=opt/SUNWdtrt/tst/common/pid/tst.provregex2.ksh.out mode=0444
1125 file path=opt/SUNWdtrt/tst/common/pid/tst.provregex3.ksh mode=0444
1126 file path=opt/SUNWdtrt/tst/common/pid/tst.provregex3.ksh.out mode=0444
1127 file path=opt/SUNWdtrt/tst/common/pid/tst.provregex4.ksh mode=0444
1128 file path=opt/SUNWdtrt/tst/common/pid/tst.provregex4.ksh.out mode=0444
1129 file path=opt/SUNWdtrt/tst/common/pid/tst.ret1.d mode=0444
1130 file path=opt/SUNWdtrt/tst/common/pid/tst.ret1.exe mode=0555
1131 file path=opt/SUNWdtrt/tst/common/pid/tst.ret2.d mode=0444
1132 file path=opt/SUNWdtrt/tst/common/pid/tst.ret2.exe mode=0555
1133 file path=opt/SUNWdtrt/tst/common/pid/tst.utf8probefunc.ksh mode=0444
1134 file path=opt/SUNWdtrt/tst/common/pid/tst.utf8probefunc.ksh.out mode=0444
1135 file path=opt/SUNWdtrt/tst/common/pid/tst.utf8probemod.ksh mode=0444
1136 file path=opt/SUNWdtrt/tst/common/pid/tst.utf8probemod.ksh.out mode=0444
1137 file path=opt/SUNWdtrt/tst/common/pid/tst.vfork.d mode=0444
1138 file path=opt/SUNWdtrt/tst/common/pid/tst.vfork.exe mode=0555
1139 file path=opt/SUNWdtrt/tst/common/pid/tst.weak1.d mode=0444
1140 file path=opt/SUNWdtrt/tst/common/pid/tst.weak1.exe mode=0555
1141 file path=opt/SUNWdtrt/tst/common/pid/tst.weak2.d mode=0444
1142 file path=opt/SUNWdtrt/tst/common/pid/tst.weak2.exe mode=0555
1143 file path=opt/SUNWdtrt/tst/common/plockstat/tst.available.d mode=0444
1144 file path=opt/SUNWdtrt/tst/common/plockstat/tst.available.exe mode=0555
1145 file path=opt/SUNWdtrt/tst/common/plockstat/tst.libmap.d mode=0444
1146 file path=opt/SUNWdtrt/tst/common/plockstat/tst.libmap.exe mode=0555
1147 file path=opt/SUNWdtrt/tst/common/pointers/err.BadAlign.d mode=0444
1148 file path=opt/SUNWdtrt/tst/common/pointers/err.D_ADDROF_VAR.ArrayVar.d \
1149     mode=0444
1150 file path=opt/SUNWdtrt/tst/common/pointers/err.D_ADDROF_VAR.DynamicVar.d \
1151     mode=0444
1152 file path=opt/SUNWdtrt/tst/common/pointers/err.D_ADDROF_VAR.agg.d mode=0444
1153 file path=opt/SUNWdtrt/tst/common/pointers/err.D_DEREF_NONPTR.noptr.d \
1154     mode=0444
1155 file path=opt/SUNWdtrt/tst/common/pointers/err.D_DEREF_VOID.VoidPointerDeref.d \
1156     mode=0444
1157 file path=opt/SUNWdtrt/tst/common/pointers/err.D_OP_ARRFUN.ArrayAssignment.d \
1158     mode=0444
1159 file \
1160     path=opt/SUNWdtrt/tst/common/pointers/err.D_OP_INCOMPAT.VoidPointerArith.d \
1161     mode=0444
1162 file path=opt/SUNWdtrt/tst/common/pointers/err.D_OP_LVAL.AddressChange.d \
1163     mode=0444
1164 file path=opt/SUNWdtrt/tst/common/pointers/err.D_OP_PTR.NonPointerAccess.d \
1165     mode=0444
1166 file path=opt/SUNWdtrt/tst/common/pointers/err.D_OP_PTR.badpointer.d mode=0444
1167 file path=opt/SUNWdtrt/tst/common/pointers/err.D_OP_SOU.BadPointerAccess.d \
1168     mode=0444
1169 file path=opt/SUNWdtrt/tst/common/pointers/err.D_OP_SOU.badpointer.d mode=0444
1170 file path=opt/SUNWdtrt/tst/common/pointers/err.InvalidAddress1.d mode=0444
1171 file path=opt/SUNWdtrt/tst/common/pointers/err.InvalidAddress2.d mode=0444
1172 file path=opt/SUNWdtrt/tst/common/pointers/err.InvalidAddress3.d mode=0444
1173 file path=opt/SUNWdtrt/tst/common/pointers/err.InvalidAddress4.d mode=0444
1174 file path=opt/SUNWdtrt/tst/common/pointers/err.InvalidAddress5.d mode=0444
1175 file path=opt/SUNWdtrt/tst/common/pointers/tst.ArrayPointer1.d mode=0444
1176 file path=opt/SUNWdtrt/tst/common/pointers/tst.ArrayPointer2.d mode=0444
1177 file path=opt/SUNWdtrt/tst/common/pointers/tst.ArrayPointer3.d mode=0444
1178 file path=opt/SUNWdtrt/tst/common/pointers/tst.GlobalVar.d mode=0444
1179 file path=opt/SUNWdtrt/tst/common/pointers/tst.IntegerArithmetic1.d mode=0444
```

```
1180 file path=opt/SUNWdtrt/tst/common/pointers/tst.PointerArithmetic1.d mode=0444
1181 file path=opt/SUNWdtrt/tst/common/pointers/tst.PointerArithmetic2.d mode=0444
1182 file path=opt/SUNWdtrt/tst/common/pointers/tst.PointerArithmetic3.d mode=0444
1183 file path=opt/SUNWdtrt/tst/common/pointers/tst.PointerAssignment.d mode=0444
1184 file path=opt/SUNWdtrt/tst/common/pointers/tst.ValidPointer1.d mode=0444
1185 file path=opt/SUNWdtrt/tst/common/pointers/tst.ValidPointer2.d mode=0444
1186 file path=opt/SUNWdtrt/tst/common/pointers/tst.VoidCast.d mode=0444
1187 file path=opt/SUNWdtrt/tst/common/pointers/tst.assigncast1.d mode=0444
1188 file path=opt/SUNWdtrt/tst/common/pointers/tst.assigncast2.d mode=0444
1189 file path=opt/SUNWdtrt/tst/common/pointers/tst.basic1.d mode=0444
1190 file path=opt/SUNWdtrt/tst/common/pointers/tst.basic2.d mode=0444
1191 file path=opt/SUNWdtrt/tst/common/pragma/err.D_PRAGERR.d mode=0444
1192 file path=opt/SUNWdtrt/tst/common/pragma/err.D_PRAGMA_DEPEND.main.d mode=0444
1193 file path=opt/SUNWdtrt/tst/common/pragma/err.D_PRAGMA_INVAL.d mode=0444
1194 file path=opt/SUNWdtrt/tst/common/pragma/err.D_PRAGMA_MALFORM.d mode=0444
1195 file path=opt/SUNWdtrt/tst/common/pragma/err.D_PRAGMA_UNUSED.UnusedPragma.d \
1196     mode=0444
1197 file path=opt/SUNWdtrt/tst/common/pragma/err.circlibdep.ksh mode=0444
1198 file path=opt/SUNWdtrt/tst/common/pragma/err.invalidlibdep.ksh mode=0444
1199 file path=opt/SUNWdtrt/tst/common/pragma/tst.libchain.ksh mode=0444
1200 file path=opt/SUNWdtrt/tst/common/pragma/tst.libdep.ksh mode=0444
1201 file path=opt/SUNWdtrt/tst/common/pragma/tst.libdepfullyconnected.ksh \
1202     mode=0444
1203 file path=opt/SUNWdtrt/tst/common/pragma/tst.libdepsepdir.ksh mode=0444
1204 file path=opt/SUNWdtrt/tst/common/pragma/tst.temporal.ksh mode=0444
1205 file path=opt/SUNWdtrt/tst/common/pragma/tst.temporal2.ksh mode=0444
1206 file path=opt/SUNWdtrt/tst/common/pragma/tst.temporal3.d mode=0444
1207 file path=opt/SUNWdtrt/tst/common/predicates/err.D_PRED_SCALAR.NonScalarPred.d \
1208     mode=0444
1209 file path=opt/SUNWdtrt/tst/common/predicates/err.D_SYNTAX.invalid.d mode=0444
1210 file path=opt/SUNWdtrt/tst/common/predicates/err.D_SYNTAX.operr.d mode=0444
1211 file path=opt/SUNWdtrt/tst/common/predicates/tst.argsnotcached.d mode=0444
1212 file path=opt/SUNWdtrt/tst/common/predicates/tst.basics.d mode=0444
1213 file path=opt/SUNWdtrt/tst/common/predicates/tst.basics.d.out mode=0444
1214 file path=opt/SUNWdtrt/tst/common/predicates/tst.complex.d mode=0444
1215 file path=opt/SUNWdtrt/tst/common/predicates/tst.complex.d.out mode=0444
1216 file path=opt/SUNWdtrt/tst/common/preprocessor/err.D_IDENT_UNDEF.afterprobe.d \
1217     mode=0444
1218 file path=opt/SUNWdtrt/tst/common/preprocessor/err.D_PRAGCTL_INVAL.tabdefine.d \
1219     mode=0444
1220 file path=opt/SUNWdtrt/tst/common/preprocessor/err.D_SYNTAX.withoutpound.d \
1221     mode=0444
1222 file path=opt/SUNWdtrt/tst/common/preprocessor/err.defincomp.d mode=0444
1223 file path=opt/SUNWdtrt/tst/common/preprocessor/err.ifdefelsenotendif.d \
1224     mode=0444
1225 file path=opt/SUNWdtrt/tst/common/preprocessor/err.ifdefincomp.d mode=0444
1226 file path=opt/SUNWdtrt/tst/common/preprocessor/err.ifdefnotendif.d mode=0444
1227 file path=opt/SUNWdtrt/tst/common/preprocessor/err.incompelse.d mode=0444
1228 file path=opt/SUNWdtrt/tst/common/preprocessor/err.mulelse.d mode=0444
1229 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.ifdef.d mode=0444
1230 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.ifdef.d.out mode=0444
1231 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.ifndef.d mode=0444
1232 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.ifndef.d.out mode=0444
1233 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.ifnotdef.d mode=0444
1234 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.ifnotdef.d.out mode=0444
1235 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.logicaland.d mode=0444
1236 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.logicaland.d.out mode=0444
1237 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.logicalandor.d mode=0444
1238 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.logicalandor.d.out \
1239     mode=0444
1240 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.logicalor.d mode=0444
1241 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.logicalor.d.out mode=0444
1242 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.muland.d mode=0444
1243 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.muland.d.out mode=0444
1244 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.mulor.d mode=0444
1245 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.mulor.d.out mode=0444
```



```
1246 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.precondi.d mode=0444
1247 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.precondi.d.out mode=0444
1248 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.predicatedeclare.d \
1249     mode=0444
1250 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.preexp.d mode=0444
1251 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.preexp.d.out mode=0444
1252 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.preexpelse.d mode=0444
1253 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.preexpelse.d.out mode=0444
1254 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.preexpif.d mode=0444
1255 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.preexpif.d.out mode=0444
1256 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.preexpifelse.d mode=0444
1257 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.preexpifelse.d.out \
1258     mode=0444
1259 file path=opt/SUNWdtrt/tst/common/preprocessor/tst.withinprobe.d mode=0444
1260 file path=opt/SUNWdtrt/tst/common/print/err.D_PRINT_AGG.bad.d mode=0444
1261 file path=opt/SUNWdtrt/tst/common/print/err.D_PRINT_VOID.bad.d mode=0444
1262 file path=opt/SUNWdtrt/tst/common/print/err.D_PROTO_LEN.bad.d mode=0444
1263 file path=opt/SUNWdtrt/tst/common/print/tst.array.d mode=0444
1264 file path=opt/SUNWdtrt/tst/common/print/tst.array.d.out mode=0444
1265 file path=opt/SUNWdtrt/tst/common/print/tst.bitfield.d mode=0444
1266 file path=opt/SUNWdtrt/tst/common/print/tst.bitfield.d.out mode=0444
1267 file path=opt/SUNWdtrt/tst/common/print/tst.dyn.d mode=0444
1268 file path=opt/SUNWdtrt/tst/common/print/tst.enum.d mode=0444
1269 file path=opt/SUNWdtrt/tst/common/print/tst.enum.d.out mode=0444
1270 file path=opt/SUNWdtrt/tst/common/print/tst.primitive.d mode=0444
1271 file path=opt/SUNWdtrt/tst/common/print/tst.primitive.d.out mode=0444
1272 file path=opt/SUNWdtrt/tst/common/print/tst.struct.d mode=0444
1273 file path=opt/SUNWdtrt/tst/common/print/tst.struct.d.out mode=0444
1274 file path=opt/SUNWdtrt/tst/common/print/tst.xlate.d mode=0444
1275 file path=opt/SUNWdtrt/tst/common/print/tst.xlate.d.out mode=0444
1276 file path=opt/SUNWdtrt/tst/common/printa/err.D_PRINTA_AGGARG.badagg.d \
1277     mode=0444
1278 file path=opt/SUNWdtrt/tst/common/printa/err.D_PRINTA_AGGARG.badfmt.d \
1279     mode=0444
1280 file path=opt/SUNWdtrt/tst/common/printa/err.D_PRINTA_AGGARG.badval.d \
1281     mode=0444
1282 file path=opt/SUNWdtrt/tst/common/printa/err.D_PRINTA_PROTO.bad.d mode=0444
1283 file path=opt/SUNWdtrt/tst/common/printa/err.D_PRINTF_ARG_TYPE.jstack.d \
1284     mode=0444
1285 file path=opt/SUNWdtrt/tst/common/printa/err.D_PRINTF_ARG_TYPE.stack.d \
1286     mode=0444
1287 file path=opt/SUNWdtrt/tst/common/printa/err.D_PRINTF_ARG_TYPE.ustack.d \
1288     mode=0444
1289 file path=opt/SUNWdtrt/tst/common/printa/tst.basics.d mode=0444
1290 file path=opt/SUNWdtrt/tst/common/printa/tst.basics.d.out mode=0444
1291 file path=opt/SUNWdtrt/tst/common/printa/tst.def.d mode=0444
1292 file path=opt/SUNWdtrt/tst/common/printa/tst.def.d.out mode=0444
1293 file path=opt/SUNWdtrt/tst/common/printa/tst.dynwidth.d mode=0444
1294 file path=opt/SUNWdtrt/tst/common/printa/tst.dynwidth.d.out mode=0444
1295 file path=opt/SUNWdtrt/tst/common/printa/tst.fmt.d mode=0444
1296 file path=opt/SUNWdtrt/tst/common/printa/tst.fmt.d.out mode=0444
1297 file path=opt/SUNWdtrt/tst/common/printa/tst.largeusersym.ksh mode=0444
1298 file path=opt/SUNWdtrt/tst/common/printa/tst.many.d mode=0444
1299 file path=opt/SUNWdtrt/tst/common/printa/tst.manyval.d mode=0444
1300 file path=opt/SUNWdtrt/tst/common/printa/tst.manyval.d.out mode=0444
1301 file path=opt/SUNWdtrt/tst/common/printa/tst.stack.d mode=0444
1302 file path=opt/SUNWdtrt/tst/common/printa/tst.tuple.d mode=0444
1303 file path=opt/SUNWdtrt/tst/common/printa/tst.tuple.d.out mode=0444
1304 file path=opt/SUNWdtrt/tst/common/printa/tst.walltimestamp.ksh mode=0444
1305 file path=opt/SUNWdtrt/tst/common/printa/tst.walltimestamp.ksh.out mode=0444
1306 file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_AGG_CONV.aggfmt.d \
1307     mode=0444
1308 file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_ARG_EXTRA.toomany.d \
1309     mode=0444
1310 file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_ARG_EXTRA.widths.d \
1311     mode=0444
```

```
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_ARG_FMT.badfmt.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_ARG_PROTO.novalue.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_ARG_TYPE.aggarg.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_ARG_TYPE.recursive.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_DYN_PROTO.noprec.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_DYN_PROTO.nowidth.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_DYN_TYPE.badprec.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PRINTF_DYN_TYPE.badwidth.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_PROTO_LEN.toofew.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_SYNTAX.badconv1.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_SYNTAX.badconv2.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/err.D_SYNTAX.badconv3.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.basics.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.basics.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.flags.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.flags.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.hello.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.hello.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.ints.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.ints.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.precs.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.precs.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.print-f.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.print-f.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printT.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printT.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printY.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printY.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printcont.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printcont.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printeE.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printeE.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printgG.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.printgG.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.rawfmt.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.rawfmt.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.signs.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.signs.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.str.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.str.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.sym.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.sym.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.uints.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.uints.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.widths.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.widths.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.widths1.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.wp.d mode=0444
file path=opt/SUNWdtrt/tst/common/printf/tst.wp.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/privs/tst.fds.ksh mode=0444
#endif /* ! codereview */
file path=opt/SUNWdtrt/tst/common/privs/tst.func_access.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/privs/tst.getf.ksh mode=0444
#endif /* ! codereview */
file path=opt/SUNWdtrt/tst/common/privs/tst.noprivdrop.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/privs/tst.noprivrestrict.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/privs/tst.op_access.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/privs/tst.procpriv.ksh mode=0444
```



```
file path=opt/SUNWdtrt/tst/common/privs/tst.providers.ksh mode=0444
#endif /* ! codereview */
file path=opt/SUNWdtrt/tst/common/privs/tst.tick.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/privs/tst.unpriv_funcs.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/probes/err.D_PDESC_ZERO.probeqtn.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/err.D_PDESC_ZERO.probestar.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/probes/err.D_PDESC_ZERO.tickstar.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/err.D_SYNTAX.assign.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/err.D_SYNTAX.declare.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/err.D_SYNTAX.declarein.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/err.D_SYNTAX.lbraces.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/err.D_SYNTAX.probespec.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/err.D_SYNTAX.rbraces.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/err.D_SYNTAX.recdec.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/tst.basic1.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/tst.check.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/tst.declare.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/tst.declareafter.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/tst.emptyprobe.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/tst.pragma.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/tst.pragmaaftertab.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/tst.pragmainside.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/tst.pragmaoutside.d mode=0444
file path=opt/SUNWdtrt/tst/common/probes/tst.probestar.d mode=0444
file path=opt/SUNWdtrt/tst/common/proc/tst.create.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/proc/tst.discard.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/proc/tst.exec.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/proc/tst.execfail.ENOENT.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/proc/tst.execfail.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/proc/tst.exitcore.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/proc/tst.exitexit.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/proc/tst.exitkilled.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/proc/tst.signal.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/proc/tst.sigwait.d mode=0444
file path=opt/SUNWdtrt/tst/common/proc/tst.sigwait.exe mode=0555
file path=opt/SUNWdtrt/tst/common/proc/tst.startexit.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/err.D_PDESC_ZERO.profile.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/err.D_PDESC_ZEROonens.d mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/err.D_PDESC_ZEROonensec.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/err.D_PDESC_ZEROoneus.d mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/err.D_PDESC_ZEROoneusec.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.argtest.d mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.argtest.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.basic.d mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.basic.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.func.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.mod.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profilehz.d mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profilehz.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profilems.d mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profilems.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profilemsec.d mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profilemsec.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profilenhz.d mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profilenhz.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profilens.d mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profilens.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profilensec.d mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profilensec.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profiles.d mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profiles.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profilesec.d mode=0444
```

```
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profilesec.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profileus.d mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profileus.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profileusec.d mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.profileusec.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.sym.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.ufunc.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.ufuncsort.exe mode=0555
file path=opt/SUNWdtrt/tst/common/profile-n/tst.ufuncsort.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.ufuncsort.ksh.out mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.umod.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/profile-n/tst.usym.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/providers/err.D_PDESC_INVAL.wrongdec4.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/providers/err.D_PDESC_ZERO.nonprofile.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/providers/err.D_PDESC_ZERO.wrongdec1.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/providers/err.D_PDESC_ZERO.wrongdec2.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/providers/err.D_PDESC_ZERO.wrongdec3.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.basics.d mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.basics.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.beginexit.d mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.beginprof.d mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.beginprof.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.probattrs.d mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.probattrs.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.probefunc.d mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.probefunc.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.probemod.d mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.probemod.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.probename.d mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.probename.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.probprov.d mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.probprov.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.profend.d mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.profend.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.profexit.d mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.profexit.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.trace.d mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.trace.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.twoprof.d mode=0444
file path=opt/SUNWdtrt/tst/common/providers/tst.twoprof.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/raise/tst.raise1.d mode=0444
file path=opt/SUNWdtrt/tst/common/raise/tst.raise1.exe mode=0555
file path=opt/SUNWdtrt/tst/common/raise/tst.raise2.d mode=0444
file path=opt/SUNWdtrt/tst/common/raise/tst.raise2.exe mode=0555
file path=opt/SUNWdtrt/tst/common/raise/tst.raise3.d mode=0444
file path=opt/SUNWdtrt/tst/common/raise/tst.raise3.exe mode=0555
file path=opt/SUNWdtrt/tst/common/rates/tst.aggrate.d mode=0444
file path=opt/SUNWdtrt/tst/common/rates/tst.aggrate.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/rates/tst.statusrate.d mode=0444
file path=opt/SUNWdtrt/tst/common/rates/tst.switchrate.d mode=0444
file path=opt/SUNWdtrt/tst/common/rates/tst.switchrate.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.basename.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.caller.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.cleanpath.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.copyin.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.copyin2.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.ddi_pathname.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.dirname.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.errno.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.execname.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.gid.d mode=0444
```



```
file path=opt/SUNWdtrt/tst/common/safety/tst.hton.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.index.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.msgdsize.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.msgsize.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.null.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.pid.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.ppid.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.progenyof.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.random.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.rw.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.shortstr.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.stack.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.stackdepth.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.stddev.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.strchr.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.strjoin.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.strstr.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.strtok.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.substr.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.ucaller.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.uid.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.unalign.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.uregs.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.ustack.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.ustackdepth.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.vahole.d mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.violentdeath.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/safety/tst.zonename.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_ARR_LOCAL.thisarray.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_DECL_CLASS.selfthis.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_DECL_CLASS.thisself.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_DECL_IDRED.errval.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_OP_INCOMPAT.dec.err.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_OP_INCOMPAT.dupgtype.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_OP_INCOMPAT.dupltype.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_OP_INCOMPAT.dupttype.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/err.D_SYNTAX.declare.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.basicvar.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.basicvar.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.localvar.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.misc.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.self.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.selfarray.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.selfarray2.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.selfthis.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.this.d mode=0444
file path=opt/SUNWdtrt/tst/common/scalars/tst.thisself.d mode=0444
file path=opt/SUNWdtrt/tst/common/sched/tst.enqueue.d mode=0444
file path=opt/SUNWdtrt/tst/common/sched/tst.oncpu.d mode=0444
file path=opt/SUNWdtrt/tst/common/sched/tst.stackdepth.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/err.D_MACRO_UNDEF.invalidargs.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/err.D_OP_LVAL.rdonly.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/err.D_OP_WRITE.usepidmacro.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/err.D_SYNTAX.concat.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/err.D_SYNTAX.desc.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/err.D_SYNTAX.inval.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/err.D_SYNTAX.pid.d mode=0444
```

```
file path=opt/SUNWdtrt/tst/common/scripting/tst.D_MACRO_UNUSED.overflow.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.arg0.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.arguments.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.assign.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.basic.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.egid.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.egid.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.euid.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.euid.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.gid.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.gid.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.pgid.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.pid.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.ppid.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.ppid.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.projid.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.projid.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.quite.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.sid.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.sid.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.stringmacro.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.taskid.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.taskid.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.trace.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.uid.d mode=0444
file path=opt/SUNWdtrt/tst/common/scripting/tst.uid.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/sdt/tst.sdtargs.d mode=0444
file path=opt/SUNWdtrt/tst/common/sdt/tst.sdtargs.exe mode=0555
file path=opt/SUNWdtrt/tst/common/sizeof/err.D_IDENT_BADREF.SizeofAssoc.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/sizeof/err.D_IDENT_UNDEF.UnknownSymbol.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/sizeof/err.D_SIZEOF_TYPE.badstruct.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/sizeof/err.D_SIZEOF_TYPE.d mode=0444
file path=opt/SUNWdtrt/tst/common/sizeof/err.D_SYNTAX.SizeofBadType.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofArray.d mode=0444
file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofDataTypes.d mode=0444
file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofExpression.d mode=0444
file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofNULL.d mode=0444
file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofStrConst.d mode=0444
file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofStrConst.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofString1.d mode=0444
file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofString1.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofString2.d mode=0444
file path=opt/SUNWdtrt/tst/common/sizeof/tst.SizeofString2.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.BufSizeVariations1.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.BufSizeVariations2.d \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_ACT_SPEC.SpeculateWithBreakPo
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_ACT_SPEC.SpeculateWithChill.d
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_ACT_SPEC.SpeculateWithCopyOut
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_ACT_SPEC.SpeculateWithCopyOut
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_ACT_SPEC.SpeculateWithPanic.d
```



```
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_ACT_SPEC.SpeculateWithRaise.d
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_ACT_SPEC.SpeculateWithStop.d
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_COMM.AggAftCommit.d \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_SPEC.SpeculateWithAvg.d \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_SPEC.SpeculateWithCount.d
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_SPEC.SpeculateWithLquant.
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_SPEC.SpeculateWithMax.d \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_SPEC.SpeculateWithMin.d \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_SPEC.SpeculateWithQuant.d
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_SPEC.SpeculateWithStddev.
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_AGG_SPEC.SpeculateWithSum.d \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_COMM_COMM.CommitAftCommit.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.D_COMM_COMM.DisjointCommit.d \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_COMM_DREC.CommitAftDataRec.d
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_DREC_COMM.DataRecAftCommit.d
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_DREC_COMM.ExitAfterCommit.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.D_EXIT_SPEC.ExitAftSpec.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.D_PRAGMA_MALFORM.NspecExpr.d \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_PRAGMA_OPTSET.HugeNspecValue.
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_PRAGMA_OPTSET.InvalidSpecSize
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/speculation/err.D_PRAGMA_OPTSET.NegSpecSize.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.D_PROTO_LEN.SpecNoId.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.D_SPEC_COMM.SpecAftCommit.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.D_SPEC_DREC.SpecAftDataRec.d \
    mode=0444
```

```
file path=opt/SUNWdtrt/tst/common/speculation/err.D_SPEC_SPEC.SpecAftSpec.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.NegativeBufSize.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.NegativeNspec.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.NegativeSpecSize.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.SpecSizeVariations1.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/err.SpecSizeVariations2.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.CommitAfterDiscard.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.CommitWithZero.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.DataRecAftDiscard.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.DiscardAftCommit.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.DiscardAftDataRec.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.DiscardAftDiscard.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.DiscardWithZero.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.ExitAftDiscard.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.NoSpecBuffer.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.SpecSizeVariations1.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.SpecSizeVariations2.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.SpecSizeVariations3.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.SpeculateWithRandom.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.SpeculationCommit.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.SpeculationDiscard.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.SpeculationID.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.SpeculationWithZero.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.TwoSpecBuffers.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.negcommit.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.negspec.d mode=0444
file path=opt/SUNWdtrt/tst/common/speculation/tst.zerosize.d mode=0444
file path=opt/SUNWdtrt/tst/common/stability/err.D_ATTR_MIN.MinAttributes.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/stack/err.D_STACK_PROTO.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/stack/err.D_STACK_SIZE.d mode=0444
file path=opt/SUNWdtrt/tst/common/stack/err.D_USTACK_FRAMES.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/stack/err.D_USTACK_PROTO.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/stack/err.D_USTACK_STRSIZE.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/stack/tst.default.d mode=0444
file path=opt/SUNWdtrt/tst/common/stackdepth/tst.default.d mode=0444
file path=opt/SUNWdtrt/tst/common/stop/tst.stop1.d mode=0444
file path=opt/SUNWdtrt/tst/common/stop/tst.stop1.exe mode=0555
file path=opt/SUNWdtrt/tst/common/stop/tst.stop2.d mode=0444
file path=opt/SUNWdtrt/tst/common/stop/tst.stop2.exe mode=0555
file path=opt/SUNWdtrt/tst/common/strlen/tst.strlen1.d mode=0444
file path=opt/SUNWdtrt/tst/common/struct/err.D_ADDROF_VAR.StructPointer.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/struct/err.D_DECL_COMBO.StructWithoutColon.d \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/struct/err.D_DECL_COMBO.StructWithoutColon1.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/struct/err.D_DECL_INCOMPLETE.circular.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/struct/err.D_DECL_INCOMPLETE.order.d \
    mode=0444
```



```
file path=opt/SUNWdtrt/tst/common/struct/err.D_DECL_INCOMPLETE.order2.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/struct/err.D_DECL_INCOMPLETE.recursive.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/struct/err.D_DECL_INCOMPLETE.simple.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/struct/err.D_DECL_VOIDOBJ.baddec.d mode=0444
file path=opt/SUNWdtrt/tst/common/struct/err.D_PROTO_ARG.DupStructAssoc.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/struct/tst.StructAssoc.d mode=0444
file path=opt/SUNWdtrt/tst/common/struct/tst.StructDataTypes.d mode=0444
file path=opt/SUNWdtrt/tst/common/struct/tst.StructInside.d mode=0444
file path=opt/SUNWdtrt/tst/common/struct/tst.clauselocal.d mode=0444
file path=opt/SUNWdtrt/tst/common/struct/tst.clauselocal.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/syscall/tst.args.d mode=0444
file path=opt/SUNWdtrt/tst/common/syscall/tst.args.exe mode=0555
file path=opt/SUNWdtrt/tst/common/syscall/tst.openret.ksh mode=0444
file path=opt/SUNWdtrt/tst/common/sysevent/tst.post.d mode=0444
file path=opt/SUNWdtrt/tst/common/sysevent/tst.post.exe mode=0555
file path=opt/SUNWdtrt/tst/common/sysevent/tst.post_chan.d mode=0444
file path=opt/SUNWdtrt/tst/common/sysevent/tst.post_chan.exe mode=0555
file path=opt/SUNWdtrt/tst/common/tick-n/err.D_PDESC_ZERO.tick.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/err.D_PDESC_ZEROonens.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/err.D_PDESC_ZEROonensec.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/err.D_PDESC_ZEROoneus.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/err.D_PDESC_ZEROoneusec.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickarg0.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickms.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickms.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickmsec.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickmsec.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickns.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickns.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.ticknsec.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.ticknsec.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.ticks.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.ticks.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.ticksec.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.ticksec.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickus.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickus.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickusec.d mode=0444
file path=opt/SUNWdtrt/tst/common/tick-n/tst.tickusec.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/trace/err.D_PROTO_LEN.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/trace/err.D_TRACE_AGG.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/trace/err.D_TRACE_VOID.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/trace/tst.dyn.d mode=0444
file path=opt/SUNWdtrt/tst/common/trace/tst.misc.d mode=0444
file path=opt/SUNWdtrt/tst/common/trace/tst.qstring.d mode=0444
file path=opt/SUNWdtrt/tst/common/trace/tst.qstring.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/trace/tst.string.d mode=0444
file path=opt/SUNWdtrt/tst/common/tracemem/err.D_PROTO_ARG.badsize.d mode=0444
file path=opt/SUNWdtrt/tst/common/tracemem/err.D_PROTO_LEN.toofew.d mode=0444
file path=opt/SUNWdtrt/tst/common/tracemem/err.D_TRACEMEM_ADDR.badaddr.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/tracemem/err.D_TRACEMEM_ARGS.d mode=0444
file path=opt/SUNWdtrt/tst/common/tracemem/err.D_TRACEMEM_DYNSIZE.d mode=0444
file path=opt/SUNWdtrt/tst/common/tracemem/err.D_TRACEMEM_SIZE.negsize.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/tracemem/err.D_TRACEMEM_SIZE.zerosize.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/tracemem/tst.dynsize.d mode=0444
file path=opt/SUNWdtrt/tst/common/tracemem/tst.dynsize.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/tracemem/tst.rootvp.d mode=0444
file path=opt/SUNWdtrt/tst/common/tracemem/tst.smallsize.d mode=0444
file path=opt/SUNWdtrt/tst/common/tracemem/tst.smallsize.d.out mode=0444
```

```
file \
    path=opt/SUNWdtrt/tst/common/translators/err.D_DECL_TYPERED.BadTransDecl.d \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/translators/err.D_OP_INCOMPLETE.NonExistentInpu
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/err.D_SYNTAX.BadTransDecl1.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/err.D_SYNTAX.BadTransDecl3.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/err.D_SYNTAX.BadTransDecl4.d \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/translators/err.D_TYPE_MEMBER.NonExistentInput2
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/translators/err.D_XLATE_INCOMPAT.BadInputType1.
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/translators/err.D_XLATE_MEMB.NonExistentOutput2
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/err.D_XLATE_NONE.BadTransDecl6.d \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/translators/err.D_XLATE_REDECL.RepeatTransDecl.
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/err.D_XLATE_SOU.BadTransDecl8.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/err.D_XLATE_SOU.BadTransInt.d \
    mode=0444
file \
    path=opt/SUNWdtrt/tst/common/translators/err.D_XLATE_SOU.NonExistentOutput1.
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.CircularTransDecl.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.EmptyTransDecl.d mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.ForwardTag.d mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.InputAliasTrans.d mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.InputIntTrans.d mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.OutputAliasTrans.d mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.PartialDereferencing.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.PartialOutputTransDefn.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.ProcModelTrans.d mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.RepeatDeclaration.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.SimultaneousTranslators.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.StructureAssignment.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.TestTransStability1.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.TestTransStability1.ksh.out \
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.TestTransStability2.ksh \
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.TestTransStability2.ksh.out \
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.TransNonPointer.d mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.TransOutputPointer.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.TransPointer.d mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.TranslateSelf.d mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.UnionInputTrans.d mode=0444
file path=opt/SUNWdtrt/tst/common/translators/tst.UnionOutputTrans.d mode=0444
```



```
file path=opt/SUNWdtrt/tst/common/typedef/err.D_DECL_IDRED.DupTypeDef.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/typedef/err.D_SYNTAX.BadExistingTypedef.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/typedef/err.D_SYNTAX.TypedefInClause.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/typedef/tst.ChainTypedef.d mode=0444
file path=opt/SUNWdtrt/tst/common/typedef/tst.TypedefDataAssign.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_CAST_INVAL.badcast.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_CG_DYN.ResultDynType.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_CHR_OFLOW.charconst.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_BADCLASS.bad.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_CHARATTR.badtype3.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_COMBO.badtype4.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_COMBO.badtype5.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_ENCONST.badeval.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_ENOFLOW.enoflow.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_ENOFLOW.enuflow.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_SCOPE.scopeop.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_DECL_USELESS.baddec.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_OP_ACT.badcond.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_OP_ARITH.badoperand.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_OP_INCOMPAT.badassign.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_OP_INT.badbitop.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_OP_INT.badshift.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_OP_SCALAR.badcond.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_OP_SCALAR.badincop.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_OP_SCALAR.badlogop.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_PROTO_LEN.badcond1.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_SYNTAX.badenum.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_SYNTAX.badid.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_SYNTAX.badstruct.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_UNKNOWN.badtype1.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_UNKNOWN.badtype2.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_UNKNOWN.dupenum.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_UNKNOWN.dupstruct.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/err.D_XLATE_REDECL.ResultDynType.d \
    mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.assignops.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.badshiftops.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.basics.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.basics.d.out mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.bitops.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.charconstants.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.complex.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.condexpr.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.const.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.constants.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.conv.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.enum.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.intincop.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.intops.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.inttypes.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.ptrincop.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.ptrops.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.relenum.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.relstring.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.shiftops.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.stringconstants.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.struct.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.typedef.d mode=0444
file path=opt/SUNWdtrt/tst/common/types/tst.unaryop.d mode=0444
file path=opt/SUNWdtrt/tst/common/union/err.D_ADDROF_VAR.UnionPointer.d \
    mode=0444
```

```
1972 file path=opt/SUNWdtrt/tst/common/union/err.D_DECL_COMBO.UnionWithoutColon.d \
1973     mode=0444
1974 file path=opt/SUNWdtrt/tst/common/union/err.D_DECL_COMBO.UnionWithoutColon1.d \
1975     mode=0444
1976 file path=opt/SUNWdtrt/tst/common/union/err.D_DECL_INCOMPLETE.circular.d \
1977     mode=0444
1978 file path=opt/SUNWdtrt/tst/common/union/err.D_DECL_INCOMPLETE.order.d \
1979     mode=0444
1980 file path=opt/SUNWdtrt/tst/common/union/err.D_DECL_INCOMPLETE.recursive.d \
1981     mode=0444
1982 file path=opt/SUNWdtrt/tst/common/union/err.D_DECL_INCOMPLETE.simple.d \
1983     mode=0444
1984 file path=opt/SUNWdtrt/tst/common/union/err.D_PROTO_ARG.DupUnionAssoc.d \
1985     mode=0444
1986 file path=opt/SUNWdtrt/tst/common/union/tst.UnionAssoc.d mode=0444
1987 file path=opt/SUNWdtrt/tst/common/union/tst.UnionDataTypes.d mode=0444
1988 file path=opt/SUNWdtrt/tst/common/union/tst.UnionInside.d mode=0444
1989 file path=opt/SUNWdtrt/tst/common/usdt/tst.andpid.ksh mode=0444
1990 file path=opt/SUNWdtrt/tst/common/usdt/tst.argmap.d mode=0444
1991 file path=opt/SUNWdtrt/tst/common/usdt/tst.argmap.exe mode=0555
1992 file path=opt/SUNWdtrt/tst/common/usdt/tst.args.d mode=0444
1993 file path=opt/SUNWdtrt/tst/common/usdt/tst.args.exe mode=0555
1994 file path=opt/SUNWdtrt/tst/common/usdt/tst.badguess.ksh mode=0444
1995 file path=opt/SUNWdtrt/tst/common/usdt/tst.corruptenv.ksh mode=0444
1996 file path=opt/SUNWdtrt/tst/common/usdt/tst.dlclose1.ksh mode=0444
1997 file path=opt/SUNWdtrt/tst/common/usdt/tst.dlclose1.ksh.out mode=0444
1998 file path=opt/SUNWdtrt/tst/common/usdt/tst.dlclose2.ksh mode=0444
1999 file path=opt/SUNWdtrt/tst/common/usdt/tst.dlclose2.ksh.out mode=0444
2000 file path=opt/SUNWdtrt/tst/common/usdt/tst.dlclose3.ksh mode=0444
2001 file path=opt/SUNWdtrt/tst/common/usdt/tst.eliminate.ksh mode=0444
2002 file path=opt/SUNWdtrt/tst/common/usdt/tst.enabled.ksh mode=0444
2003 file path=opt/SUNWdtrt/tst/common/usdt/tst.enabled.ksh.out mode=0444
2004 file path=opt/SUNWdtrt/tst/common/usdt/tst.enabled2.ksh mode=0444
2005 file path=opt/SUNWdtrt/tst/common/usdt/tst.enabled2.ksh.out mode=0444
2006 file path=opt/SUNWdtrt/tst/common/usdt/tst.entryreturn.ksh mode=0444
2007 file path=opt/SUNWdtrt/tst/common/usdt/tst.entryreturn.ksh.out mode=0444
2008 file path=opt/SUNWdtrt/tst/common/usdt/tst.fork.ksh mode=0444
2009 file path=opt/SUNWdtrt/tst/common/usdt/tst.fork.ksh.out mode=0444
2010 file path=opt/SUNWdtrt/tst/common/usdt/tst.forker.exe mode=0555
2011 file path=opt/SUNWdtrt/tst/common/usdt/tst.forker.ksh mode=0444
2012 file path=opt/SUNWdtrt/tst/common/usdt/tst.guess32.ksh mode=0444
2013 file path=opt/SUNWdtrt/tst/common/usdt/tst.guess64.ksh mode=0444
2014 file path=opt/SUNWdtrt/tst/common/usdt/tst.header.ksh mode=0444
2015 file path=opt/SUNWdtrt/tst/common/usdt/tst.include.ksh mode=0444
2016 file path=opt/SUNWdtrt/tst/common/usdt/tst.lazyprobe.exe mode=0555
2017 file path=opt/SUNWdtrt/tst/common/usdt/tst.lazyprobe1.ksh mode=0444
2018 file path=opt/SUNWdtrt/tst/common/usdt/tst.lazyprobe2.ksh mode=0444
2019 file path=opt/SUNWdtrt/tst/common/usdt/tst.linkpriv.ksh mode=0444
2020 file path=opt/SUNWdtrt/tst/common/usdt/tst.linkunpriv.ksh mode=0444
2021 file path=opt/SUNWdtrt/tst/common/usdt/tst.multiple.ksh mode=0444
2022 file path=opt/SUNWdtrt/tst/common/usdt/tst.multiple.ksh.out mode=0444
2023 file path=opt/SUNWdtrt/tst/common/usdt/tst.multiprov.ksh mode=0444
2024 file path=opt/SUNWdtrt/tst/common/usdt/tst.multiprov.ksh.out mode=0444
2025 file path=opt/SUNWdtrt/tst/common/usdt/tst.nodtrace.ksh mode=0444
2026 file path=opt/SUNWdtrt/tst/common/usdt/tst.noprobes.ksh mode=0444
2027 file path=opt/SUNWdtrt/tst/common/usdt/tst.noreap.ksh mode=0444
2028 file path=opt/SUNWdtrt/tst/common/usdt/tst.noreapring.ksh mode=0444
2029 file path=opt/SUNWdtrt/tst/common/usdt/tst.onlyenabled.ksh mode=0444
2030 file path=opt/SUNWdtrt/tst/common/usdt/tst.reap.ksh mode=0444
2031 file path=opt/SUNWdtrt/tst/common/usdt/tst.reeval.ksh mode=0444
2032 file path=opt/SUNWdtrt/tst/common/usdt/tst.static.ksh mode=0444
2033 file path=opt/SUNWdtrt/tst/common/usdt/tst.static.ksh.out mode=0444
2034 file path=opt/SUNWdtrt/tst/common/usdt/tst.static2.ksh mode=0444
2035 file path=opt/SUNWdtrt/tst/common/usdt/tst.static2.ksh.out mode=0444
2036 file path=opt/SUNWdtrt/tst/common/usdt/tst.user.ksh mode=0444
2037 file path=opt/SUNWdtrt/tst/common/usdt/tst.user.ksh.out mode=0444
```



```
2038 file path=opt/SUNWdtrt/tst/common/ustack/tst.bigstack.d mode=0444
2039 file path=opt/SUNWdtrt/tst/common/ustack/tst.bigstack.exe mode=0555
2040 file path=opt/SUNWdtrt/tst/common/ustack/tst.depth.ksh mode=0444
2041 file path=opt/SUNWdtrt/tst/common/ustack/tst.spin.exe mode=0555
2042 file path=opt/SUNWdtrt/tst/common/ustack/tst.spin.ksh mode=0444
2043 file path=opt/SUNWdtrt/tst/common/vars/tst.gid.d mode=0444
2044 file path=opt/SUNWdtrt/tst/common/vars/tst.nullassign.d mode=0444
2045 file path=opt/SUNWdtrt/tst/common/vars/tst.ppid.d mode=0444
2046 file path=opt/SUNWdtrt/tst/common/vars/tst.ucaller.ksh mode=0444
2047 file path=opt/SUNWdtrt/tst/common/vars/tst.ucaller.ksh.out mode=0444
2048 file path=opt/SUNWdtrt/tst/common/vars/tst.uid.d mode=0444
2049 file path=opt/SUNWdtrt/tst/common/vars/tst.walltimestamp.d mode=0444
2050 file path=opt/SUNWdtrt/tst/common/version/tst.1.0.d mode=0444
2051 $(i386_ONLY)file path=opt/SUNWdtrt/tst/i86xpv/xdt/tst.basic.ksh mode=0444
2052 $(i386_ONLY)file path=opt/SUNWdtrt/tst/i86xpv/xdt/tst.hvmenable.ksh mode=0444
2053 $(i386_ONLY)file path=opt/SUNWdtrt/tst/i86xpv/xdt/tst.memenable.ksh mode=0444
2054 $(i386_ONLY)file path=opt/SUNWdtrt/tst/i86xpv/xdt/tst.schedargs.ksh mode=0444
2055 $(i386_ONLY)file path=opt/SUNWdtrt/tst/i86xpv/xdt/tst.schedenable.ksh \
2056     mode=0444
2057 legacy pkg=SUNWdtrt category=internal \
2058     desc="DTrace Test Suite Internal Distribution" \
2059     hotline="Contact the DTrace discussion forum" name="DTrace Test Suite"
2060 license cr_Sun license=cr_Sun
2061 license lic_CDDL license=lic_CDDL
2062 depend fmri=runtime/java type=require
2063 depend fmri=runtime/java/runtime64 type=require
```

```
**********************************************************
  420740 Tue Jan 14 16:50:01 2014
new/usr/src/uts/common/dtrace/dtrace.c
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

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

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

/*
 * DTrace - Dynamic Tracing for Solaris
 *
 * This is the implementation of the Solaris Dynamic Tracing framework
 * (DTrace).  The user-visible interface to DTrace is described at length in
 * the "Solaris Dynamic Tracing Guide".  The interfaces between the libdtrace
 * library, the in-kernel DTrace framework, and the DTrace providers are
 * described in the block comments in the <sys/dtrace.h> header file.  The
 * internal architecture of DTrace is described in the block comments in the
 * <sys/dtrace_impl.h> header file.  The comments contained within the DTrace
 * implementation very much assume mastery of all of these sources; if one has
 * an unanswered question about the implementation, one should consult them
 * first.
 *
 * The functions here are ordered roughly as follows:
 *
 *   - Probe context functions
 *   - Probe hashing functions
 *   - Non-probe context utility functions
 *   - Matching functions
 *   - Provider-to-Framework API functions
 *   - Probe management functions
 *   - DIF object functions
 *   - Format functions
 *   - Predicate functions
 *   - ECB functions
 *   - Buffer functions
 *   - Enabling functions
 *   - DOF functions
 *   - Anonymous enabling functions
```



```
 *   - Consumer state functions
 *   - Helper functions
 *   - Hook functions
 *   - Driver cookbook functions
 *
 * Each group of functions begins with a block comment labelled the "DTrace
 * [Group] Functions", allowing one to find each block by searching forward
 * on capital-f functions.
 */
#include <sys/errno.h>
#include <sys/stat.h>
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/systm.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/cpuvar.h>
#include <sys/kmem.h>
#include <sys/strsubr.h>
#include <sys/sysmacros.h>
#include <sys/dtrace_impl.h>
#include <sys/atomic.h>
#include <sys/cmn_err.h>
#include <sys/mutex_impl.h>
#include <sys/rwlock_impl.h>
#include <sys/ctf_api.h>
#include <sys/panic.h>
#include <sys/priv_impl.h>
#include <sys/policy.h>
#include <sys/cred_impl.h>
#include <sys/procfs_isa.h>
#include <sys/taskq.h>
#include <sys/mkdev.h>
#include <sys/kdi.h>
#include <sys/zone.h>
#include <sys/socket.h>
#include <netinet/in.h>

/*
 * DTrace Tunable Variables
 *
 * The following variables may be tuned by adding a line to /etc/system that
 * includes both the name of the DTrace module ("dtrace") and the name of the
 * variable.  For example:
 *
 *   set dtrace:dtrace_destructive_disallow = 1
 *
 * In general, the only variables that one should be tuning this way are those
 * that affect system-wide DTrace behavior, and for which the default behavior
 * is undesirable.  Most of these variables are tunable on a per-consumer
 * basis using DTrace options, and need not be tuned on a system-wide basis.
 * When tuning these variables, avoid pathological values; while some attempt
 * is made to verify the integrity of these variables, they are not considered
 * part of the supported interface to DTrace, and they are therefore not
 * checked comprehensively.  Further, these variables should not be tuned
 * dynamically via "mdb -kw" or other means; they should only be tuned via
 * /etc/system.
 */
int             dtrace_destructive_disallow = 0;
dtrace_optval_t dtrace_nonroot_maxsize = (16 * 1024 * 1024);
size_t          dtrace_difo_maxsize = (256 * 1024);
dtrace_optval_t dtrace_dof_maxsize = (256 * 1024);
size_t          dtrace_global_maxsize = (16 * 1024);
size_t          dtrace_actions_max = (16 * 1024);
size_t          dtrace_retain_max = 1024;
dtrace_optval_t dtrace_helper_actions_max = 1024;
```

```
dtrace_optval_t dtrace_helper_providers_max = 32;
dtrace_optval_t dtrace_dstate_defsize = (1 * 1024 * 1024);
size_t          dtrace_strsize_default = 256;
dtrace_optval_t dtrace_cleanrate_default = 9900990;             /* 101 hz */
dtrace_optval_t dtrace_cleanrate_min = 200000;                  /* 5000 hz */
dtrace_optval_t dtrace_cleanrate_max = (uint64_t)60 * NANOSEC;  /* 1/minute */
dtrace_optval_t dtrace_aggrate_default = NANOSEC;               /* 1 hz */
dtrace_optval_t dtrace_statusrate_default = NANOSEC;            /* 1 hz */
dtrace_optval_t dtrace_statusrate_max = (hrtime_t)10 * NANOSEC;  /* 6/minute */
dtrace_optval_t dtrace_switchrate_default = NANOSEC;            /* 1 hz */
dtrace_optval_t dtrace_nspec_default = 1;
dtrace_optval_t dtrace_specsize_default = 32 * 1024;
dtrace_optval_t dtrace_stackframes_default = 20;
dtrace_optval_t dtrace_ustackframes_default = 20;
dtrace_optval_t dtrace_jstackframes_default = 50;
dtrace_optval_t dtrace_jstackstrsize_default = 512;
int             dtrace_msgdsize_max = 128;
hrtime_t        dtrace_chill_max = 500 * (NANOSEC / MILLISEC);  /* 500 ms */
hrtime_t        dtrace_chill_interval = NANOSEC;                /* 1000 ms */
int             dtrace_devdepth_max = 32;
int             dtrace_err_verbose;
hrtime_t        dtrace_deadman_interval = NANOSEC;
hrtime_t        dtrace_deadman_timeout = (hrtime_t)10 * NANOSEC;
hrtime_t        dtrace_deadman_user = (hrtime_t)30 * NANOSEC;
hrtime_t        dtrace_unregister_defunct_reap = (hrtime_t)60 * NANOSEC;

/*
 * DTrace External Variables
 *
 * As dtrace(7D) is a kernel module, any DTrace variables are obviously
 * available to DTrace consumers via the backtick (`) syntax.  One of these,
 * dtrace_zero, is made deliberately so:  it is provided as a source of
 * well-known, zero-filled memory.  While this variable is not documented,
 * it is used by some translators as an implementation detail.
 */
const char      dtrace_zero[256] = { 0 };       /* zero-filled memory */

/*
 * DTrace Internal Variables
 */
static dev_info_t       *dtrace_devi;           /* device info */
static vmem_t           *dtrace_arena;          /* probe ID arena */
static vmem_t           *dtrace_minor;          /* minor number arena */
static taskq_t          *dtrace_taskq;          /* task queue */
static dtrace_probe_t   **dtrace_probes;        /* array of all probes */
static int              dtrace_nprobes;         /* number of probes */
static dtrace_provider_t *dtrace_provider;      /* provider list */
static dtrace_meta_t    *dtrace_meta_pid;       /* user-land meta provider */
static int              dtrace_opens;           /* number of opens */
static int              dtrace_helpers;         /* number of helpers */
static int              dtrace_getf;            /* number of unpriv getf()s */
#endif /* ! codereview */
static void             *dtrace_softstate;      /* softstate pointer */
static dtrace_hash_t    *dtrace_bymod;          /* probes hashed by module */
static dtrace_hash_t    *dtrace_byfunc;         /* probes hashed by function */
static dtrace_hash_t    *dtrace_byname;         /* probes hashed by name */
static dtrace_toxrange_t *dtrace_toxrange;      /* toxic range array */
static int              dtrace_toxranges;       /* number of toxic ranges */
static int              dtrace_toxranges_max;   /* size of toxic range array */
static dtrace_anon_t    dtrace_anon;            /* anonymous enabling */
static kmem_cache_t     *dtrace_state_cache;    /* cache for dynamic state */
static uint64_t         dtrace_vtime_references; /* number of vtimestamp refs */
static kthread_t        *dtrace_panicked;       /* panicking thread */
static dtrace_ecb_t     *dtrace_ecb_create_cache; /* cached created ECB */
static dtrace_genid_t   dtrace_probegen;        /* current probe generation */
static dtrace_helpers_t *dtrace_deferred_pid;   /* deferred helper list */
```



```
static dtrace_enabling_t *dtrace_retained;      /* list of retained enablings */
static dtrace_genid_t   dtrace_retained_gen;    /* current retained enab gen */
static dtrace_dynvar_t  dtrace_dynhash_sink;    /* end of dynamic hash chains */
static int              dtrace_dynvar_failclean; /* dynvars failed to clean */

/*
 * DTrace Locking
 * DTrace is protected by three (relatively coarse-grained) locks:
 *
 * (1) dtrace_lock is required to manipulate essentially any DTrace state,
 *     including enabling state, probes, ECBs, consumer state, helper state,
 *     etc.  Importantly, dtrace_lock is _not_ required when in probe context;
 *     probe context is lock-free -- synchronization is handled via the
 *     dtrace_sync() cross call mechanism.
 *
 * (2) dtrace_provider_lock is required when manipulating provider state, or
 *     when provider state must be held constant.
 *
 * (3) dtrace_meta_lock is required when manipulating meta provider state, or
 *     when meta provider state must be held constant.
 *
 * The lock ordering between these three locks is dtrace_meta_lock before
 * dtrace_provider_lock before dtrace_lock.  (In particular, there are
 * several places where dtrace_provider_lock is held by the framework as it
 * calls into the providers -- which then call back into the framework,
 * grabbing dtrace_lock.)
 *
 * There are two other locks in the mix:  mod_lock and cpu_lock.  With respect
 * to dtrace_provider_lock and dtrace_lock, cpu_lock continues its historical
 * role as a coarse-grained lock; it is acquired before both of these locks.
 * With respect to dtrace_meta_lock, its behavior is stranger:  cpu_lock must
 * be acquired _between_ dtrace_meta_lock and any other DTrace locks.
 * mod_lock is similar with respect to dtrace_provider_lock in that it must be
 * acquired _between_ dtrace_provider_lock and dtrace_lock.
 */
static kmutex_t         dtrace_lock;            /* probe state lock */
static kmutex_t         dtrace_provider_lock;   /* provider state lock */
static kmutex_t         dtrace_meta_lock;       /* meta-provider state lock */

/*
 * DTrace Provider Variables
 *
 * These are the variables relating to DTrace as a provider (that is, the
 * provider of the BEGIN, END, and ERROR probes).
 */
static dtrace_pattr_t   dtrace_provider_attr = {
{ DTRACE_STABILITY_STABLE, DTRACE_STABILITY_STABLE, DTRACE_CLASS_COMMON },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_STABLE, DTRACE_STABILITY_STABLE, DTRACE_CLASS_COMMON },
{ DTRACE_STABILITY_STABLE, DTRACE_STABILITY_STABLE, DTRACE_CLASS_COMMON },
};

static void
dtrace_nullop(void)
{}

static int
dtrace_enable_nullop(void)
{
        return (0);
}

static dtrace_pops_t    dtrace_provider_ops = {
        (void (*)(void *, const dtrace_probedesc_t *))dtrace_nullop,
        (void (*)(void *, struct modctl *))dtrace_nullop,
```

```
256         (int (*)(void *, dtrace_id_t, void *))dtrace_enable_nullop,
257         (void (*)(void *, dtrace_id_t, void *))dtrace_nullop,
258         (void (*)(void *, dtrace_id_t, void *))dtrace_nullop,
259         (void (*)(void *, dtrace_id_t, void *))dtrace_nullop,
260         NULL,
261         NULL,
262         NULL,
263         (void (*)(void *, dtrace_id_t, void *))dtrace_nullop
264 };

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 uint32_t dtrace_helptrace_next = 0;
274 uint32_t dtrace_helptrace_nlocals;
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

284 /*
285  * DTrace Error Hashing
286  *
287  * On DEBUG kernels, DTrace will track the errors that has seen in a hash
288  * table.  This is very useful for checking coverage of tests that are
289  * expected to induce DIF or DOF processing errors, and may be useful for
290  * debugging problems in the DIF code generator or in DOF generation .  The
291  * error hash may be examined with the ::dtrace_errhash MDB dcmd.
292  */
293 #ifdef DEBUG
294 static dtrace_errhash_t dtrace_errhash[DTRACE_ERRHASHSZ];
295 static const char *dtrace_errlast;
296 static kthread_t *dtrace_errthread;
297 static kmutex_t dtrace_errlock;
298 #endif

300 /*
301  * DTrace Macros and Constants
302  *
303  * These are various macros that are useful in various spots in the
304  * implementation, along with a few random constants that have no meaning
305  * outside of the implementation.  There is no real structure to this cpp
306  * mishmash -- but is there ever?
307  */
308 #define DTRACE_HASHSTR(hash, probe)     \
309         dtrace_hash_str(*((char **)((uintptr_t)(probe) + (hash)->dth_stroffs)))

311 #define DTRACE_HASHNEXT(hash, probe)    \
312         (dtrace_probe_t **)((uintptr_t)(probe) + (hash)->dth_nextoffs)

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

317 #define DTRACE_HASHEQ(hash, lhs, rhs)   \
318         (strcmp(*((char **)((uintptr_t)(lhs) + (hash)->dth_stroffs)), \
319             *((char **)((uintptr_t)(rhs) + (hash)->dth_stroffs))) == 0)

321 #define DTRACE_AGGHASHSIZE_SLEW         17
```



```
323 #define DTRACE_V4MAPPED_OFFSET          (sizeof (uint32_t) * 3)

325 /*
326  * The key for a thread-local variable consists of the lower 61 bits of the
327  * t_did, plus the 3 bits of the highest active interrupt above LOCK_LEVEL.
328  * We add DIF_VARIABLE_MAX to t_did to assure that the thread key is never
329  * equal to a variable identifier.  This is necessary (but not sufficient) to
330  * assure that global associative arrays never collide with thread-local
331  * variables.  To guarantee that they cannot collide, we must also define the
332  * order for keying dynamic variables.  That order is:
333  *
334  *   [ key0 ] ... [ keyn ] [ variable-key ] [ tls-key ]
335  *
336  * Because the variable-key and the tls-key are in orthogonal spaces, there is
337  * no way for a global variable key signature to match a thread-local key
338  * signature.
339  */
340 #define DTRACE_TLS_THRKEY(where) { \
341         uint_t intr = 0; \
342         uint_t actv = CPU->cpu_intr_actv >> (LOCK_LEVEL + 1); \
343         for (; actv; actv >>= 1) \
344                 intr++; \
345         ASSERT(intr < (1 << 3)); \
346         (where) = ((curthread->t_did + DIF_VARIABLE_MAX) & \
347             (((uint64_t)1 << 61) - 1)) | ((uint64_t)intr << 61); \
348 }

350 #define DT_BSWAP_8(x)   ((x) & 0xff)
351 #define DT_BSWAP_16(x)  ((DT_BSWAP_8(x) << 8) | DT_BSWAP_8((x) >> 8))
352 #define DT_BSWAP_32(x)  ((DT_BSWAP_16(x) << 16) | DT_BSWAP_16((x) >> 16))
353 #define DT_BSWAP_64(x)  ((DT_BSWAP_32(x) << 32) | DT_BSWAP_32((x) >> 32))

355 #define DT_MASK_LO 0x00000000FFFFFFFFULL

357 #define DTRACE_STORE(type, tomax, offset, what) \
358         *((type *)((uintptr_t)(tomax) + (uintptr_t)offset)) = (type)(what);

360 #ifndef __x86
361 #define DTRACE_ALIGNCHECK(addr, size, flags)                            \
362         if (addr & (size - 1)) {                                        \
363                 *flags |= CPU_DTRACE_BADALIGN;                          \
364                 cpu_core[CPU->cpu_id].cpuc_dtrace_illval = addr;        \
365                 return (0);                                             \
366         }
367 #else
368 #define DTRACE_ALIGNCHECK(addr, size, flags)
369 #endif

371 /*
372  * Test whether a range of memory starting at testaddr of size testsz falls
373  * within the range of memory described by addr, sz.  We take care to avoid
374  * problems with overflow and underflow of the unsigned quantities, and
375  * disallow all negative sizes.  Ranges of size 0 are allowed.
376  */
377 #define DTRACE_INRANGE(testaddr, testsz, baseaddr, basesz) \
378         ((testaddr) - (uintptr_t)(baseaddr) < (basesz) && \
379         (testaddr) + (testsz) - (uintptr_t)(baseaddr) <= (basesz) && \
174         ((testaddr) - (baseaddr) < (basesz) && \
175         (testaddr) + (testsz) - (baseaddr) <= (basesz) && \
380         (testaddr) + (testsz) >= (testaddr))

382 /*
383  * Test whether alloc_sz bytes will fit in the scratch region.  We isolate
384  * alloc_sz on the righthand side of the comparison in order to avoid overflow
385  * or underflow in the comparison with it.  This is simpler than the INRANGE
```

```
 386  * check above, because we know that the dtms_scratch_ptr is valid in the
 387  * range.  Allocations of size zero are allowed.
 388  */
 389 #define DTRACE_INSCRATCH(mstate, alloc_sz) \
 390         ((mstate)->dtms_scratch_base + (mstate)->dtms_scratch_size - \
 391         (mstate)->dtms_scratch_ptr >= (alloc_sz))

 393 #define DTRACE_LOADFUNC(bits)                                           \
 394 /*CSTYLED*/                                                             \
 395 uint##bits##_t                                                          \
 396 dtrace_load##bits(uintptr_t addr)                                       \
 397 {                                                                       \
 398         size_t size = bits / NBBY;                                      \
 399         /*CSTYLED*/                                                     \
 400         uint##bits##_t rval;                                            \
 401         int i;                                                          \
 402         volatile uint16_t *flags = (volatile uint16_t *)                \
 403             &cpu_core[CPU->cpu_id].cpuc_dtrace_flags;                   \
 404                                                                         \
 405         DTRACE_ALIGNCHECK(addr, size, flags);                           \
 406                                                                         \
 407         for (i = 0; i < dtrace_toxranges; i++) {                        \
 408                 if (addr >= dtrace_toxrange[i].dtt_limit)               \
 409                         continue;                                       \
 410                                                                         \
 411                 if (addr + size <= dtrace_toxrange[i].dtt_base)         \
 412                         continue;                                       \
 413                                                                         \
 414                 /*                                                      \
 415                  * This address falls within a toxic region; return 0.  \
 416                  */                                                     \
 417                 *flags |= CPU_DTRACE_BADADDR;                           \
 418                 cpu_core[CPU->cpu_id].cpuc_dtrace_illval = addr;        \
 419                 return (0);                                             \
 420         }                                                               \
 421                                                                         \
 422         *flags |= CPU_DTRACE_NOFAULT;                                   \
 423         /*CSTYLED*/                                                     \
 424         rval = *((volatile uint##bits##_t *)addr);                      \
 425         *flags &= ~CPU_DTRACE_NOFAULT;                                  \
 426                                                                         \
 427         return (!(*flags & CPU_DTRACE_FAULT) ? rval : 0);               \
 428 }

 430 #ifdef _LP64
 431 #define dtrace_loadptr  dtrace_load64
 432 #else
 433 #define dtrace_loadptr  dtrace_load32
 434 #endif

 436 #define DTRACE_DYNHASH_FREE     0
 437 #define DTRACE_DYNHASH_SINK     1
 438 #define DTRACE_DYNHASH_VALID    2

 440 #define DTRACE_MATCH_FAIL       -1
 441 #define DTRACE_MATCH_NEXT       0
 442 #define DTRACE_MATCH_DONE       1
 443 #define DTRACE_ANCHORED(probe)  ((probe)->dtpr_func[0] != '\0')
 444 #define DTRACE_STATE_ALIGN      64

 446 #define DTRACE_FLAGS2FLT(flags)                                         \
 447         (((flags) & CPU_DTRACE_BADADDR) ? DTRACEFLT_BADADDR :          \
 448         ((flags) & CPU_DTRACE_ILLOP) ? DTRACEFLT_ILLOP :               \
 449         ((flags) & CPU_DTRACE_DIVZERO) ? DTRACEFLT_DIVZERO :           \
 450         ((flags) & CPU_DTRACE_KPRIV) ? DTRACEFLT_KPRIV :               \
 451         ((flags) & CPU_DTRACE_UPRIV) ? DTRACEFLT_UPRIV :               \
```



```
 452         ((flags) & CPU_DTRACE_TUPOFLOW) ?  DTRACEFLT_TUPOFLOW :         \
 453         ((flags) & CPU_DTRACE_BADALIGN) ?  DTRACEFLT_BADALIGN :         \
 454         ((flags) & CPU_DTRACE_NOSCRATCH) ?  DTRACEFLT_NOSCRATCH :       \
 455         ((flags) & CPU_DTRACE_BADSTACK) ?  DTRACEFLT_BADSTACK :         \
 456         DTRACEFLT_UNKNOWN)

 458 #define DTRACEACT_ISSTRING(act)                                         \
 459         ((act)->dta_kind == DTRACEACT_DIFEXPR &&                        \
 460         (act)->dta_difo->dtdo_rtype.dtdt_kind == DIF_TYPE_STRING)

 462 static size_t dtrace_strlen(const char *, size_t);
 463 static dtrace_probe_t *dtrace_probe_lookup_id(dtrace_id_t id);
 464 static void dtrace_enabling_provide(dtrace_provider_t *);
 465 static int dtrace_enabling_match(dtrace_enabling_t *, int *);
 466 static void dtrace_enabling_matchall(void);
 467 static void dtrace_enabling_reap(void);
 468 static dtrace_state_t *dtrace_anon_grab(void);
 469 static uint64_t dtrace_helper(int, dtrace_mstate_t *,
 470     dtrace_state_t *, uint64_t, uint64_t);
 471 static dtrace_helpers_t *dtrace_helpers_create(proc_t *);
 472 static void dtrace_buffer_drop(dtrace_buffer_t *);
 473 static int dtrace_buffer_consumed(dtrace_buffer_t *, hrtime_t when);
 474 static intptr_t dtrace_buffer_reserve(dtrace_buffer_t *, size_t, size_t,
 475     dtrace_state_t *, dtrace_mstate_t *);
 476 static int dtrace_state_option(dtrace_state_t *, dtrace_optid_t,
 477     dtrace_optval_t);
 478 static int dtrace_ecb_create_enable(dtrace_probe_t *, void *);
 479 static void dtrace_helper_provider_destroy(dtrace_helper_provider_t *);
 480 static int dtrace_priv_proc(dtrace_state_t *, dtrace_mstate_t *);
 481 static void dtrace_getf_barrier(void);
 482 #endif /* ! codereview */

 484 /*
 485  * DTrace Probe Context Functions
 486  *
 487  * These functions are called from probe context.  Because probe context is
 488  * any context in which C may be called, arbitrarily locks may be held,
 489  * interrupts may be disabled, we may be in arbitrary dispatched state, etc.
 490  * As a result, functions called from probe context may only call other DTrace
 491  * support functions -- they may not interact at all with the system at large.
 492  * (Note that the ASSERT macro is made probe-context safe by redefining it in
 493  * terms of dtrace_assfail(), a probe-context safe function.) If arbitrary
 494  * loads are to be performed from probe context, they _must_ be in terms of
 495  * the safe dtrace_load*() variants.
 496  *
 497  * Some functions in this block are not actually called from probe context;
 498  * for these functions, there will be a comment above the function reading
 499  * "Note:  not called from probe context."
 500  */
 501 void
 502 dtrace_panic(const char *format, ...)
 503 {
 504         va_list alist;

 506         va_start(alist, format);
 507         dtrace_vpanic(format, alist);
 508         va_end(alist);
 509 }

 511 int
 512 dtrace_assfail(const char *a, const char *f, int l)
 513 {
 514         dtrace_panic("assertion failed: %s, file: %s, line: %d", a, f, l);

 516         /*
 517          * We just need something here that even the most clever compiler
```

```
518          * cannot optimize away.
519          */
520         return (a[(uintptr_t)f]);
521 }
523 /*
524  * Atomically increment a specified error counter from probe context.
525  */
526 static void
527 dtrace_error(uint32_t *counter)
528 {
529         /*
530          * Most counters stored to in probe context are per-CPU counters.
531          * However, there are some error conditions that are sufficiently
532          * arcane that they don't merit per-CPU storage.  If these counters
533          * are incremented concurrently on different CPUs, scalability will be
534          * adversely affected -- but we don't expect them to be white-hot in a
535          * correctly constructed enabling...
536          */
537         uint32_t oval, nval;
539         do {
540                 oval = *counter;
542                 if ((nval = oval + 1) == 0) {
543                         /*
544                          * If the counter would wrap, set it to 1 -- assuring
545                          * that the counter is never zero when we have seen
546                          * errors.  (The counter must be 32-bits because we
547                          * aren't guaranteed a 64-bit compare&swap operation.)
548                          * To save this code both the infamy of being fingered
549                          * by a priggish news story and the indignity of being
550                          * the target of a neo-puritan witch trial, we're
551                          * carefully avoiding any colorful description of the
552                          * likelihood of this condition -- but suffice it to
553                          * say that it is only slightly more likely than the
554                          * overflow of predicate cache IDs, as discussed in
555                          * dtrace_predicate_create().
556                          */
557                         nval = 1;
558                 }
559         } while (dtrace_cas32(counter, oval, nval) != oval);
560 }
562 /*
563  * Use the DTRACE_LOADFUNC macro to define functions for each of loading a
564  * uint8_t, a uint16_t, a uint32_t and a uint64_t.
565  */
566 DTRACE_LOADFUNC(8)
567 DTRACE_LOADFUNC(16)
568 DTRACE_LOADFUNC(32)
569 DTRACE_LOADFUNC(64)
571 static int
572 dtrace_inscratch(uintptr_t dest, size_t size, dtrace_mstate_t *mstate)
573 {
574         if (dest < mstate->dtms_scratch_base)
575                 return (0);
577         if (dest + size < dest)
578                 return (0);
580         if (dest + size > mstate->dtms_scratch_ptr)
581                 return (0);
583         return (1);
```



```
584 }
586 static int
587 dtrace_canstore_statvar(uint64_t addr, size_t sz,
588     dtrace_statvar_t **svars, int nsvars)
589 {
590         int i;
592         for (i = 0; i < nsvars; i++) {
593                 dtrace_statvar_t *svar = svars[i];
595                 if (svar == NULL || svar->dtsv_size == 0)
596                         continue;
598                 if (DTRACE_INRANGE(addr, sz, svar->dtsv_data, svar->dtsv_size))
599                         return (1);
600         }
602         return (0);
603 }
605 /*
606  * Check to see if the address is within a memory region to which a store may
607  * be issued.  This includes the DTrace scratch areas, and any DTrace variable
608  * region.  The caller of dtrace_canstore() is responsible for performing any
609  * alignment checks that are needed before stores are actually executed.
610  */
611 static int
612 dtrace_canstore(uint64_t addr, size_t sz, dtrace_mstate_t *mstate,
613     dtrace_vstate_t *vstate)
614 {
615         /*
616          * First, check to see if the address is in scratch space...
617          */
618         if (DTRACE_INRANGE(addr, sz, mstate->dtms_scratch_base,
619             mstate->dtms_scratch_size))
620                 return (1);
622         /*
623          * Now check to see if it's a dynamic variable.  This check will pick
624          * up both thread-local variables and any global dynamically-allocated
625          * variables.
626          */
627         if (DTRACE_INRANGE(addr, sz, vstate->dtvs_dynvars.dtds_base,
276         if (DTRACE_INRANGE(addr, sz, (uintptr_t)vstate->dtvs_dynvars.dtds_base,
628             vstate->dtvs_dynvars.dtds_size)) {
629                 dtrace_dstate_t *dstate = &vstate->dtvs_dynvars;
630                 uintptr_t base = (uintptr_t)dstate->dtds_base +
631                     (dstate->dtds_hashsize * sizeof (dtrace_dynhash_t));
632                 uintptr_t chunkoffs;
634                 /*
635                  * Before we assume that we can store here, we need to make
636                  * sure that it isn't in our metadata -- storing to our
637                  * dynamic variable metadata would corrupt our state.  For
638                  * the range to not include any dynamic variable metadata,
639                  * it must:
640                  *
641                  *      (1) Start above the hash table that is at the base of
642                  *      the dynamic variable space
643                  *
644                  *      (2) Have a starting chunk offset that is beyond the
645                  *      dtrace_dynvar_t that is at the base of every chunk
646                  *
647                  *      (3) Not span a chunk boundary
648                  *
```

```
649                  */
650                 if (addr < base)
651                         return (0);

653                 chunkoffs = (addr - base) % dstate->dtds_chunksize;

655                 if (chunkoffs < sizeof (dtrace_dynvar_t))
656                         return (0);

658                 if (chunkoffs + sz > dstate->dtds_chunksize)
659                         return (0);

661                 return (1);
662         }

664         /*
665          * Finally, check the static local and global variables.  These checks
666          * take the longest, so we perform them last.
667          */
668         if (dtrace_canstore_statvar(addr, sz,
669             vstate->dtvs_locals, vstate->dtvs_nlocals))
670                 return (1);

672         if (dtrace_canstore_statvar(addr, sz,
673             vstate->dtvs_globals, vstate->dtvs_nglobals))
674                 return (1);

676         return (0);
677 }


680 /*
681  * Convenience routine to check to see if the address is within a memory
682  * region in which a load may be issued given the user's privilege level;
683  * if not, it sets the appropriate error flags and loads 'addr' into the
684  * illegal value slot.
685  *
686  * DTrace subroutines (DIF_SUBR_*) should use this helper to implement
687  * appropriate memory access protection.
688  */
689 static int
690 dtrace_canload(uint64_t addr, size_t sz, dtrace_mstate_t *mstate,
691     dtrace_vstate_t *vstate)
692 {
693         volatile uintptr_t *illval = &cpu_core[CPU->cpu_id].cpuc_dtrace_illval;
694         file_t *fp;
695 #endif /* ! codereview */

697         /*
698          * If we hold the privilege to read from kernel memory, then
699          * everything is readable.
700          */
701         if ((mstate->dtms_access & DTRACE_ACCESS_KERNEL) != 0)
702                 return (1);

704         /*
705          * You can obviously read that which you can store.
706          */
707         if (dtrace_canstore(addr, sz, mstate, vstate))
708                 return (1);

710         /*
711          * We're allowed to read from our own string table.
712          */
713         if (DTRACE_INRANGE(addr, sz, mstate->dtms_difo->dtdo_strtab,
343         if (DTRACE_INRANGE(addr, sz, (uintptr_t)mstate->dtms_difo->dtdo_strtab,
```



```
714             mstate->dtms_difo->dtdo_strlen))
715                 return (1);

717         if (vstate->dtvs_state != NULL &&
718             dtrace_priv_proc(vstate->dtvs_state, mstate)) {
719                 proc_t *p;

721                 /*
722                  * When we have privileges to the current process, there are
723                  * several context-related kernel structures that are safe to
724                  * read, even absent the privilege to read from kernel memory.
725                  * These reads are safe because these structures contain only
726                  * state that (1) we're permitted to read, (2) is harmless or
727                  * (3) contains pointers to additional kernel state that we're
728                  * not permitted to read (and as such, do not present an
729                  * opportunity for privilege escalation).  Finally (and
730                  * critically), because of the nature of their relation with
731                  * the current thread context, the memory associated with these
732                  * structures cannot change over the duration of probe context,
733                  * and it is therefore impossible for this memory to be
734                  * deallocated and reallocated as something else while it's
735                  * being operated upon.
736                  */
737                 if (DTRACE_INRANGE(addr, sz, curthread, sizeof (kthread_t)))
738                         return (1);

740                 if ((p = curthread->t_procp) != NULL && DTRACE_INRANGE(addr,
741                     sz, curthread->t_procp, sizeof (proc_t))) {
742                         return (1);
743                 }

745                 if (curthread->t_cred != NULL && DTRACE_INRANGE(addr, sz,
746                     curthread->t_cred, sizeof (cred_t))) {
747                         return (1);
748                 }

750                 if (p != NULL && p->p_pidp != NULL && DTRACE_INRANGE(addr, sz,
751                     &(p->p_pidp->pid_id), sizeof (pid_t))) {
752                         return (1);
753                 }

755                 if (curthread->t_cpu != NULL && DTRACE_INRANGE(addr, sz,
756                     curthread->t_cpu, offsetof(cpu_t, cpu_pause_thread))) {
757                         return (1);
758                 }
759         }

761         if ((fp = mstate->dtms_getf) != NULL) {
762                 uintptr_t psz = sizeof (void *);
763                 vnode_t *vp;
764                 vnodeops_t *op;

766                 /*
767                  * When getf() returns a file_t, the enabling is implicitly
768                  * granted the (transient) right to read the returned file_t
769                  * as well as the v_path and v_op->vnop_name of the underlying
770                  * vnode.  These accesses are allowed after a successful
771                  * getf() because the members that they refer to cannot change
772                  * once set -- and the barrier logic in the kernel's closef()
773                  * path assures that the file_t and its referenced vode_t
774                  * cannot themselves be stale (that is, it impossible for
775                  * either dtms_getf itself or its f_vnode member to reference
776                  * freed memory).
777                  */
778                 if (DTRACE_INRANGE(addr, sz, fp, sizeof (file_t)))
779                         return (1);
```

```
781                 if ((vp = fp->f_vnode) != NULL) {
782                         if (DTRACE_INRANGE(addr, sz, &vp->v_path, psz))
783                                 return (1);
784
785                         if (vp->v_path != NULL && DTRACE_INRANGE(addr, sz,
786                             vp->v_path, strlen(vp->v_path) + 1)) {
787                                 return (1);
788                         }
789
790                         if (DTRACE_INRANGE(addr, sz, &vp->v_op, psz))
791                                 return (1);
792
793                         if ((op = vp->v_op) != NULL &&
794                             DTRACE_INRANGE(addr, sz, &op->vnop_name, psz)) {
795                                 return (1);
796                         }
797
798                         if (op != NULL && op->vnop_name != NULL &&
799                             DTRACE_INRANGE(addr, sz, op->vnop_name,
800                             strlen(op->vnop_name) + 1)) {
801                                 return (1);
802                         }
803                 }
804         }
805
806 #endif /* ! codereview */
807         DTRACE_CPUFLAG_SET(CPU_DTRACE_KPRIV);
808         *illval = addr;
809         return (0);
810 }
811
812 /*
813  * Convenience routine to check to see if a given string is within a memory
814  * region in which a load may be issued given the user's privilege level;
815  * this exists so that we don't need to issue unnecessary dtrace_strlen()
816  * calls in the event that the user has all privileges.
817  */
818 static int
819 dtrace_strcanload(uint64_t addr, size_t sz, dtrace_mstate_t *mstate,
820     dtrace_vstate_t *vstate)
821 {
822         size_t strsz;
823
824         /*
825          * If we hold the privilege to read from kernel memory, then
826          * everything is readable.
827          */
828         if ((mstate->dtms_access & DTRACE_ACCESS_KERNEL) != 0)
829                 return (1);
830
831         strsz = 1 + dtrace_strlen((char *)(uintptr_t)addr, sz);
832         if (dtrace_canload(addr, strsz, mstate, vstate))
833                 return (1);
834
835         return (0);
836 }
837
838 /*
839  * Convenience routine to check to see if a given variable is within a memory
840  * region in which a load may be issued given the user's privilege level.
841  */
842 static int
843 dtrace_vcanload(void *src, dtrace_diftype_t *type, dtrace_mstate_t *mstate,
844     dtrace_vstate_t *vstate)
845 {
```



```
846         size_t sz;
847         ASSERT(type->dtdt_flags & DIF_TF_BYREF);
848
849         /*
850          * If we hold the privilege to read from kernel memory, then
851          * everything is readable.
852          */
853         if ((mstate->dtms_access & DTRACE_ACCESS_KERNEL) != 0)
854                 return (1);
855
856         if (type->dtdt_kind == DIF_TYPE_STRING)
857                 sz = dtrace_strlen(src,
858                     vstate->dtvs_state->dts_options[DTRACEOPT_STRSIZE]) + 1;
859         else
860                 sz = type->dtdt_size;
861
862         return (dtrace_canload((uintptr_t)src, sz, mstate, vstate));
863 }
864
865 /*
866  * Compare two strings using safe loads.
867  */
868 static int
869 dtrace_strncmp(char *s1, char *s2, size_t limit)
870 {
871         uint8_t c1, c2;
872         volatile uint16_t *flags;
873
874         if (s1 == s2 || limit == 0)
875                 return (0);
876
877         flags = (volatile uint16_t *)&cpu_core[CPU->cpu_id].cpuc_dtrace_flags;
878
879         do {
880                 if (s1 == NULL) {
881                         c1 = '\0';
882                 } else {
883                         c1 = dtrace_load8((uintptr_t)s1++);
884                 }
885
886                 if (s2 == NULL) {
887                         c2 = '\0';
888                 } else {
889                         c2 = dtrace_load8((uintptr_t)s2++);
890                 }
891
892                 if (c1 != c2)
893                         return (c1 - c2);
894         } while (--limit && c1 != '\0' && !(*flags & CPU_DTRACE_FAULT));
895
896         return (0);
897 }
898
899 /*
900  * Compute strlen(s) for a string using safe memory accesses.  The additional
901  * len parameter is used to specify a maximum length to ensure completion.
902  */
903 static size_t
904 dtrace_strlen(const char *s, size_t lim)
905 {
906         uint_t len;
907
908         for (len = 0; len != lim; len++) {
909                 if (dtrace_load8((uintptr_t)s++) == '\0')
910                         break;
911         }
```

```
913         return (len);
914 }

916 /*
917  * Check if an address falls within a toxic region.
918  */
919 static int
920 dtrace_istoxic(uintptr_t kaddr, size_t size)
921 {
922         uintptr_t taddr, tsize;
923         int i;

925         for (i = 0; i < dtrace_toxranges; i++) {
926                 taddr = dtrace_toxrange[i].dtt_base;
927                 tsize = dtrace_toxrange[i].dtt_limit - taddr;

929                 if (kaddr - taddr < tsize) {
930                         DTRACE_CPUFLAG_SET(CPU_DTRACE_BADADDR);
931                         cpu_core[CPU->cpu_id].cpuc_dtrace_illval = kaddr;
932                         return (1);
933                 }

935                 if (taddr - kaddr < size) {
936                         DTRACE_CPUFLAG_SET(CPU_DTRACE_BADADDR);
937                         cpu_core[CPU->cpu_id].cpuc_dtrace_illval = taddr;
938                         return (1);
939                 }
940         }

942         return (0);
943 }

945 /*
946  * Copy src to dst using safe memory accesses.  The src is assumed to be unsafe
947  * memory specified by the DIF program.  The dst is assumed to be safe memory
948  * that we can store to directly because it is managed by DTrace.  As with
949  * standard bcopy, overlapping copies are handled properly.
950  */
951 static void
952 dtrace_bcopy(const void *src, void *dst, size_t len)
953 {
954         if (len != 0) {
955                 uint8_t *s1 = dst;
956                 const uint8_t *s2 = src;

958                 if (s1 <= s2) {
959                         do {
960                                 *s1++ = dtrace_load8((uintptr_t)s2++);
961                         } while (--len != 0);
962                 } else {
963                         s2 += len;
964                         s1 += len;

966                         do {
967                                 *--s1 = dtrace_load8((uintptr_t)--s2);
968                         } while (--len != 0);
969                 }
970         }
971 }

973 /*
974  * Copy src to dst using safe memory accesses, up to either the specified
975  * length, or the point that a nul byte is encountered.  The src is assumed to
976  * be unsafe memory specified by the DIF program.  The dst is assumed to be
977  * safe memory that we can store to directly because it is managed by DTrace.
```



```
978  * Unlike dtrace_bcopy(), overlapping regions are not handled.
979  */
980 static void
981 dtrace_strcpy(const void *src, void *dst, size_t len)
982 {
983         if (len != 0) {
984                 uint8_t *s1 = dst, c;
985                 const uint8_t *s2 = src;

987                 do {
988                         *s1++ = c = dtrace_load8((uintptr_t)s2++);
989                 } while (--len != 0 && c != '\0');
990         }
991 }

993 /*
994  * Copy src to dst, deriving the size and type from the specified (BYREF)
995  * variable type.  The src is assumed to be unsafe memory specified by the DIF
996  * program.  The dst is assumed to be DTrace variable memory that is of the
997  * specified type; we assume that we can store to directly.
998  */
999 static void
1000 dtrace_vcopy(void *src, void *dst, dtrace_diftype_t *type)
1001 {
1002         ASSERT(type->dtdt_flags & DIF_TF_BYREF);

1004         if (type->dtdt_kind == DIF_TYPE_STRING) {
1005                 dtrace_strcpy(src, dst, type->dtdt_size);
1006         } else {
1007                 dtrace_bcopy(src, dst, type->dtdt_size);
1008         }
1009 }

1011 /*
1012  * Compare s1 to s2 using safe memory accesses.  The s1 data is assumed to be
1013  * unsafe memory specified by the DIF program.  The s2 data is assumed to be
1014  * safe memory that we can access directly because it is managed by DTrace.
1015  */
1016 static int
1017 dtrace_bcmp(const void *s1, const void *s2, size_t len)
1018 {
1019         volatile uint16_t *flags;

1021         flags = (volatile uint16_t *)&cpu_core[CPU->cpu_id].cpuc_dtrace_flags;

1023         if (s1 == s2)
1024                 return (0);

1026         if (s1 == NULL || s2 == NULL)
1027                 return (1);

1029         if (s1 != s2 && len != 0) {
1030                 const uint8_t *ps1 = s1;
1031                 const uint8_t *ps2 = s2;

1033                 do {
1034                         if (dtrace_load8((uintptr_t)ps1++) != *ps2++)
1035                                 return (1);
1036                 } while (--len != 0 && !(*flags & CPU_DTRACE_FAULT));
1037         }
1038         return (0);
1039 }

1041 /*
1042  * Zero the specified region using a simple byte-by-byte loop.  Note that this
1043  * is for safe DTrace-managed memory only.
```

```
 */
static void
dtrace_bzero(void *dst, size_t len)
{
        uchar_t *cp;

        for (cp = dst; len != 0; len--)
                *cp++ = 0;
}

static void
dtrace_add_128(uint64_t *addend1, uint64_t *addend2, uint64_t *sum)
{
        uint64_t result[2];

        result[0] = addend1[0] + addend2[0];
        result[1] = addend1[1] + addend2[1] +
            (result[0] < addend1[0] || result[0] < addend2[0] ? 1 : 0);

        sum[0] = result[0];
        sum[1] = result[1];
}

/*
 * Shift the 128-bit value in a by b. If b is positive, shift left.
 * If b is negative, shift right.
 */
static void
dtrace_shift_128(uint64_t *a, int b)
{
        uint64_t mask;

        if (b == 0)
                return;

        if (b < 0) {
                b = -b;
                if (b >= 64) {
                        a[0] = a[1] >> (b - 64);
                        a[1] = 0;
                } else {
                        a[0] >>= b;
                        mask = 1LL << (64 - b);
                        mask -= 1;
                        a[0] |= ((a[1] & mask) << (64 - b));
                        a[1] >>= b;
                }
        } else {
                if (b >= 64) {
                        a[1] = a[0] << (b - 64);
                        a[0] = 0;
                } else {
                        a[1] <<= b;
                        mask = a[0] >> (64 - b);
                        a[1] |= mask;
                        a[0] <<= b;
                }
        }
}

/*
 * The basic idea is to break the 2 64-bit values into 4 32-bit values,
 * use native multiplication on those, and then re-combine into the
 * resulting 128-bit value.
 *
 * (hi1 << 32 + lo1) * (hi2 << 32 + lo2) =
```



```
 *     hi1 * hi2 << 64 +
 *     hi1 * lo2 << 32 +
 *     hi2 * lo1 << 32 +
 *     lo1 * lo2
 */
static void
dtrace_multiply_128(uint64_t factor1, uint64_t factor2, uint64_t *product)
{
        uint64_t hi1, hi2, lo1, lo2;
        uint64_t tmp[2];

        hi1 = factor1 >> 32;
        hi2 = factor2 >> 32;

        lo1 = factor1 & DT_MASK_LO;
        lo2 = factor2 & DT_MASK_LO;

        product[0] = lo1 * lo2;
        product[1] = hi1 * hi2;

        tmp[0] = hi1 * lo2;
        tmp[1] = 0;
        dtrace_shift_128(tmp, 32);
        dtrace_add_128(product, tmp, product);

        tmp[0] = hi2 * lo1;
        tmp[1] = 0;
        dtrace_shift_128(tmp, 32);
        dtrace_add_128(product, tmp, product);
}

/*
 * This privilege check should be used by actions and subroutines to
 * verify that the user credentials of the process that enabled the
 * invoking ECB match the target credentials
 */
static int
dtrace_priv_proc_common_user(dtrace_state_t *state)
{
        cred_t *cr, *s_cr = state->dts_cred.dcr_cred;

        /*
         * We should always have a non-NULL state cred here, since if cred
         * is null (anonymous tracing), we fast-path bypass this routine.
         */
        ASSERT(s_cr != NULL);

        if ((cr = CRED()) != NULL &&
            s_cr->cr_uid == cr->cr_uid &&
            s_cr->cr_uid == cr->cr_ruid &&
            s_cr->cr_uid == cr->cr_suid &&
            s_cr->cr_gid == cr->cr_gid &&
            s_cr->cr_gid == cr->cr_rgid &&
            s_cr->cr_gid == cr->cr_sgid)
                return (1);

        return (0);
}

/*
 * This privilege check should be used by actions and subroutines to
 * verify that the zone of the process that enabled the invoking ECB
 * matches the target credentials
 */
static int
dtrace_priv_proc_common_zone(dtrace_state_t *state)
```

```
1176 {
1177         cred_t *cr, *s_cr = state->dts_cred.dcr_cred;

1179         /*
1180          * We should always have a non-NULL state cred here, since if cred
1181          * is null (anonymous tracing), we fast-path bypass this routine.
1182          */
1183         ASSERT(s_cr != NULL);

1185         if ((cr = CRED()) != NULL && s_cr->cr_zone == cr->cr_zone)
 347         if ((cr = CRED()) != NULL &&
 348             s_cr->cr_zone == cr->cr_zone)
1186                 return (1);

1188         return (0);
1189 }
_____unchanged_portion_omitted_

1287 /*
1288  * Determine if the dte_cond of the specified ECB allows for processing of
1289  * the current probe to continue.  Note that this routine may allow continued
1290  * processing, but with access(es) stripped from the mstate's dtms_access
1291  * field.
1292  */
1293 static int
1294 dtrace_priv_probe(dtrace_state_t *state, dtrace_mstate_t *mstate,
1295     dtrace_ecb_t *ecb)
1296 {
1297         dtrace_probe_t *probe = ecb->dte_probe;
1298         dtrace_provider_t *prov = probe->dtpr_provider;
1299         dtrace_pops_t *pops = &prov->dtpv_pops;
1300         int mode = DTRACE_MODE_NOPRIV_DROP;

1302         ASSERT(ecb->dte_cond);

1304         if (pops->dtps_mode != NULL) {
1305                 mode = pops->dtps_mode(prov->dtpv_arg,
1306                     probe->dtpr_id, probe->dtpr_arg);

1308                 ASSERT(mode & (DTRACE_MODE_USER | DTRACE_MODE_KERNEL));
1309                 ASSERT(mode & (DTRACE_MODE_NOPRIV_RESTRICT |
1310                     DTRACE_MODE_NOPRIV_DROP));
 471                 ASSERT((mode & DTRACE_MODE_USER) ||
 472                     (mode & DTRACE_MODE_KERNEL));
 473                 ASSERT((mode & DTRACE_MODE_NOPRIV_RESTRICT) ||
 474                     (mode & DTRACE_MODE_NOPRIV_DROP));
1311         }

1313         /*
1314          * If the dte_cond bits indicate that this consumer is only allowed to
1315          * see user-mode firings of this probe, check that the probe was fired
1316          * while in a user context.  If that's not the case, use the policy
1317          * specified by the provider to determine if we drop the probe or
1318          * merely restrict operation.
 479          * see user-mode firings of this probe, call the provider's dtps_mode()
 480          * entry point to check that the probe was fired while in a user
 481          * context.  If that's not the case, use the policy specified by the
 482          * provider to determine if we drop the probe or merely restrict
 483          * operation.
1319          */
1320         if (ecb->dte_cond & DTRACE_COND_USERMODE) {
1321                 ASSERT(mode != DTRACE_MODE_NOPRIV_DROP);

1323                 if (!(mode & DTRACE_MODE_USER)) {
1324                         if (mode & DTRACE_MODE_NOPRIV_DROP)
1325                                 return (0);
```



```
1327                         mstate->dtms_access &= ~DTRACE_ACCESS_ARGS;
1328                 }
1329         }

1331         /*
1332          * This is more subtle than it looks. We have to be absolutely certain
1333          * that CRED() isn't going to change out from under us so it's only
1334          * legit to examine that structure if we're in constrained situations.
1335          * Currently, the only times we'll this check is if a non-super-user
1336          * has enabled the profile or syscall providers -- providers that
1337          * allow visibility of all processes. For the profile case, the check
1338          * above will ensure that we're examining a user context.
1339          */
1340         if (ecb->dte_cond & DTRACE_COND_OWNER) {
1341                 cred_t *cr;
1342                 cred_t *s_cr = state->dts_cred.dcr_cred;
1343                 proc_t *proc;

1345                 ASSERT(s_cr != NULL);

1347                 if ((cr = CRED()) == NULL ||
1348                     s_cr->cr_uid != cr->cr_uid ||
1349                     s_cr->cr_uid != cr->cr_ruid ||
1350                     s_cr->cr_uid != cr->cr_suid ||
1351                     s_cr->cr_gid != cr->cr_gid ||
1352                     s_cr->cr_gid != cr->cr_rgid ||
1353                     s_cr->cr_gid != cr->cr_sgid ||
1354                     (proc = ttoproc(curthread)) == NULL ||
1355                     (proc->p_flag & SNOCD)) {
1356                         if (mode & DTRACE_MODE_NOPRIV_DROP)
1357                                 return (0);

1359                         mstate->dtms_access &= ~DTRACE_ACCESS_PROC;
1360                 }
1361         }

1363         /*
1364          * If our dte_cond is set to DTRACE_COND_ZONEOWNER and we are not
1365          * in our zone, check to see if our mode policy is to restrict rather
1366          * than to drop; if to restrict, strip away both DTRACE_ACCESS_PROC
1367          * and DTRACE_ACCESS_ARGS
1368          */
1369         if (ecb->dte_cond & DTRACE_COND_ZONEOWNER) {
1370                 cred_t *cr;
1371                 cred_t *s_cr = state->dts_cred.dcr_cred;

1373                 ASSERT(s_cr != NULL);

1375                 if ((cr = CRED()) == NULL ||
1376                     s_cr->cr_zone->zone_id != cr->cr_zone->zone_id) {
1377                         if (mode & DTRACE_MODE_NOPRIV_DROP)
1378                                 return (0);

1380                         mstate->dtms_access &=
1381                             ~(DTRACE_ACCESS_PROC | DTRACE_ACCESS_ARGS);
1382                 }
1383         }

1385         /*
1386          * By merits of being in this code path at all, we have limited
1387          * privileges.  If the provider has indicated that limited privileges
1388          * are to denote restricted operation, strip off the ability to access
1389          * arguments.
1390          */
1391         if (mode & DTRACE_MODE_LIMITEDPRIV_RESTRICT)
```

```
1392                 mstate->dtms_access &= ~DTRACE_ACCESS_ARGS;

1394 #endif /* ! codereview */
1395         return (1);
1396 }

1398 /*
1399  * Note:  not called from probe context.  This function is called
1400  * asynchronously (and at a regular interval) from outside of probe context to
1401  * clean the dirty dynamic variable lists on all CPUs.  Dynamic variable
1402  * cleaning is explained in detail in <sys/dtrace_impl.h>.
1403  */
1404 void
1405 dtrace_dynvar_clean(dtrace_dstate_t *dstate)
1406 {
1407         dtrace_dynvar_t *dirty;
1408         dtrace_dstate_percpu_t *dcpu;
1409         dtrace_dynvar_t **rinsep;
1410         int i, j, work = 0;

1412         for (i = 0; i < NCPU; i++) {
1413                 dcpu = &dstate->dtds_percpu[i];
1414                 rinsep = &dcpu->dtdsc_rinsing;

1416                 /*
1417                  * If the dirty list is NULL, there is no dirty work to do.
1418                  */
1419                 if (dcpu->dtdsc_dirty == NULL)
1420                         continue;

1422                 if (dcpu->dtdsc_rinsing != NULL) {
1423                         /*
1424                          * If the rinsing list is non-NULL, then it is because
1425                          * this CPU was selected to accept another CPU's
1426                          * dirty list -- and since that time, dirty buffers
1427                          * have accumulated.  This is a highly unlikely
1428                          * condition, but we choose to ignore the dirty
1429                          * buffers -- they'll be picked up a future cleanse.
1430                          */
1431                         continue;
1432                 }

1434                 if (dcpu->dtdsc_clean != NULL) {
1435                         /*
1436                          * If the clean list is non-NULL, then we're in a
1437                          * situation where a CPU has done deallocations (we
1438                          * have a non-NULL dirty list) but no allocations (we
1439                          * also have a non-NULL clean list).  We can't simply
1440                          * move the dirty list into the clean list on this
1441                          * CPU, yet we also don't want to allow this condition
1442                          * to persist, lest a short clean list prevent a
1443                          * massive dirty list from being cleaned (which in
1444                          * turn could lead to otherwise avoidable dynamic
1445                          * drops).  To deal with this, we look for some CPU
1446                          * with a NULL clean list, NULL dirty list, and NULL
1447                          * rinsing list -- and then we borrow this CPU to
1448                          * rinse our dirty list.
1449                          */
1450                         for (j = 0; j < NCPU; j++) {
1451                                 dtrace_dstate_percpu_t *rinser;

1453                                 rinser = &dstate->dtds_percpu[j];

1455                                 if (rinser->dtdsc_rinsing != NULL)
1456                                         continue;
```



```
1458                                 if (rinser->dtdsc_dirty != NULL)
1459                                         continue;

1461                                 if (rinser->dtdsc_clean != NULL)
1462                                         continue;

1464                                 rinsep = &rinser->dtdsc_rinsing;
1465                                 break;
1466                         }

1468                         if (j == NCPU) {
1469                                 /*
1470                                  * We were unable to find another CPU that
1471                                  * could accept this dirty list -- we are
1472                                  * therefore unable to clean it now.
1473                                  */
1474                                 dtrace_dynvar_failclean++;
1475                                 continue;
1476                         }
1477                 }

1479                 work = 1;

1481                 /*
1482                  * Atomically move the dirty list aside.
1483                  */
1484                 do {
1485                         dirty = dcpu->dtdsc_dirty;

1487                         /*
1488                          * Before we zap the dirty list, set the rinsing list.
1489                          * (This allows for a potential assertion in
1490                          * dtrace_dynvar():  if a free dynamic variable appears
1491                          * on a hash chain, either the dirty list or the
1492                          * rinsing list for some CPU must be non-NULL.)
1493                          */
1494                         *rinsep = dirty;
1495                         dtrace_membar_producer();
1496                 } while (dtrace_casptr(&dcpu->dtdsc_dirty,
1497                     dirty, NULL) != dirty);
1498         }

1500         if (!work) {
1501                 /*
1502                  * We have no work to do; we can simply return.
1503                  */
1504                 return;
1505         }

1507         dtrace_sync();

1509         for (i = 0; i < NCPU; i++) {
1510                 dcpu = &dstate->dtds_percpu[i];

1512                 if (dcpu->dtdsc_rinsing == NULL)
1513                         continue;

1515                 /*
1516                  * We are now guaranteed that no hash chain contains a pointer
1517                  * into this dirty list; we can make it clean.
1518                  */
1519                 ASSERT(dcpu->dtdsc_clean == NULL);
1520                 dcpu->dtdsc_clean = dcpu->dtdsc_rinsing;
1521                 dcpu->dtdsc_rinsing = NULL;
1522         }
```

```
1524         /*
1525          * Before we actually set the state to be DTRACE_DSTATE_CLEAN, make
1526          * sure that all CPUs have seen all of the dtdsc_clean pointers.
1527          * This prevents a race whereby a CPU incorrectly decides that
1528          * the state should be something other than DTRACE_DSTATE_CLEAN
1529          * after dtrace_dynvar_clean() has completed.
1530          */
1531         dtrace_sync();

1533         dstate->dtds_state = DTRACE_DSTATE_CLEAN;
1534 }

1536 /*
1537  * Depending on the value of the op parameter, this function looks-up,
1538  * allocates or deallocates an arbitrarily-keyed dynamic variable.  If an
1539  * allocation is requested, this function will return a pointer to a
1540  * dtrace_dynvar_t corresponding to the allocated variable -- or NULL if no
1541  * variable can be allocated.  If NULL is returned, the appropriate counter
1542  * will be incremented.
1543  */
1544 dtrace_dynvar_t *
1545 dtrace_dynvar(dtrace_dstate_t *dstate, uint_t nkeys,
1546     dtrace_key_t *key, size_t dsize, dtrace_dynvar_op_t op,
1547     dtrace_mstate_t *mstate, dtrace_vstate_t *vstate)
1548 {
1549         uint64_t hashval = DTRACE_DYNHASH_VALID;
1550         dtrace_dynhash_t *hash = dstate->dtds_hash;
1551         dtrace_dynvar_t *free, *new_free, *next, *dvar, *start, *prev = NULL;
1552         processorid_t me = CPU->cpu_id, cpu = me;
1553         dtrace_dstate_percpu_t *dcpu = &dstate->dtds_percpu[me];
1554         size_t bucket, ksize;
1555         size_t chunksize = dstate->dtds_chunksize;
1556         uintptr_t kdata, lock, nstate;
1557         uint_t i;

1559         ASSERT(nkeys != 0);

1561         /*
1562          * Hash the key.  As with aggregations, we use Jenkins' "One-at-a-time"
1563          * algorithm.  For the by-value portions, we perform the algorithm in
1564          * 16-bit chunks (as opposed to 8-bit chunks).  This speeds things up a
1565          * bit, and seems to have only a minute effect on distribution.  For
1566          * the by-reference data, we perform "One-at-a-time" iterating (safely)
1567          * over each referenced byte.  It's painful to do this, but it's much
1568          * better than pathological hash distribution.  The efficacy of the
1569          * hashing algorithm (and a comparison with other algorithms) may be
1570          * found by running the ::dtrace_dynstat MDB dcmd.
1571          */
1572         for (i = 0; i < nkeys; i++) {
1573                 if (key[i].dttk_size == 0) {
1574                         uint64_t val = key[i].dttk_value;

1576                         hashval += (val >> 48) & 0xffff;
1577                         hashval += (hashval << 10);
1578                         hashval ^= (hashval >> 6);

1580                         hashval += (val >> 32) & 0xffff;
1581                         hashval += (hashval << 10);
1582                         hashval ^= (hashval >> 6);

1584                         hashval += (val >> 16) & 0xffff;
1585                         hashval += (hashval << 10);
1586                         hashval ^= (hashval >> 6);

1588                         hashval += val & 0xffff;
1589                         hashval += (hashval << 10);
```



```
1590                         hashval ^= (hashval >> 6);
1591                 } else {
1592                         /*
1593                          * This is incredibly painful, but it beats the hell
1594                          * out of the alternative.
1595                          */
1596                         uint64_t j, size = key[i].dttk_size;
1597                         uintptr_t base = (uintptr_t)key[i].dttk_value;

1599                         if (!dtrace_canload(base, size, mstate, vstate))
1600                                 break;

1602                         for (j = 0; j < size; j++) {
1603                                 hashval += dtrace_load8(base + j);
1604                                 hashval += (hashval << 10);
1605                                 hashval ^= (hashval >> 6);
1606                         }
1607                 }
1608         }

1610         if (DTRACE_CPUFLAG_ISSET(CPU_DTRACE_FAULT))
1611                 return (NULL);

1613         hashval += (hashval << 3);
1614         hashval ^= (hashval >> 11);
1615         hashval += (hashval << 15);

1617         /*
1618          * There is a remote chance (ideally, 1 in 2^31) that our hashval
1619          * comes out to be one of our two sentinel hash values.  If this
1620          * actually happens, we set the hashval to be a value known to be a
1621          * non-sentinel value.
1622          */
1623         if (hashval == DTRACE_DYNHASH_FREE || hashval == DTRACE_DYNHASH_SINK)
1624                 hashval = DTRACE_DYNHASH_VALID;

1626         /*
1627          * Yes, it's painful to do a divide here.  If the cycle count becomes
1628          * important here, tricks can be pulled to reduce it.  (However, it's
1629          * critical that hash collisions be kept to an absolute minimum;
1630          * they're much more painful than a divide.)  It's better to have a
1631          * solution that generates few collisions and still keeps things
1632          * relatively simple.
1633          */
1634         bucket = hashval % dstate->dtds_hashsize;

1636         if (op == DTRACE_DYNVAR_DEALLOC) {
1637                 volatile uintptr_t *lockp = &hash[bucket].dtdh_lock;

1639                 for (;;) {
1640                         while ((lock = *lockp) & 1)
1641                                 continue;

1643                         if (dtrace_casptr((void *)lockp,
1644                             (void *)lock, (void *)(lock + 1)) == (void *)lock)
1645                                 break;
1646                 }

1648                 dtrace_membar_producer();
1649         }

1651 top:
1652         prev = NULL;
1653         lock = hash[bucket].dtdh_lock;

1655         dtrace_membar_consumer();
```

```
1657         start = hash[bucket].dtdh_chain;
1658         ASSERT(start != NULL && (start->dtdv_hashval == DTRACE_DYNHASH_SINK ||
1659             start->dtdv_hashval != DTRACE_DYNHASH_FREE ||
1660             op != DTRACE_DYNVAR_DEALLOC));
1661
1662         for (dvar = start; dvar != NULL; dvar = dvar->dtdv_next) {
1663                 dtrace_tuple_t *dtuple = &dvar->dtdv_tuple;
1664                 dtrace_key_t *dkey = &dtuple->dtt_key[0];
1665
1666                 if (dvar->dtdv_hashval != hashval) {
1667                         if (dvar->dtdv_hashval == DTRACE_DYNHASH_SINK) {
1668                                 /*
1669                                  * We've reached the sink, and therefore the
1670                                  * end of the hash chain; we can kick out of
1671                                  * the loop knowing that we have seen a valid
1672                                  * snapshot of state.
1673                                  */
1674                                 ASSERT(dvar->dtdv_next == NULL);
1675                                 ASSERT(dvar == &dtrace_dynhash_sink);
1676                                 break;
1677                         }
1678
1679                         if (dvar->dtdv_hashval == DTRACE_DYNHASH_FREE) {
1680                                 /*
1681                                  * We've gone off the rails:  somewhere along
1682                                  * the line, one of the members of this hash
1683                                  * chain was deleted.  Note that we could also
1684                                  * detect this by simply letting this loop run
1685                                  * to completion, as we would eventually hit
1686                                  * the end of the dirty list.  However, we
1687                                  * want to avoid running the length of the
1688                                  * dirty list unnecessarily (it might be quite
1689                                  * long), so we catch this as early as
1690                                  * possible by detecting the hash marker.  In
1691                                  * this case, we simply set dvar to NULL and
1692                                  * break; the conditional after the loop will
1693                                  * send us back to top.
1694                                  */
1695                                 dvar = NULL;
1696                                 break;
1697                         }
1698
1699                         goto next;
1700                 }
1701
1702                 if (dtuple->dtt_nkeys != nkeys)
1703                         goto next;
1704
1705                 for (i = 0; i < nkeys; i++, dkey++) {
1706                         if (dkey->dttk_size != key[i].dttk_size)
1707                                 goto next; /* size or type mismatch */
1708
1709                         if (dkey->dttk_size != 0) {
1710                                 if (dtrace_bcmp(
1711                                     (void *)(uintptr_t)key[i].dttk_value,
1712                                     (void *)(uintptr_t)dkey->dttk_value,
1713                                     dkey->dttk_size))
1714                                         goto next;
1715                         } else {
1716                                 if (dkey->dttk_value != key[i].dttk_value)
1717                                         goto next;
1718                         }
1719                 }
1720
1721                 if (op != DTRACE_DYNVAR_DEALLOC)
```



```
1722                         return (dvar);
1723
1724                 ASSERT(dvar->dtdv_next == NULL ||
1725                     dvar->dtdv_next->dtdv_hashval != DTRACE_DYNHASH_FREE);
1726
1727                 if (prev != NULL) {
1728                         ASSERT(hash[bucket].dtdh_chain != dvar);
1729                         ASSERT(start != dvar);
1730                         ASSERT(prev->dtdv_next == dvar);
1731                         prev->dtdv_next = dvar->dtdv_next;
1732                 } else {
1733                         if (dtrace_casptr(&hash[bucket].dtdh_chain,
1734                             start, dvar->dtdv_next) != start) {
1735                                 /*
1736                                  * We have failed to atomically swing the
1737                                  * hash table head pointer, presumably because
1738                                  * of a conflicting allocation on another CPU.
1739                                  * We need to reread the hash chain and try
1740                                  * again.
1741                                  */
1742                                 goto top;
1743                         }
1744                 }
1745
1746                 dtrace_membar_producer();
1747
1748                 /*
1749                  * Now set the hash value to indicate that it's free.
1750                  */
1751                 ASSERT(hash[bucket].dtdh_chain != dvar);
1752                 dvar->dtdv_hashval = DTRACE_DYNHASH_FREE;
1753
1754                 dtrace_membar_producer();
1755
1756                 /*
1757                  * Set the next pointer to point at the dirty list, and
1758                  * atomically swing the dirty pointer to the newly freed dvar.
1759                  */
1760                 do {
1761                         next = dcpu->dtdsc_dirty;
1762                         dvar->dtdv_next = next;
1763                 } while (dtrace_casptr(&dcpu->dtdsc_dirty, next, dvar) != next);
1764
1765                 /*
1766                  * Finally, unlock this hash bucket.
1767                  */
1768                 ASSERT(hash[bucket].dtdh_lock == lock);
1769                 ASSERT(lock & 1);
1770                 hash[bucket].dtdh_lock++;
1771
1772                 return (NULL);
1773 next:
1774                 prev = dvar;
1775                 continue;
1776         }
1777
1778         if (dvar == NULL) {
1779                 /*
1780                  * If dvar is NULL, it is because we went off the rails:
1781                  * one of the elements that we traversed in the hash chain
1782                  * was deleted while we were traversing it.  In this case,
1783                  * we assert that we aren't doing a dealloc (deallocs lock
1784                  * the hash bucket to prevent themselves from racing with
1785                  * one another), and retry the hash chain traversal.
1786                  */
1787                 ASSERT(op != DTRACE_DYNVAR_DEALLOC);
```

```
1788                    goto top;
1789            }

1791            if (op != DTRACE_DYNVAR_ALLOC) {
1792                    /*
1793                     * If we are not to allocate a new variable, we want to
1794                     * return NULL now.  Before we return, check that the value
1795                     * of the lock word hasn't changed.  If it has, we may have
1796                     * seen an inconsistent snapshot.
1797                     */
1798                    if (op == DTRACE_DYNVAR_NOALLOC) {
1799                            if (hash[bucket].dtdh_lock != lock)
1800                                    goto top;
1801                    } else {
1802                            ASSERT(op == DTRACE_DYNVAR_DEALLOC);
1803                            ASSERT(hash[bucket].dtdh_lock == lock);
1804                            ASSERT(lock & 1);
1805                            hash[bucket].dtdh_lock++;
1806                    }

1808                    return (NULL);
1809            }

1811            /*
1812             * We need to allocate a new dynamic variable.  The size we need is the
1813             * size of dtrace_dynvar plus the size of nkeys dtrace_key_t's plus the
1814             * size of any auxiliary key data (rounded up to 8-byte alignment) plus
1815             * the size of any referred-to data (dsize).  We then round the final
1816             * size up to the chunksize for allocation.
1817             */
1818            for (ksize = 0, i = 0; i < nkeys; i++)
1819                    ksize += P2ROUNDUP(key[i].dttk_size, sizeof (uint64_t));

1821            /*
1822             * This should be pretty much impossible, but could happen if, say,
1823             * strange DIF specified the tuple.  Ideally, this should be an
1824             * assertion and not an error condition -- but that requires that the
1825             * chunksize calculation in dtrace_difo_chunksize() be absolutely
1826             * bullet-proof.  (That is, it must not be able to be fooled by
1827             * malicious DIF.)  Given the lack of backwards branches in DIF,
1828             * solving this would presumably not amount to solving the Halting
1829             * Problem -- but it still seems awfully hard.
1830             */
1831            if (sizeof (dtrace_dynvar_t) + sizeof (dtrace_key_t) * (nkeys - 1) +
1832                ksize + dsize > chunksize) {
1833                    dcpu->dtdsc_drops++;
1834                    return (NULL);
1835            }

1837            nstate = DTRACE_DSTATE_EMPTY;

1839            do {
1840    retry:
1841                    free = dcpu->dtdsc_free;

1843                    if (free == NULL) {
1844                            dtrace_dynvar_t *clean = dcpu->dtdsc_clean;
1845                            void *rval;

1847                            if (clean == NULL) {
1848                                    /*
1849                                     * We're out of dynamic variable space on
1850                                     * this CPU.  Unless we have tried all CPUs,
1851                                     * we'll try to allocate from a different
1852                                     * CPU.
1853                                     */
```



```
1854                                    switch (dstate->dtds_state) {
1855                                    case DTRACE_DSTATE_CLEAN: {
1856                                            void *sp = &dstate->dtds_state;

1858                                            if (++cpu >= NCPU)
1859                                                    cpu = 0;

1861                                            if (dcpu->dtdsc_dirty != NULL &&
1862                                                nstate == DTRACE_DSTATE_EMPTY)
1863                                                    nstate = DTRACE_DSTATE_DIRTY;

1865                                            if (dcpu->dtdsc_rinsing != NULL)
1866                                                    nstate = DTRACE_DSTATE_RINSING;

1868                                            dcpu = &dstate->dtds_percpu[cpu];

1870                                            if (cpu != me)
1871                                                    goto retry;

1873                                            (void) dtrace_cas32(sp,
1874                                                DTRACE_DSTATE_CLEAN, nstate);

1876                                            /*
1877                                             * To increment the correct bean
1878                                             * counter, take another lap.
1879                                             */
1880                                            goto retry;
1881                                    }

1883                                    case DTRACE_DSTATE_DIRTY:
1884                                            dcpu->dtdsc_dirty_drops++;
1885                                            break;

1887                                    case DTRACE_DSTATE_RINSING:
1888                                            dcpu->dtdsc_rinsing_drops++;
1889                                            break;

1891                                    case DTRACE_DSTATE_EMPTY:
1892                                            dcpu->dtdsc_drops++;
1893                                            break;
1894                                    }

1896                                    DTRACE_CPUFLAG_SET(CPU_DTRACE_DROP);
1897                                    return (NULL);
1898                            }

1900                            /*
1901                             * The clean list appears to be non-empty.  We want to
1902                             * move the clean list to the free list; we start by
1903                             * moving the clean pointer aside.
1904                             */
1905                            if (dtrace_casptr(&dcpu->dtdsc_clean,
1906                                clean, NULL) != clean) {
1907                                    /*
1908                                     * We are in one of two situations:
1909                                     *
1910                                     *  (a) The clean list was switched to the
1911                                     *      free list by another CPU.
1912                                     *
1913                                     *  (b) The clean list was added to by the
1914                                     *      cleansing cyclic.
1915                                     *
1916                                     * In either of these situations, we can
1917                                     * just reattempt the free list allocation.
1918                                     */
1919                                    goto retry;
```

```
1920                        }

1922                        ASSERT(clean->dtdv_hashval == DTRACE_DYNHASH_FREE);

1924                        /*
1925                         * Now we'll move the clean list to our free list.
1926                         * It's impossible for this to fail:  the only way
1927                         * the free list can be updated is through this
1928                         * code path, and only one CPU can own the clean list.
1929                         * Thus, it would only be possible for this to fail if
1930                         * this code were racing with dtrace_dynvar_clean().
1931                         * (That is, if dtrace_dynvar_clean() updated the clean
1932                         * list, and we ended up racing to update the free
1933                         * list.)  This race is prevented by the dtrace_sync()
1934                         * in dtrace_dynvar_clean() -- which flushes the
1935                         * owners of the clean lists out before resetting
1936                         * the clean lists.
1937                         */
1938                        dcpu = &dstate->dtds_percpu[me];
1939                        rval = dtrace_casptr(&dcpu->dtdsc_free, NULL, clean);
1940                        ASSERT(rval == NULL);
1941                        goto retry;
1942                }

1944                dvar = free;
1945                new_free = dvar->dtdv_next;
1946        } while (dtrace_casptr(&dcpu->dtdsc_free, free, new_free) != free);

1948        /*
1949         * We have now allocated a new chunk.  We copy the tuple keys into the
1950         * tuple array and copy any referenced key data into the data space
1951         * following the tuple array.  As we do this, we relocate dttk_value
1952         * in the final tuple to point to the key data address in the chunk.
1953         */
1954        kdata = (uintptr_t)&dvar->dtdv_tuple.dtt_key[nkeys];
1955        dvar->dtdv_data = (void *)(kdata + ksize);
1956        dvar->dtdv_tuple.dtt_nkeys = nkeys;

1958        for (i = 0; i < nkeys; i++) {
1959                dtrace_key_t *dkey = &dvar->dtdv_tuple.dtt_key[i];
1960                size_t kesize = key[i].dttk_size;

1962                if (kesize != 0) {
1963                        dtrace_bcopy(
1964                            (const void *)(uintptr_t)key[i].dttk_value,
1965                            (void *)kdata, kesize);
1966                        dkey->dttk_value = kdata;
1967                        kdata += P2ROUNDUP(kesize, sizeof (uint64_t));
1968                } else {
1969                        dkey->dttk_value = key[i].dttk_value;
1970                }

1972                dkey->dttk_size = kesize;
1973        }

1975        ASSERT(dvar->dtdv_hashval == DTRACE_DYNHASH_FREE);
1976        dvar->dtdv_hashval = hashval;
1977        dvar->dtdv_next = start;

1979        if (dtrace_casptr(&hash[bucket].dtdh_chain, start, dvar) == start)
1980                return (dvar);

1982        /*
1983         * The cas has failed.  Either another CPU is adding an element to
1984         * this hash chain, or another CPU is deleting an element from this
1985         * hash chain.  The simplest way to deal with both of these cases
```



```
1986         * (though not necessarily the most efficient) is to free our
1987         * allocated block and tail-call ourselves.  Note that the free is
1988         * to the dirty list and _not_ to the free list.  This is to prevent
1989         * races with allocators, above.
1990         */
1991        dvar->dtdv_hashval = DTRACE_DYNHASH_FREE;

1993        dtrace_membar_producer();

1995        do {
1996                free = dcpu->dtdsc_dirty;
1997                dvar->dtdv_next = free;
1998        } while (dtrace_casptr(&dcpu->dtdsc_dirty, free, dvar) != free);

2000        return (dtrace_dynvar(dstate, nkeys, key, dsize, op, mstate, vstate));
2001 }

2003 /*ARGSUSED*/
2004 static void
2005 dtrace_aggregate_min(uint64_t *oval, uint64_t nval, uint64_t arg)
2006 {
2007        if ((int64_t)nval < (int64_t)*oval)
2008                *oval = nval;
2009 }

2011 /*ARGSUSED*/
2012 static void
2013 dtrace_aggregate_max(uint64_t *oval, uint64_t nval, uint64_t arg)
2014 {
2015        if ((int64_t)nval > (int64_t)*oval)
2016                *oval = nval;
2017 }

2019 static void
2020 dtrace_aggregate_quantize(uint64_t *quanta, uint64_t nval, uint64_t incr)
2021 {
2022        int i, zero = DTRACE_QUANTIZE_ZEROBUCKET;
2023        int64_t val = (int64_t)nval;

2025        if (val < 0) {
2026                for (i = 0; i < zero; i++) {
2027                        if (val <= DTRACE_QUANTIZE_BUCKETVAL(i)) {
2028                                quanta[i] += incr;
2029                                return;
2030                        }
2031                }
2032        } else {
2033                for (i = zero + 1; i < DTRACE_QUANTIZE_NBUCKETS; i++) {
2034                        if (val < DTRACE_QUANTIZE_BUCKETVAL(i)) {
2035                                quanta[i - 1] += incr;
2036                                return;
2037                        }
2038                }

2040                quanta[DTRACE_QUANTIZE_NBUCKETS - 1] += incr;
2041                return;
2042        }

2044        ASSERT(0);
2045 }

2047 static void
2048 dtrace_aggregate_lquantize(uint64_t *lquanta, uint64_t nval, uint64_t incr)
2049 {
2050        uint64_t arg = *lquanta++;
2051        int32_t base = DTRACE_LQUANTIZE_BASE(arg);
```

```
2052         uint16_t step = DTRACE_LQUANTIZE_STEP(arg);
2053         uint16_t levels = DTRACE_LQUANTIZE_LEVELS(arg);
2054         int32_t val = (int32_t)nval, level;

2056         ASSERT(step != 0);
2057         ASSERT(levels != 0);

2059         if (val < base) {
2060                 /*
2061                  * This is an underflow.
2062                  */
2063                 lquanta[0] += incr;
2064                 return;
2065         }

2067         level = (val - base) / step;

2069         if (level < levels) {
2070                 lquanta[level + 1] += incr;
2071                 return;
2072         }

2074         /*
2075          * This is an overflow.
2076          */
2077         lquanta[levels + 1] += incr;
2078 }

2080 static int
2081 dtrace_aggregate_llquantize_bucket(uint16_t factor, uint16_t low,
2082     uint16_t high, uint16_t nsteps, int64_t value)
2083 {
2084         int64_t this = 1, last, next;
2085         int base = 1, order;

2087         ASSERT(factor <= nsteps);
2088         ASSERT(nsteps % factor == 0);

2090         for (order = 0; order < low; order++)
2091                 this *= factor;

2093         /*
2094          * If our value is less than our factor taken to the power of the
2095          * low order of magnitude, it goes into the zeroth bucket.
2096          */
2097         if (value < (last = this))
2098                 return (0);

2100         for (this *= factor; order <= high; order++) {
2101                 int nbuckets = this > nsteps ? nsteps : this;

2103                 if ((next = this * factor) < this) {
2104                         /*
2105                          * We should not generally get log/linear quantizations
2106                          * with a high magnitude that allows 64-bits to
2107                          * overflow, but we nonetheless protect against this
2108                          * by explicitly checking for overflow, and clamping
2109                          * our value accordingly.
2110                          */
2111                         value = this - 1;
2112                 }

2114                 if (value < this) {
2115                         /*
2116                          * If our value lies within this order of magnitude,
2117                          * determine its position by taking the offset within
```



```
2118                          * the order of magnitude, dividing by the bucket
2119                          * width, and adding to our (accumulated) base.
2120                          */
2121                         return (base + (value - last) / (this / nbuckets));
2122                 }

2124                 base += nbuckets - (nbuckets / factor);
2125                 last = this;
2126                 this = next;
2127         }

2129         /*
2130          * Our value is greater than or equal to our factor taken to the
2131          * power of one plus the high magnitude -- return the top bucket.
2132          */
2133         return (base);
2134 }

2136 static void
2137 dtrace_aggregate_llquantize(uint64_t *llquanta, uint64_t nval, uint64_t incr)
2138 {
2139         uint64_t arg = *llquanta++;
2140         uint16_t factor = DTRACE_LLQUANTIZE_FACTOR(arg);
2141         uint16_t low = DTRACE_LLQUANTIZE_LOW(arg);
2142         uint16_t high = DTRACE_LLQUANTIZE_HIGH(arg);
2143         uint16_t nsteps = DTRACE_LLQUANTIZE_NSTEP(arg);

2145         llquanta[dtrace_aggregate_llquantize_bucket(factor,
2146             low, high, nsteps, nval)] += incr;
2147 }

2149 /*ARGSUSED*/
2150 static void
2151 dtrace_aggregate_avg(uint64_t *data, uint64_t nval, uint64_t arg)
2152 {
2153         data[0]++;
2154         data[1] += nval;
2155 }

2157 /*ARGSUSED*/
2158 static void
2159 dtrace_aggregate_stddev(uint64_t *data, uint64_t nval, uint64_t arg)
2160 {
2161         int64_t snval = (int64_t)nval;
2162         uint64_t tmp[2];

2164         data[0]++;
2165         data[1] += nval;

2167         /*
2168          * What we want to say here is:
2169          *
2170          * data[2] += nval * nval;
2171          *
2172          * But given that nval is 64-bit, we could easily overflow, so
2173          * we do this as 128-bit arithmetic.
2174          */
2175         if (snval < 0)
2176                 snval = -snval;

2178         dtrace_multiply_128((uint64_t)snval, (uint64_t)snval, tmp);
2179         dtrace_add_128(data + 2, tmp, data + 2);
2180 }

2182 /*ARGSUSED*/
2183 static void
```

```
dtrace_aggregate_count(uint64_t *oval, uint64_t nval, uint64_t arg)
{
        *oval = *oval + 1;
}

/*ARGSUSED*/
static void
dtrace_aggregate_sum(uint64_t *oval, uint64_t nval, uint64_t arg)
{
        *oval += nval;
}

/*
 * Aggregate given the tuple in the principal data buffer, and the aggregating
 * action denoted by the specified dtrace_aggregation_t.  The aggregation
 * buffer is specified as the buf parameter.  This routine does not return
 * failure; if there is no space in the aggregation buffer, the data will be
 * dropped, and a corresponding counter incremented.
 */
static void
dtrace_aggregate(dtrace_aggregation_t *agg, dtrace_buffer_t *dbuf,
    intptr_t offset, dtrace_buffer_t *buf, uint64_t expr, uint64_t arg)
{
        dtrace_recdesc_t *rec = &agg->dtag_action.dta_rec;
        uint32_t i, ndx, size, fsize;
        uint32_t align = sizeof (uint64_t) - 1;
        dtrace_aggbuffer_t *agb;
        dtrace_aggkey_t *key;
        uint32_t hashval = 0, limit, isstr;
        caddr_t tomax, data, kdata;
        dtrace_actkind_t action;
        dtrace_action_t *act;
        uintptr_t offs;

        if (buf == NULL)
                return;

        if (!agg->dtag_hasarg) {
                /*
                 * Currently, only quantize() and lquantize() take additional
                 * arguments, and they have the same semantics:  an increment
                 * value that defaults to 1 when not present.  If additional
                 * aggregating actions take arguments, the setting of the
                 * default argument value will presumably have to become more
                 * sophisticated...
                 */
                arg = 1;
        }

        action = agg->dtag_action.dta_kind - DTRACEACT_AGGREGATION;
        size = rec->dtrd_offset - agg->dtag_base;
        fsize = size + rec->dtrd_size;

        ASSERT(dbuf->dtb_tomax != NULL);
        data = dbuf->dtb_tomax + offset + agg->dtag_base;

        if ((tomax = buf->dtb_tomax) == NULL) {
                dtrace_buffer_drop(buf);
                return;
        }

        /*
         * The metastructure is always at the bottom of the buffer.
         */
        agb = (dtrace_aggbuffer_t *)(tomax + buf->dtb_size -
            sizeof (dtrace_aggbuffer_t));
```



```
        if (buf->dtb_offset == 0) {
                /*
                 * We just kludge up approximately 1/8th of the size to be
                 * buckets.  If this guess ends up being routinely
                 * off-the-mark, we may need to dynamically readjust this
                 * based on past performance.
                 */
                uintptr_t hashsize = (buf->dtb_size >> 3) / sizeof (uintptr_t);

                if ((uintptr_t)agb - hashsize * sizeof (dtrace_aggkey_t *) <
                    (uintptr_t)tomax || hashsize == 0) {
                        /*
                         * We've been given a ludicrously small buffer;
                         * increment our drop count and leave.
                         */
                        dtrace_buffer_drop(buf);
                        return;
                }

                /*
                 * And now, a pathetic attempt to try to get a an odd (or
                 * perchance, a prime) hash size for better hash distribution.
                 */
                if (hashsize > (DTRACE_AGGHASHSIZE_SLEW << 3))
                        hashsize -= DTRACE_AGGHASHSIZE_SLEW;

                agb->dtagb_hashsize = hashsize;
                agb->dtagb_hash = (dtrace_aggkey_t **)((uintptr_t)agb -
                    agb->dtagb_hashsize * sizeof (dtrace_aggkey_t *));
                agb->dtagb_free = (uintptr_t)agb->dtagb_hash;

                for (i = 0; i < agb->dtagb_hashsize; i++)
                        agb->dtagb_hash[i] = NULL;
        }

        ASSERT(agg->dtag_first != NULL);
        ASSERT(agg->dtag_first->dta_intuple);

        /*
         * Calculate the hash value based on the key.  Note that we _don't_
         * include the aggid in the hashing (but we will store it as part of
         * the key).  The hashing algorithm is Bob Jenkins' "One-at-a-time"
         * algorithm: a simple, quick algorithm that has no known funnels, and
         * gets good distribution in practice.  The efficacy of the hashing
         * algorithm (and a comparison with other algorithms) may be found by
         * running the ::dtrace_aggstat MDB dcmd.
         */
        for (act = agg->dtag_first; act->dta_intuple; act = act->dta_next) {
                i = act->dta_rec.dtrd_offset - agg->dtag_base;
                limit = i + act->dta_rec.dtrd_size;
                ASSERT(limit <= size);
                isstr = DTRACEACT_ISSTRING(act);

                for (; i < limit; i++) {
                        hashval += data[i];
                        hashval += (hashval << 10);
                        hashval ^= (hashval >> 6);

                        if (isstr && data[i] == '\0')
                                break;
                }
        }

        hashval += (hashval << 3);
        hashval ^= (hashval >> 11);
```

```
2316         hashval += (hashval << 15);
2317
2318         /*
2319          * Yes, the divide here is expensive -- but it's generally the least
2320          * of the performance issues given the amount of data that we iterate
2321          * over to compute hash values, compare data, etc.
2322          */
2323         ndx = hashval % agb->dtagb_hashsize;
2324
2325         for (key = agb->dtagb_hash[ndx]; key != NULL; key = key->dtak_next) {
2326                 ASSERT((caddr_t)key >= tomax);
2327                 ASSERT((caddr_t)key < tomax + buf->dtb_size);
2328
2329                 if (hashval != key->dtak_hashval || key->dtak_size != size)
2330                         continue;
2331
2332                 kdata = key->dtak_data;
2333                 ASSERT(kdata >= tomax && kdata < tomax + buf->dtb_size);
2334
2335                 for (act = agg->dtag_first; act->dta_intuple;
2336                     act = act->dta_next) {
2337                         i = act->dta_rec.dtrd_offset - agg->dtag_base;
2338                         limit = i + act->dta_rec.dtrd_size;
2339                         ASSERT(limit <= size);
2340                         isstr = DTRACEACT_ISSTRING(act);
2341
2342                         for (; i < limit; i++) {
2343                                 if (kdata[i] != data[i])
2344                                         goto next;
2345
2346                                 if (isstr && data[i] == '\0')
2347                                         break;
2348                         }
2349                 }
2350
2351                 if (action != key->dtak_action) {
2352                         /*
2353                          * We are aggregating on the same value in the same
2354                          * aggregation with two different aggregating actions.
2355                          * (This should have been picked up in the compiler,
2356                          * so we may be dealing with errant or devious DIF.)
2357                          * This is an error condition; we indicate as much,
2358                          * and return.
2359                          */
2360                         DTRACE_CPUFLAG_SET(CPU_DTRACE_ILLOP);
2361                         return;
2362                 }
2363
2364                 /*
2365                  * This is a hit:  we need to apply the aggregator to
2366                  * the value at this key.
2367                  */
2368                 agg->dtag_aggregate((uint64_t *)(kdata + size), expr, arg);
2369                 return;
2370 next:
2371                 continue;
2372         }
2373
2374         /*
2375          * We didn't find it.  We need to allocate some zero-filled space,
2376          * link it into the hash table appropriately, and apply the aggregator
2377          * to the (zero-filled) value.
2378          */
2379         offs = buf->dtb_offset;
2380         while (offs & (align - 1))
2381                 offs += sizeof (uint32_t);
```



```
2383         /*
2384          * If we don't have enough room to both allocate a new key _and_
2385          * its associated data, increment the drop count and return.
2386          */
2387         if ((uintptr_t)tomax + offs + fsize >
2388             agb->dtagb_free - sizeof (dtrace_aggkey_t)) {
2389                 dtrace_buffer_drop(buf);
2390                 return;
2391         }
2392
2393         /*CONSTCOND*/
2394         ASSERT(!(sizeof (dtrace_aggkey_t) & (sizeof (uintptr_t) - 1)));
2395         key = (dtrace_aggkey_t *)(agb->dtagb_free - sizeof (dtrace_aggkey_t));
2396         agb->dtagb_free -= sizeof (dtrace_aggkey_t);
2397
2398         key->dtak_data = kdata = tomax + offs;
2399         buf->dtb_offset = offs + fsize;
2400
2401         /*
2402          * Now copy the data across.
2403          */
2404         *((dtrace_aggid_t *)kdata) = agg->dtag_id;
2405
2406         for (i = sizeof (dtrace_aggid_t); i < size; i++)
2407                 kdata[i] = data[i];
2408
2409         /*
2410          * Because strings are not zeroed out by default, we need to iterate
2411          * looking for actions that store strings, and we need to explicitly
2412          * pad these strings out with zeroes.
2413          */
2414         for (act = agg->dtag_first; act->dta_intuple; act = act->dta_next) {
2415                 int nul;
2416
2417                 if (!DTRACEACT_ISSTRING(act))
2418                         continue;
2419
2420                 i = act->dta_rec.dtrd_offset - agg->dtag_base;
2421                 limit = i + act->dta_rec.dtrd_size;
2422                 ASSERT(limit <= size);
2423
2424                 for (nul = 0; i < limit; i++) {
2425                         if (nul) {
2426                                 kdata[i] = '\0';
2427                                 continue;
2428                         }
2429
2430                         if (data[i] != '\0')
2431                                 continue;
2432
2433                         nul = 1;
2434                 }
2435         }
2436
2437         for (i = size; i < fsize; i++)
2438                 kdata[i] = 0;
2439
2440         key->dtak_hashval = hashval;
2441         key->dtak_size = size;
2442         key->dtak_action = action;
2443         key->dtak_next = agb->dtagb_hash[ndx];
2444         agb->dtagb_hash[ndx] = key;
2445
2446         /*
2447          * Finally, apply the aggregator.
```

```
2448          */
2449         *((uint64_t *)(key->dtak_data + size)) = agg->dtag_initial;
2450         agg->dtag_aggregate((uint64_t *)(key->dtak_data + size), expr, arg);
2451 }

2453 /*
2454  * Given consumer state, this routine finds a speculation in the INACTIVE
2455  * state and transitions it into the ACTIVE state.  If there is no speculation
2456  * in the INACTIVE state, 0 is returned.  In this case, no error counter is
2457  * incremented -- it is up to the caller to take appropriate action.
2458  */
2459 static int
2460 dtrace_speculation(dtrace_state_t *state)
2461 {
2462         int i = 0;
2463         dtrace_speculation_state_t current;
2464         uint32_t *stat = &state->dts_speculations_unavail, count;

2466         while (i < state->dts_nspeculations) {
2467                 dtrace_speculation_t *spec = &state->dts_speculations[i];

2469                 current = spec->dtsp_state;

2471                 if (current != DTRACESPEC_INACTIVE) {
2472                         if (current == DTRACESPEC_COMMITTINGMANY ||
2473                             current == DTRACESPEC_COMMITTING ||
2474                             current == DTRACESPEC_DISCARDING)
2475                                 stat = &state->dts_speculations_busy;
2476                         i++;
2477                         continue;
2478                 }

2480                 if (dtrace_cas32((uint32_t *)&spec->dtsp_state,
2481                     current, DTRACESPEC_ACTIVE) == current)
2482                         return (i + 1);
2483         }

2485         /*
2486          * We couldn't find a speculation.  If we found as much as a single
2487          * busy speculation buffer, we'll attribute this failure as "busy"
2488          * instead of "unavail".
2489          */
2490         do {
2491                 count = *stat;
2492         } while (dtrace_cas32(stat, count, count + 1) != count);

2494         return (0);
2495 }

2497 /*
2498  * This routine commits an active speculation.  If the specified speculation
2499  * is not in a valid state to perform a commit(), this routine will silently do
2500  * nothing.  The state of the specified speculation is transitioned according
2501  * to the state transition diagram outlined in <sys/dtrace_impl.h>
2502  */
2503 static void
2504 dtrace_speculation_commit(dtrace_state_t *state, processorid_t cpu,
2505     dtrace_specid_t which)
2506 {
2507         dtrace_speculation_t *spec;
2508         dtrace_buffer_t *src, *dest;
2509         uintptr_t daddr, saddr, dlimit, slimit;
2510         dtrace_speculation_state_t current, new;
2511         intptr_t offs;
2512         uint64_t timestamp;
```



```
2514         if (which == 0)
2515                 return;

2517         if (which > state->dts_nspeculations) {
2518                 cpu_core[cpu].cpuc_dtrace_flags |= CPU_DTRACE_ILLOP;
2519                 return;
2520         }

2522         spec = &state->dts_speculations[which - 1];
2523         src = &spec->dtsp_buffer[cpu];
2524         dest = &state->dts_buffer[cpu];

2526         do {
2527                 current = spec->dtsp_state;

2529                 if (current == DTRACESPEC_COMMITTINGMANY)
2530                         break;

2532                 switch (current) {
2533                 case DTRACESPEC_INACTIVE:
2534                 case DTRACESPEC_DISCARDING:
2535                         return;

2537                 case DTRACESPEC_COMMITTING:
2538                         /*
2539                          * This is only possible if we are (a) commit()'ing
2540                          * without having done a prior speculate() on this CPU
2541                          * and (b) racing with another commit() on a different
2542                          * CPU.  There's nothing to do -- we just assert that
2543                          * our offset is 0.
2544                          */
2545                         ASSERT(src->dtb_offset == 0);
2546                         return;

2548                 case DTRACESPEC_ACTIVE:
2549                         new = DTRACESPEC_COMMITTING;
2550                         break;

2552                 case DTRACESPEC_ACTIVEONE:
2553                         /*
2554                          * This speculation is active on one CPU.  If our
2555                          * buffer offset is non-zero, we know that the one CPU
2556                          * must be us.  Otherwise, we are committing on a
2557                          * different CPU from the speculate(), and we must
2558                          * rely on being asynchronously cleaned.
2559                          */
2560                         if (src->dtb_offset != 0) {
2561                                 new = DTRACESPEC_COMMITTING;
2562                                 break;
2563                         }
2564                         /*FALLTHROUGH*/

2566                 case DTRACESPEC_ACTIVEMANY:
2567                         new = DTRACESPEC_COMMITTINGMANY;
2568                         break;

2570                 default:
2571                         ASSERT(0);
2572                 }
2573         } while (dtrace_cas32((uint32_t *)&spec->dtsp_state,
2574             current, new) != current);

2576         /*
2577          * We have set the state to indicate that we are committing this
2578          * speculation.  Now reserve the necessary space in the destination
2579          * buffer.
```

```
2580          */
2581         if ((offs = dtrace_buffer_reserve(dest, src->dtb_offset,
2582             sizeof (uint64_t), state, NULL)) < 0) {
2583                 dtrace_buffer_drop(dest);
2584                 goto out;
2585         }
2586
2587         /*
2588          * We have sufficient space to copy the speculative buffer into the
2589          * primary buffer.  First, modify the speculative buffer, filling
2590          * in the timestamp of all entries with the current time.  The data
2591          * must have the commit() time rather than the time it was traced,
2592          * so that all entries in the primary buffer are in timestamp order.
2593          */
2594         timestamp = dtrace_gethrtime();
2595         saddr = (uintptr_t)src->dtb_tomax;
2596         slimit = saddr + src->dtb_offset;
2597         while (saddr < slimit) {
2598                 size_t size;
2599                 dtrace_rechdr_t *dtrh = (dtrace_rechdr_t *)saddr;
2600
2601                 if (dtrh->dtrh_epid == DTRACE_EPIDNONE) {
2602                         saddr += sizeof (dtrace_epid_t);
2603                         continue;
2604                 }
2605                 ASSERT3U(dtrh->dtrh_epid, <=, state->dts_necbs);
2606                 size = state->dts_ecbs[dtrh->dtrh_epid - 1]->dte_size;
2607
2608                 ASSERT3U(saddr + size, <=, slimit);
2609                 ASSERT3U(size, >=, sizeof (dtrace_rechdr_t));
2610                 ASSERT3U(DTRACE_RECORD_LOAD_TIMESTAMP(dtrh), ==, UINT64_MAX);
2611
2612                 DTRACE_RECORD_STORE_TIMESTAMP(dtrh, timestamp);
2613
2614                 saddr += size;
2615         }
2616
2617         /*
2618          * Copy the buffer across.  (Note that this is a
2619          * highly subobtimal bcopy(); in the unlikely event that this becomes
2620          * a serious performance issue, a high-performance DTrace-specific
2621          * bcopy() should obviously be invented.)
2622          */
2623         daddr = (uintptr_t)dest->dtb_tomax + offs;
2624         dlimit = daddr + src->dtb_offset;
2625         saddr = (uintptr_t)src->dtb_tomax;
2626
2627         /*
2628          * First, the aligned portion.
2629          */
2630         while (dlimit - daddr >= sizeof (uint64_t)) {
2631                 *((uint64_t *)daddr) = *((uint64_t *)saddr);
2632
2633                 daddr += sizeof (uint64_t);
2634                 saddr += sizeof (uint64_t);
2635         }
2636
2637         /*
2638          * Now any left-over bit...
2639          */
2640         while (dlimit - daddr)
2641                 *((uint8_t *)daddr++) = *((uint8_t *)saddr++);
2642
2643         /*
2644          * Finally, commit the reserved space in the destination buffer.
2645          */
```



```
2646         dest->dtb_offset = offs + src->dtb_offset;
2647
2648 out:
2649         /*
2650          * If we're lucky enough to be the only active CPU on this speculation
2651          * buffer, we can just set the state back to DTRACESPEC_INACTIVE.
2652          */
2653         if (current == DTRACESPEC_ACTIVE ||
2654             (current == DTRACESPEC_ACTIVEONE && new == DTRACESPEC_COMMITTING)) {
2655                 uint32_t rval = dtrace_cas32((uint32_t *)&spec->dtsp_state,
2656                     DTRACESPEC_COMMITTING, DTRACESPEC_INACTIVE);
2657
2658                 ASSERT(rval == DTRACESPEC_COMMITTING);
2659         }
2660
2661         src->dtb_offset = 0;
2662         src->dtb_xamot_drops += src->dtb_drops;
2663         src->dtb_drops = 0;
2664 }
2665
2666 /*
2667  * This routine discards an active speculation.  If the specified speculation
2668  * is not in a valid state to perform a discard(), this routine will silently
2669  * do nothing.  The state of the specified speculation is transitioned
2670  * according to the state transition diagram outlined in <sys/dtrace_impl.h>
2671  */
2672 static void
2673 dtrace_speculation_discard(dtrace_state_t *state, processorid_t cpu,
2674     dtrace_specid_t which)
2675 {
2676         dtrace_speculation_t *spec;
2677         dtrace_speculation_state_t current, new;
2678         dtrace_buffer_t *buf;
2679
2680         if (which == 0)
2681                 return;
2682
2683         if (which > state->dts_nspeculations) {
2684                 cpu_core[cpu].cpuc_dtrace_flags |= CPU_DTRACE_ILLOP;
2685                 return;
2686         }
2687
2688         spec = &state->dts_speculations[which - 1];
2689         buf = &spec->dtsp_buffer[cpu];
2690
2691         do {
2692                 current = spec->dtsp_state;
2693
2694                 switch (current) {
2695                 case DTRACESPEC_INACTIVE:
2696                 case DTRACESPEC_COMMITTINGMANY:
2697                 case DTRACESPEC_COMMITTING:
2698                 case DTRACESPEC_DISCARDING:
2699                         return;
2700
2701                 case DTRACESPEC_ACTIVE:
2702                 case DTRACESPEC_ACTIVEMANY:
2703                         new = DTRACESPEC_DISCARDING;
2704                         break;
2705
2706                 case DTRACESPEC_ACTIVEONE:
2707                         if (buf->dtb_offset != 0) {
2708                                 new = DTRACESPEC_INACTIVE;
2709                         } else {
2710                                 new = DTRACESPEC_DISCARDING;
2711                         }
```

```
2712                        break;

2714                default:
2715                        ASSERT(0);
2716                }
2717        } while (dtrace_cas32((uint32_t *)&spec->dtsp_state,
2718            current, new) != current);

2720        buf->dtb_offset = 0;
2721        buf->dtb_drops = 0;
2722 }

2724 /*
2725  * Note:  not called from probe context.  This function is called
2726  * asynchronously from cross call context to clean any speculations that are
2727  * in the COMMITTINGMANY or DISCARDING states.  These speculations may not be
2728  * transitioned back to the INACTIVE state until all CPUs have cleaned the
2729  * speculation.
2730  */
2731 static void
2732 dtrace_speculation_clean_here(dtrace_state_t *state)
2733 {
2734        dtrace_icookie_t cookie;
2735        processorid_t cpu = CPU->cpu_id;
2736        dtrace_buffer_t *dest = &state->dts_buffer[cpu];
2737        dtrace_specid_t i;

2739        cookie = dtrace_interrupt_disable();

2741        if (dest->dtb_tomax == NULL) {
2742                dtrace_interrupt_enable(cookie);
2743                return;
2744        }

2746        for (i = 0; i < state->dts_nspeculations; i++) {
2747                dtrace_speculation_t *spec = &state->dts_speculations[i];
2748                dtrace_buffer_t *src = &spec->dtsp_buffer[cpu];

2750                if (src->dtb_tomax == NULL)
2751                        continue;

2753                if (spec->dtsp_state == DTRACESPEC_DISCARDING) {
2754                        src->dtb_offset = 0;
2755                        continue;
2756                }

2758                if (spec->dtsp_state != DTRACESPEC_COMMITTINGMANY)
2759                        continue;

2761                if (src->dtb_offset == 0)
2762                        continue;

2764                dtrace_speculation_commit(state, cpu, i + 1);
2765        }

2767        dtrace_interrupt_enable(cookie);
2768 }

2770 /*
2771  * Note:  not called from probe context.  This function is called
2772  * asynchronously (and at a regular interval) to clean any speculations that
2773  * are in the COMMITTINGMANY or DISCARDING states.  If it discovers that there
2774  * is work to be done, it cross calls all CPUs to perform that work;
2775  * COMMITMANY and DISCARDING speculations may not be transitioned back to the
2776  * INACTIVE state until they have been cleaned by all CPUs.
2777  */
```



```
2778 static void
2779 dtrace_speculation_clean(dtrace_state_t *state)
2780 {
2781        int work = 0, rv;
2782        dtrace_specid_t i;

2784        for (i = 0; i < state->dts_nspeculations; i++) {
2785                dtrace_speculation_t *spec = &state->dts_speculations[i];

2787                ASSERT(!spec->dtsp_cleaning);

2789                if (spec->dtsp_state != DTRACESPEC_DISCARDING &&
2790                    spec->dtsp_state != DTRACESPEC_COMMITTINGMANY)
2791                        continue;

2793                work++;
2794                spec->dtsp_cleaning = 1;
2795        }

2797        if (!work)
2798                return;

2800        dtrace_xcall(DTRACE_CPUALL,
2801            (dtrace_xcall_t)dtrace_speculation_clean_here, state);

2803        /*
2804         * We now know that all CPUs have committed or discarded their
2805         * speculation buffers, as appropriate.  We can now set the state
2806         * to inactive.
2807         */
2808        for (i = 0; i < state->dts_nspeculations; i++) {
2809                dtrace_speculation_t *spec = &state->dts_speculations[i];
2810                dtrace_speculation_state_t current, new;

2812                if (!spec->dtsp_cleaning)
2813                        continue;

2815                current = spec->dtsp_state;
2816                ASSERT(current == DTRACESPEC_DISCARDING ||
2817                    current == DTRACESPEC_COMMITTINGMANY);

2819                new = DTRACESPEC_INACTIVE;

2821                rv = dtrace_cas32((uint32_t *)&spec->dtsp_state, current, new);
2822                ASSERT(rv == current);
2823                spec->dtsp_cleaning = 0;
2824        }
2825 }

2827 /*
2828  * Called as part of a speculate() to get the speculative buffer associated
2829  * with a given speculation.  Returns NULL if the specified speculation is not
2830  * in an ACTIVE state.  If the speculation is in the ACTIVEONE state -- and
2831  * the active CPU is not the specified CPU -- the speculation will be
2832  * atomically transitioned into the ACTIVEMANY state.
2833  */
2834 static dtrace_buffer_t *
2835 dtrace_speculation_buffer(dtrace_state_t *state, processorid_t cpuid,
2836     dtrace_specid_t which)
2837 {
2838        dtrace_speculation_t *spec;
2839        dtrace_speculation_state_t current, new;
2840        dtrace_buffer_t *buf;

2842        if (which == 0)
2843                return (NULL);
```

```
2845         if (which > state->dts_nspeculations) {
2846                 cpu_core[cpuid].cpuc_dtrace_flags |= CPU_DTRACE_ILLOP;
2847                 return (NULL);
2848         }
2849
2850         spec = &state->dts_speculations[which - 1];
2851         buf = &spec->dtsp_buffer[cpuid];
2852
2853         do {
2854                 current = spec->dtsp_state;
2855
2856                 switch (current) {
2857                 case DTRACESPEC_INACTIVE:
2858                 case DTRACESPEC_COMMITTINGMANY:
2859                 case DTRACESPEC_DISCARDING:
2860                         return (NULL);
2861
2862                 case DTRACESPEC_COMMITTING:
2863                         ASSERT(buf->dtb_offset == 0);
2864                         return (NULL);
2865
2866                 case DTRACESPEC_ACTIVEONE:
2867                         /*
2868                          * This speculation is currently active on one CPU.
2869                          * Check the offset in the buffer; if it's non-zero,
2870                          * that CPU must be us (and we leave the state alone).
2871                          * If it's zero, assume that we're starting on a new
2872                          * CPU -- and change the state to indicate that the
2873                          * speculation is active on more than one CPU.
2874                          */
2875                         if (buf->dtb_offset != 0)
2876                                 return (buf);
2877
2878                         new = DTRACESPEC_ACTIVEMANY;
2879                         break;
2880
2881                 case DTRACESPEC_ACTIVEMANY:
2882                         return (buf);
2883
2884                 case DTRACESPEC_ACTIVE:
2885                         new = DTRACESPEC_ACTIVEONE;
2886                         break;
2887
2888                 default:
2889                         ASSERT(0);
2890                 }
2891         } while (dtrace_cas32((uint32_t *)&spec->dtsp_state,
2892             current, new) != current);
2893
2894         ASSERT(new == DTRACESPEC_ACTIVEONE || new == DTRACESPEC_ACTIVEMANY);
2895         return (buf);
2896 }
2897
2898 /*
2899  * Return a string.  In the event that the user lacks the privilege to access
2900  * arbitrary kernel memory, we copy the string out to scratch memory so that we
2901  * don't fail access checking.
2902  *
2903  * dtrace_dif_variable() uses this routine as a helper for various
2904  * builtin values such as 'execname' and 'probefunc.'
2905  */
2906 uintptr_t
2907 dtrace_dif_varstr(uintptr_t addr, dtrace_state_t *state,
2908     dtrace_mstate_t *mstate)
2909 {
```



```
2910         uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
2911         uintptr_t ret;
2912         size_t strsz;
2913
2914         /*
2915          * The easy case: this probe is allowed to read all of memory, so
2916          * we can just return this as a vanilla pointer.
2917          */
2918         if ((mstate->dtms_access & DTRACE_ACCESS_KERNEL) != 0)
2919                 return (addr);
2920
2921         /*
2922          * This is the tougher case: we copy the string in question from
2923          * kernel memory into scratch memory and return it that way: this
2924          * ensures that we won't trip up when access checking tests the
2925          * BYREF return value.
2926          */
2927         strsz = dtrace_strlen((char *)addr, size) + 1;
2928
2929         if (mstate->dtms_scratch_ptr + strsz >
2930             mstate->dtms_scratch_base + mstate->dtms_scratch_size) {
2931                 DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
2932                 return (NULL);
2933         }
2934
2935         dtrace_strcpy((const void *)addr, (void *)mstate->dtms_scratch_ptr,
2936             strsz);
2937         ret = mstate->dtms_scratch_ptr;
2938         mstate->dtms_scratch_ptr += strsz;
2939         return (ret);
2940 }
2941
2942 /*
2943  * This function implements the DIF emulator's variable lookups.  The emulator
2944  * passes a reserved variable identifier and optional built-in array index.
2945  */
2946 static uint64_t
2947 dtrace_dif_variable(dtrace_mstate_t *mstate, dtrace_state_t *state, uint64_t v,
2948     uint64_t ndx)
2949 {
2950         /*
2951          * If we're accessing one of the uncached arguments, we'll turn this
2952          * into a reference in the args array.
2953          */
2954         if (v >= DIF_VAR_ARG0 && v <= DIF_VAR_ARG9) {
2955                 ndx = v - DIF_VAR_ARG0;
2956                 v = DIF_VAR_ARGS;
2957         }
2958
2959         switch (v) {
2960         case DIF_VAR_ARGS:
2961                 if (!(mstate->dtms_access & DTRACE_ACCESS_ARGS)) {
2962                         cpu_core[CPU->cpu_id].cpuc_dtrace_flags |=
2963                             CPU_DTRACE_KPRIV;
2964                         return (0);
2965                 }
2966
2967                 ASSERT(mstate->dtms_present & DTRACE_MSTATE_ARGS);
2968                 if (ndx >= sizeof (mstate->dtms_arg) /
2969                     sizeof (mstate->dtms_arg[0])) {
2970                         int aframes = mstate->dtms_probe->dtpr_aframes + 2;
2971                         dtrace_provider_t *pv;
2972                         uint64_t val;
2973
2974                         pv = mstate->dtms_probe->dtpr_provider;
2975                         if (pv->dtpv_pops.dtps_getargval != NULL)
```

```
2976                                val = pv->dtpv_pops.dtps_getargval(pv->dtpv_arg,
2977                                    mstate->dtms_probe->dtpr_id,
2978                                    mstate->dtms_probe->dtpr_arg, ndx, aframes);
2979                        else
2980                                val = dtrace_getarg(ndx, aframes);

2982                        /*
2983                         * This is regrettably required to keep the compiler
2984                         * from tail-optimizing the call to dtrace_getarg().
2985                         * The condition always evaluates to true, but the
2986                         * compiler has no way of figuring that out a priori.
2987                         * (None of this would be necessary if the compiler
2988                         * could be relied upon to _always_ tail-optimize
2989                         * the call to dtrace_getarg() -- but it can't.)
2990                         */
2991                        if (mstate->dtms_probe != NULL)
2992                                return (val);

2994                        ASSERT(0);
2995                }

2997                return (mstate->dtms_arg[ndx]);

2999        case DIF_VAR_UREGS: {
3000                klwp_t *lwp;

3002                if (!dtrace_priv_proc(state, mstate))
3003                        return (0);

3005                if ((lwp = curthread->t_lwp) == NULL) {
3006                        DTRACE_CPUFLAG_SET(CPU_DTRACE_BADADDR);
3007                        cpu_core[CPU->cpu_id].cpuc_dtrace_illval = NULL;
3008                        return (0);
3009                }

3011                return (dtrace_getreg(lwp->lwp_regs, ndx));
3012        }

3014        case DIF_VAR_VMREGS: {
3015                uint64_t rval;

3017                if (!dtrace_priv_kernel(state))
3018                        return (0);

3020                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);

3022                rval = dtrace_getvmreg(ndx,
3023                    &cpu_core[CPU->cpu_id].cpuc_dtrace_flags);

3025                DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);

3027                return (rval);
3028        }

3030        case DIF_VAR_CURTHREAD:
3031                if (!dtrace_priv_proc(state, mstate))
 550                if (!dtrace_priv_kernel(state))
3032                        return (0);
3033                return ((uint64_t)(uintptr_t)curthread);

3035        case DIF_VAR_TIMESTAMP:
3036                if (!(mstate->dtms_present & DTRACE_MSTATE_TIMESTAMP)) {
3037                        mstate->dtms_timestamp = dtrace_gethrtime();
3038                        mstate->dtms_present |= DTRACE_MSTATE_TIMESTAMP;
3039                }
3040                return (mstate->dtms_timestamp);
```



```
3042        case DIF_VAR_VTIMESTAMP:
3043                ASSERT(dtrace_vtime_references != 0);
3044                return (curthread->t_dtrace_vtime);

3046        case DIF_VAR_WALLTIMESTAMP:
3047                if (!(mstate->dtms_present & DTRACE_MSTATE_WALLTIMESTAMP)) {
3048                        mstate->dtms_walltimestamp = dtrace_gethrestime();
3049                        mstate->dtms_present |= DTRACE_MSTATE_WALLTIMESTAMP;
3050                }
3051                return (mstate->dtms_walltimestamp);

3053        case DIF_VAR_IPL:
3054                if (!dtrace_priv_kernel(state))
3055                        return (0);
3056                if (!(mstate->dtms_present & DTRACE_MSTATE_IPL)) {
3057                        mstate->dtms_ipl = dtrace_getipl();
3058                        mstate->dtms_present |= DTRACE_MSTATE_IPL;
3059                }
3060                return (mstate->dtms_ipl);

3062        case DIF_VAR_EPID:
3063                ASSERT(mstate->dtms_present & DTRACE_MSTATE_EPID);
3064                return (mstate->dtms_epid);

3066        case DIF_VAR_ID:
3067                ASSERT(mstate->dtms_present & DTRACE_MSTATE_PROBE);
3068                return (mstate->dtms_probe->dtpr_id);

3070        case DIF_VAR_STACKDEPTH:
3071                if (!dtrace_priv_kernel(state))
3072                        return (0);
3073                if (!(mstate->dtms_present & DTRACE_MSTATE_STACKDEPTH)) {
3074                        int aframes = mstate->dtms_probe->dtpr_aframes + 2;

3076                        mstate->dtms_stackdepth = dtrace_getstackdepth(aframes);
3077                        mstate->dtms_present |= DTRACE_MSTATE_STACKDEPTH;
3078                }
3079                return (mstate->dtms_stackdepth);

3081        case DIF_VAR_USTACKDEPTH:
3082                if (!dtrace_priv_proc(state, mstate))
3083                        return (0);
3084                if (!(mstate->dtms_present & DTRACE_MSTATE_USTACKDEPTH)) {
3085                        /*
3086                         * See comment in DIF_VAR_PID.
3087                         */
3088                        if (DTRACE_ANCHORED(mstate->dtms_probe) &&
3089                            CPU_ON_INTR(CPU)) {
3090                                mstate->dtms_ustackdepth = 0;
3091                        } else {
3092                                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
3093                                mstate->dtms_ustackdepth =
3094                                    dtrace_getustackdepth();
3095                                DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
3096                        }
3097                        mstate->dtms_present |= DTRACE_MSTATE_USTACKDEPTH;
3098                }
3099                return (mstate->dtms_ustackdepth);

3101        case DIF_VAR_CALLER:
3102                if (!dtrace_priv_kernel(state))
3103                        return (0);
3104                if (!(mstate->dtms_present & DTRACE_MSTATE_CALLER)) {
3105                        int aframes = mstate->dtms_probe->dtpr_aframes + 2;
```

```
3107                        if (!DTRACE_ANCHORED(mstate->dtms_probe)) {
3108                                /*
3109                                 * If this is an unanchored probe, we are
3110                                 * required to go through the slow path:
3111                                 * dtrace_caller() only guarantees correct
3112                                 * results for anchored probes.
3113                                 */
3114                                pc_t caller[2];

3116                                dtrace_getpcstack(caller, 2, aframes,
3117                                    (uint32_t *)(uintptr_t)mstate->dtms_arg[0]);
3118                                mstate->dtms_caller = caller[1];
3119                        } else if ((mstate->dtms_caller =
3120                            dtrace_caller(aframes)) == -1) {
3121                                /*
3122                                 * We have failed to do this the quick way;
3123                                 * we must resort to the slower approach of
3124                                 * calling dtrace_getpcstack().
3125                                 */
3126                                pc_t caller;

3128                                dtrace_getpcstack(&caller, 1, aframes, NULL);
3129                                mstate->dtms_caller = caller;
3130                        }

3132                        mstate->dtms_present |= DTRACE_MSTATE_CALLER;
3133                }
3134                return (mstate->dtms_caller);

3136        case DIF_VAR_UCALLER:
3137                if (!dtrace_priv_proc(state, mstate))
3138                        return (0);

3140                if (!(mstate->dtms_present & DTRACE_MSTATE_UCALLER)) {
3141                        uint64_t ustack[3];

3143                        /*
3144                         * dtrace_getupcstack() fills in the first uint64_t
3145                         * with the current PID.  The second uint64_t will
3146                         * be the program counter at user-level.  The third
3147                         * uint64_t will contain the caller, which is what
3148                         * we're after.
3149                         */
3150                        ustack[2] = NULL;
3151                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
3152                        dtrace_getupcstack(ustack, 3);
3153                        DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
3154                        mstate->dtms_ucaller = ustack[2];
3155                        mstate->dtms_present |= DTRACE_MSTATE_UCALLER;
3156                }

3158                return (mstate->dtms_ucaller);

3160        case DIF_VAR_PROBEPROV:
3161                ASSERT(mstate->dtms_present & DTRACE_MSTATE_PROBE);
3162                return (dtrace_dif_varstr(
3163                    (uintptr_t)mstate->dtms_probe->dtpr_provider->dtpv_name,
3164                    state, mstate));

3166        case DIF_VAR_PROBEMOD:
3167                ASSERT(mstate->dtms_present & DTRACE_MSTATE_PROBE);
3168                return (dtrace_dif_varstr(
3169                    (uintptr_t)mstate->dtms_probe->dtpr_mod,
3170                    state, mstate));

3172        case DIF_VAR_PROBEFUNC:
```



```
3173                ASSERT(mstate->dtms_present & DTRACE_MSTATE_PROBE);
3174                return (dtrace_dif_varstr(
3175                    (uintptr_t)mstate->dtms_probe->dtpr_func,
3176                    state, mstate));

3178        case DIF_VAR_PROBENAME:
3179                ASSERT(mstate->dtms_present & DTRACE_MSTATE_PROBE);
3180                return (dtrace_dif_varstr(
3181                    (uintptr_t)mstate->dtms_probe->dtpr_name,
3182                    state, mstate));

3184        case DIF_VAR_PID:
3185                if (!dtrace_priv_proc(state, mstate))
3186                        return (0);

3188                /*
3189                 * Note that we are assuming that an unanchored probe is
3190                 * always due to a high-level interrupt.  (And we're assuming
3191                 * that there is only a single high level interrupt.)
3192                 */
3193                if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3194                        return (pid0.pid_id);

3196                /*
3197                 * It is always safe to dereference one's own t_procp pointer:
3198                 * it always points to a valid, allocated proc structure.
3199                 * Further, it is always safe to dereference the p_pidp member
3200                 * of one's own proc structure.  (These are truisms becuase
3201                 * threads and processes don't clean up their own state --
3202                 * they leave that task to whomever reaps them.)
3203                 */
3204                return ((uint64_t)curthread->t_procp->p_pidp->pid_id);

3206        case DIF_VAR_PPID:
3207                if (!dtrace_priv_proc(state, mstate))
3208                        return (0);

3210                /*
3211                 * See comment in DIF_VAR_PID.
3212                 */
3213                if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3214                        return (pid0.pid_id);

3216                /*
3217                 * It is always safe to dereference one's own t_procp pointer:
3218                 * it always points to a valid, allocated proc structure.
3219                 * (This is true because threads don't clean up their own
3220                 * state -- they leave that task to whomever reaps them.)
3221                 */
3222                return ((uint64_t)curthread->t_procp->p_ppid);

3224        case DIF_VAR_TID:
3225                /*
3226                 * See comment in DIF_VAR_PID.
3227                 */
3228                if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3229                        return (0);

3231                return ((uint64_t)curthread->t_tid);

3233        case DIF_VAR_EXECNAME:
3234                if (!dtrace_priv_proc(state, mstate))
3235                        return (0);

3237                /*
3238                 * See comment in DIF_VAR_PID.
```

```
3239                 */
3240                if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3241                        return ((uint64_t)(uintptr_t)p0.p_user.u_comm);

3243                /*
3244                 * It is always safe to dereference one's own t_procp pointer:
3245                 * it always points to a valid, allocated proc structure.
3246                 * (This is true because threads don't clean up their own
3247                 * state -- they leave that task to whomever reaps them.)
3248                 */
3249                return (dtrace_dif_varstr(
3250                    (uintptr_t)curthread->t_procp->p_user.u_comm,
3251                    state, mstate));

3253        case DIF_VAR_ZONENAME:
3254                if (!dtrace_priv_proc(state, mstate))
3255                        return (0);

3257                /*
3258                 * See comment in DIF_VAR_PID.
3259                 */
3260                if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3261                        return ((uint64_t)(uintptr_t)p0.p_zone->zone_name);

3263                /*
3264                 * It is always safe to dereference one's own t_procp pointer:
3265                 * it always points to a valid, allocated proc structure.
3266                 * (This is true because threads don't clean up their own
3267                 * state -- they leave that task to whomever reaps them.)
3268                 */
3269                return (dtrace_dif_varstr(
3270                    (uintptr_t)curthread->t_procp->p_zone->zone_name,
3271                    state, mstate));

3273        case DIF_VAR_UID:
3274                if (!dtrace_priv_proc(state, mstate))
3275                        return (0);

3277                /*
3278                 * See comment in DIF_VAR_PID.
3279                 */
3280                if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3281                        return ((uint64_t)p0.p_cred->cr_uid);

3283                /*
3284                 * It is always safe to dereference one's own t_procp pointer:
3285                 * it always points to a valid, allocated proc structure.
3286                 * (This is true because threads don't clean up their own
3287                 * state -- they leave that task to whomever reaps them.)
3288                 *
3289                 * Additionally, it is safe to dereference one's own process
3290                 * credential, since this is never NULL after process birth.
3291                 */
3292                return ((uint64_t)curthread->t_procp->p_cred->cr_uid);

3294        case DIF_VAR_GID:
3295                if (!dtrace_priv_proc(state, mstate))
3296                        return (0);

3298                /*
3299                 * See comment in DIF_VAR_PID.
3300                 */
3301                if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3302                        return ((uint64_t)p0.p_cred->cr_gid);

3304                /*
```



```
3305                 * It is always safe to dereference one's own t_procp pointer:
3306                 * it always points to a valid, allocated proc structure.
3307                 * (This is true because threads don't clean up their own
3308                 * state -- they leave that task to whomever reaps them.)
3309                 *
3310                 * Additionally, it is safe to dereference one's own process
3311                 * credential, since this is never NULL after process birth.
3312                 */
3313                return ((uint64_t)curthread->t_procp->p_cred->cr_gid);

3315        case DIF_VAR_ERRNO: {
3316                klwp_t *lwp;
3317                if (!dtrace_priv_proc(state, mstate))
3318                        return (0);

3320                /*
3321                 * See comment in DIF_VAR_PID.
3322                 */
3323                if (DTRACE_ANCHORED(mstate->dtms_probe) && CPU_ON_INTR(CPU))
3324                        return (0);

3326                /*
3327                 * It is always safe to dereference one's own t_lwp pointer in
3328                 * the event that this pointer is non-NULL.  (This is true
3329                 * because threads and lwps don't clean up their own state --
3330                 * they leave that task to whomever reaps them.)
3331                 */
3332                if ((lwp = curthread->t_lwp) == NULL)
3333                        return (0);

3335                return ((uint64_t)lwp->lwp_errno);
3336        }
3337        default:
3338                DTRACE_CPUFLAG_SET(CPU_DTRACE_ILLOP);
3339                return (0);
3340        }
3341 }

3343 /*
3344  * Emulate the execution of DTrace ID subroutines invoked by the call opcode.
3345  * Notice that we don't bother validating the proper number of arguments or
3346  * their types in the tuple stack.  This isn't needed because all argument
3347  * interpretation is safe because of our load safety -- the worst that can
3348  * happen is that a bogus program can obtain bogus results.
3349  */
3350 static void
3351 dtrace_dif_subr(uint_t subr, uint_t rd, uint64_t *regs,
3352     dtrace_key_t *tupregs, int nargs,
3353     dtrace_mstate_t *mstate, dtrace_state_t *state)
3354 {
3355        volatile uint16_t *flags = &cpu_core[CPU->cpu_id].cpuc_dtrace_flags;
3356        volatile uintptr_t *illval = &cpu_core[CPU->cpu_id].cpuc_dtrace_illval;
3357        dtrace_vstate_t *vstate = &state->dts_vstate;

3359        union {
3360                mutex_impl_t mi;
3361                uint64_t mx;
3362        } m;

3364        union {
3365                krwlock_t ri;
3366                uintptr_t rw;
3367        } r;

3369        switch (subr) {
3370        case DIF_SUBR_RAND:
```

```
3371                regs[rd] = (dtrace_gethrtime() * 2416 + 374441) % 1771875;
3372                break;

3374        case DIF_SUBR_MUTEX_OWNED:
3375                if (!dtrace_canload(tupregs[0].dttk_value, sizeof (kmutex_t),
3376                    mstate, vstate)) {
3377                        regs[rd] = NULL;
3378                        break;
3379                }

3381                m.mx = dtrace_load64(tupregs[0].dttk_value);
3382                if (MUTEX_TYPE_ADAPTIVE(&m.mi))
3383                        regs[rd] = MUTEX_OWNER(&m.mi) != MUTEX_NO_OWNER;
3384                else
3385                        regs[rd] = LOCK_HELD(&m.mi.m_spin.m_spinlock);
3386                break;

3388        case DIF_SUBR_MUTEX_OWNER:
3389                if (!dtrace_canload(tupregs[0].dttk_value, sizeof (kmutex_t),
3390                    mstate, vstate)) {
3391                        regs[rd] = NULL;
3392                        break;
3393                }

3395                m.mx = dtrace_load64(tupregs[0].dttk_value);
3396                if (MUTEX_TYPE_ADAPTIVE(&m.mi) &&
3397                    MUTEX_OWNER(&m.mi) != MUTEX_NO_OWNER)
3398                        regs[rd] = (uintptr_t)MUTEX_OWNER(&m.mi);
3399                else
3400                        regs[rd] = 0;
3401                break;

3403        case DIF_SUBR_MUTEX_TYPE_ADAPTIVE:
3404                if (!dtrace_canload(tupregs[0].dttk_value, sizeof (kmutex_t),
3405                    mstate, vstate)) {
3406                        regs[rd] = NULL;
3407                        break;
3408                }

3410                m.mx = dtrace_load64(tupregs[0].dttk_value);
3411                regs[rd] = MUTEX_TYPE_ADAPTIVE(&m.mi);
3412                break;

3414        case DIF_SUBR_MUTEX_TYPE_SPIN:
3415                if (!dtrace_canload(tupregs[0].dttk_value, sizeof (kmutex_t),
3416                    mstate, vstate)) {
3417                        regs[rd] = NULL;
3418                        break;
3419                }

3421                m.mx = dtrace_load64(tupregs[0].dttk_value);
3422                regs[rd] = MUTEX_TYPE_SPIN(&m.mi);
3423                break;

3425        case DIF_SUBR_RW_READ_HELD: {
3426                uintptr_t tmp;

3428                if (!dtrace_canload(tupregs[0].dttk_value, sizeof (uintptr_t),
3429                    mstate, vstate)) {
3430                        regs[rd] = NULL;
3431                        break;
3432                }

3434                r.rw = dtrace_loadptr(tupregs[0].dttk_value);
3435                regs[rd] = _RW_READ_HELD(&r.ri, tmp);
3436                break;
```



```
3437        }

3439        case DIF_SUBR_RW_WRITE_HELD:
3440                if (!dtrace_canload(tupregs[0].dttk_value, sizeof (krwlock_t),
3441                    mstate, vstate)) {
3442                        regs[rd] = NULL;
3443                        break;
3444                }

3446                r.rw = dtrace_loadptr(tupregs[0].dttk_value);
3447                regs[rd] = _RW_WRITE_HELD(&r.ri);
3448                break;

3450        case DIF_SUBR_RW_ISWRITER:
3451                if (!dtrace_canload(tupregs[0].dttk_value, sizeof (krwlock_t),
3452                    mstate, vstate)) {
3453                        regs[rd] = NULL;
3454                        break;
3455                }

3457                r.rw = dtrace_loadptr(tupregs[0].dttk_value);
3458                regs[rd] = _RW_ISWRITER(&r.ri);
3459                break;

3461        case DIF_SUBR_BCOPY: {
3462                /*
3463                 * We need to be sure that the destination is in the scratch
3464                 * region -- no other region is allowed.
3465                 */
3466                uintptr_t src = tupregs[0].dttk_value;
3467                uintptr_t dest = tupregs[1].dttk_value;
3468                size_t size = tupregs[2].dttk_value;

3470                if (!dtrace_inscratch(dest, size, mstate)) {
3471                        *flags |= CPU_DTRACE_BADADDR;
3472                        *illval = regs[rd];
3473                        break;
3474                }

3476                if (!dtrace_canload(src, size, mstate, vstate)) {
3477                        regs[rd] = NULL;
3478                        break;
3479                }

3481                dtrace_bcopy((void *)src, (void *)dest, size);
3482                break;
3483        }

3485        case DIF_SUBR_ALLOCA:
3486        case DIF_SUBR_COPYIN: {
3487                uintptr_t dest = P2ROUNDUP(mstate->dtms_scratch_ptr, 8);
3488                uint64_t size =
3489                    tupregs[subr == DIF_SUBR_ALLOCA ? 0 : 1].dttk_value;
3490                size_t scratch_size = (dest - mstate->dtms_scratch_ptr) + size;

3492                /*
3493                 * This action doesn't require any credential checks since
3494                 * probes will not activate in user contexts to which the
3495                 * enabling user does not have permissions.
3496                 */

3498                /*
3499                 * Rounding up the user allocation size could have overflowed
3500                 * a large, bogus allocation (like -1ULL) to 0.
3501                 */
3502                if (scratch_size < size ||
```

```
3503                    !DTRACE_INSCRATCH(mstate, scratch_size)) {
3504                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
3505                        regs[rd] = NULL;
3506                        break;
3507                }

3509                if (subr == DIF_SUBR_COPYIN) {
3510                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
3511                        dtrace_copyin(tupregs[0].dttk_value, dest, size, flags);
3512                        DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
3513                }

3515                mstate->dtms_scratch_ptr += scratch_size;
3516                regs[rd] = dest;
3517                break;
3518        }

3520        case DIF_SUBR_COPYINTO: {
3521                uint64_t size = tupregs[1].dttk_value;
3522                uintptr_t dest = tupregs[2].dttk_value;

3524                /*
3525                 * This action doesn't require any credential checks since
3526                 * probes will not activate in user contexts to which the
3527                 * enabling user does not have permissions.
3528                 */
3529                if (!dtrace_inscratch(dest, size, mstate)) {
3530                        *flags |= CPU_DTRACE_BADADDR;
3531                        *illval = regs[rd];
3532                        break;
3533                }

3535                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
3536                dtrace_copyin(tupregs[0].dttk_value, dest, size, flags);
3537                DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
3538                break;
3539        }

3541        case DIF_SUBR_COPYINSTR: {
3542                uintptr_t dest = mstate->dtms_scratch_ptr;
3543                uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];

3545                if (nargs > 1 && tupregs[1].dttk_value < size)
3546                        size = tupregs[1].dttk_value + 1;

3548                /*
3549                 * This action doesn't require any credential checks since
3550                 * probes will not activate in user contexts to which the
3551                 * enabling user does not have permissions.
3552                 */
3553                if (!DTRACE_INSCRATCH(mstate, size)) {
3554                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
3555                        regs[rd] = NULL;
3556                        break;
3557                }

3559                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
3560                dtrace_copyinstr(tupregs[0].dttk_value, dest, size, flags);
3561                DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);

3563                ((char *)dest)[size - 1] = '\0';
3564                mstate->dtms_scratch_ptr += size;
3565                regs[rd] = dest;
3566                break;
3567        }
```



```
3569        case DIF_SUBR_MSGSIZE:
3570        case DIF_SUBR_MSGDSIZE: {
3571                uintptr_t baddr = tupregs[0].dttk_value, daddr;
3572                uintptr_t wptr, rptr;
3573                size_t count = 0;
3574                int cont = 0;

3576                while (baddr != NULL && !(*flags & CPU_DTRACE_FAULT)) {

3578                        if (!dtrace_canload(baddr, sizeof (mblk_t), mstate,
3579                            vstate)) {
3580                                regs[rd] = NULL;
3581                                break;
3582                        }

3584                        wptr = dtrace_loadptr(baddr +
3585                            offsetof(mblk_t, b_wptr));

3587                        rptr = dtrace_loadptr(baddr +
3588                            offsetof(mblk_t, b_rptr));

3590                        if (wptr < rptr) {
3591                                *flags |= CPU_DTRACE_BADADDR;
3592                                *illval = tupregs[0].dttk_value;
3593                                break;
3594                        }

3596                        daddr = dtrace_loadptr(baddr +
3597                            offsetof(mblk_t, b_datap));

3599                        baddr = dtrace_loadptr(baddr +
3600                            offsetof(mblk_t, b_cont));

3602                        /*
3603                         * We want to prevent against denial-of-service here,
3604                         * so we're only going to search the list for
3605                         * dtrace_msgdsize_max mblks.
3606                         */
3607                        if (cont++ > dtrace_msgdsize_max) {
3608                                *flags |= CPU_DTRACE_ILLOP;
3609                                break;
3610                        }

3612                        if (subr == DIF_SUBR_MSGDSIZE) {
3613                                if (dtrace_load8(daddr +
3614                                    offsetof(dblk_t, db_type)) != M_DATA)
3615                                        continue;
3616                        }

3618                        count += wptr - rptr;
3619                }

3621                if (!(*flags & CPU_DTRACE_FAULT))
3622                        regs[rd] = count;

3624                break;
3625        }

3627        case DIF_SUBR_PROGENYOF: {
3628                pid_t pid = tupregs[0].dttk_value;
3629                proc_t *p;
3630                int rval = 0;

3632                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);

3634                for (p = curthread->t_procp; p != NULL; p = p->p_parent) {
```

```
3635                        if (p->p_pidp->pid_id == pid) {
3636                                rval = 1;
3637                                break;
3638                        }
3639                }

3641                DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);

3643                regs[rd] = rval;
3644                break;
3645        }

3647        case DIF_SUBR_SPECULATION:
3648                regs[rd] = dtrace_speculation(state);
3649                break;

3651        case DIF_SUBR_COPYOUT: {
3652                uintptr_t kaddr = tupregs[0].dttk_value;
3653                uintptr_t uaddr = tupregs[1].dttk_value;
3654                uint64_t size = tupregs[2].dttk_value;

3656                if (!dtrace_destructive_disallow &&
3657                    dtrace_priv_proc_control(state, mstate) &&
3658                    !dtrace_istoxic(kaddr, size)) {
3659                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
3660                        dtrace_copyout(kaddr, uaddr, size, flags);
3661                        DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
3662                }
3663                break;
3664        }

3666        case DIF_SUBR_COPYOUTSTR: {
3667                uintptr_t kaddr = tupregs[0].dttk_value;
3668                uintptr_t uaddr = tupregs[1].dttk_value;
3669                uint64_t size = tupregs[2].dttk_value;

3671                if (!dtrace_destructive_disallow &&
3672                    dtrace_priv_proc_control(state, mstate) &&
3673                    !dtrace_istoxic(kaddr, size)) {
3674                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
3675                        dtrace_copyoutstr(kaddr, uaddr, size, flags);
3676                        DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
3677                }
3678                break;
3679        }

3681        case DIF_SUBR_STRLEN: {
3682                size_t sz;
3683                uintptr_t addr = (uintptr_t)tupregs[0].dttk_value;
3684                sz = dtrace_strlen((char *)addr,
3685                    state->dts_options[DTRACEOPT_STRSIZE]);

3687                if (!dtrace_canload(addr, sz + 1, mstate, vstate)) {
3688                        regs[rd] = NULL;
3689                        break;
3690                }

3692                regs[rd] = sz;

3694                break;
3695        }

3697        case DIF_SUBR_STRCHR:
3698        case DIF_SUBR_STRRCHR: {
3699                /*
3700                 * We're going to iterate over the string looking for the
```



```
3701                 * specified character.  We will iterate until we have reached
3702                 * the string length or we have found the character.  If this
3703                 * is DIF_SUBR_STRRCHR, we will look for the last occurrence
3704                 * of the specified character instead of the first.
3705                 */
3706                uintptr_t saddr = tupregs[0].dttk_value;
3707                uintptr_t addr = tupregs[0].dttk_value;
3708                uintptr_t limit = addr + state->dts_options[DTRACEOPT_STRSIZE];
3709                char c, target = (char)tupregs[1].dttk_value;

3711                for (regs[rd] = NULL; addr < limit; addr++) {
3712                        if ((c = dtrace_load8(addr)) == target) {
3713                                regs[rd] = addr;

3715                                if (subr == DIF_SUBR_STRCHR)
3716                                        break;
3717                        }

3719                        if (c == '\0')
3720                                break;
3721                }

3723                if (!dtrace_canload(saddr, addr - saddr, mstate, vstate)) {
3724                        regs[rd] = NULL;
3725                        break;
3726                }

3728                break;
3729        }

3731        case DIF_SUBR_STRSTR:
3732        case DIF_SUBR_INDEX:
3733        case DIF_SUBR_RINDEX: {
3734                /*
3735                 * We're going to iterate over the string looking for the
3736                 * specified string.  We will iterate until we have reached
3737                 * the string length or we have found the string.  (Yes, this
3738                 * is done in the most naive way possible -- but considering
3739                 * that the string we're searching for is likely to be
3740                 * relatively short, the complexity of Rabin-Karp or similar
3741                 * hardly seems merited.)
3742                 */
3743                char *addr = (char *)(uintptr_t)tupregs[0].dttk_value;
3744                char *substr = (char *)(uintptr_t)tupregs[1].dttk_value;
3745                uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
3746                size_t len = dtrace_strlen(addr, size);
3747                size_t sublen = dtrace_strlen(substr, size);
3748                char *limit = addr + len, *orig = addr;
3749                int notfound = subr == DIF_SUBR_STRSTR ? 0 : -1;
3750                int inc = 1;

3752                regs[rd] = notfound;

3754                if (!dtrace_canload((uintptr_t)addr, len + 1, mstate, vstate)) {
3755                        regs[rd] = NULL;
3756                        break;
3757                }

3759                if (!dtrace_canload((uintptr_t)substr, sublen + 1, mstate,
3760                    vstate)) {
3761                        regs[rd] = NULL;
3762                        break;
3763                }

3765                /*
3766                 * strstr() and index()/rindex() have similar semantics if
```

```
3767                 * both strings are the empty string: strstr() returns a
3768                 * pointer to the (empty) string, and index() and rindex()
3769                 * both return index 0 (regardless of any position argument).
3770                 */
3771                if (sublen == 0 && len == 0) {
3772                        if (subr == DIF_SUBR_STRSTR)
3773                                regs[rd] = (uintptr_t)addr;
3774                        else
3775                                regs[rd] = 0;
3776                        break;
3777                }

3779                if (subr != DIF_SUBR_STRSTR) {
3780                        if (subr == DIF_SUBR_RINDEX) {
3781                                limit = orig - 1;
3782                                addr += len;
3783                                inc = -1;
3784                        }

3786                        /*
3787                         * Both index() and rindex() take an optional position
3788                         * argument that denotes the starting position.
3789                         */
3790                        if (nargs == 3) {
3791                                int64_t pos = (int64_t)tupregs[2].dttk_value;

3793                                /*
3794                                 * If the position argument to index() is
3795                                 * negative, Perl implicitly clamps it at
3796                                 * zero.  This semantic is a little surprising
3797                                 * given the special meaning of negative
3798                                 * positions to similar Perl functions like
3799                                 * substr(), but it appears to reflect a
3800                                 * notion that index() can start from a
3801                                 * negative index and increment its way up to
3802                                 * the string.  Given this notion, Perl's
3803                                 * rindex() is at least self-consistent in
3804                                 * that it implicitly clamps positions greater
3805                                 * than the string length to be the string
3806                                 * length.  Where Perl completely loses
3807                                 * coherence, however, is when the specified
3808                                 * substring is the empty string ("").  In
3809                                 * this case, even if the position is
3810                                 * negative, rindex() returns 0 -- and even if
3811                                 * the position is greater than the length,
3812                                 * index() returns the string length.  These
3813                                 * semantics violate the notion that index()
3814                                 * should never return a value less than the
3815                                 * specified position and that rindex() should
3816                                 * never return a value greater than the
3817                                 * specified position.  (One assumes that
3818                                 * these semantics are artifacts of Perl's
3819                                 * implementation and not the results of
3820                                 * deliberate design -- it beggars belief that
3821                                 * even Larry Wall could desire such oddness.)
3822                                 * While in the abstract one would wish for
3823                                 * consistent position semantics across
3824                                 * substr(), index() and rindex() -- or at the
3825                                 * very least self-consistent position
3826                                 * semantics for index() and rindex() -- we
3827                                 * instead opt to keep with the extant Perl
3828                                 * semantics, in all their broken glory.  (Do
3829                                 * we have more desire to maintain Perl's
3830                                 * semantics than Perl does?  Probably.)
3831                                 */
3832                                if (subr == DIF_SUBR_RINDEX) {
```



```
3833                                        if (pos < 0) {
3834                                                if (sublen == 0)
3835                                                        regs[rd] = 0;
3836                                                break;
3837                                        }

3839                                        if (pos > len)
3840                                                pos = len;
3841                                } else {
3842                                        if (pos < 0)
3843                                                pos = 0;

3845                                        if (pos >= len) {
3846                                                if (sublen == 0)
3847                                                        regs[rd] = len;
3848                                                break;
3849                                        }
3850                                }

3852                                addr = orig + pos;
3853                        }
3854                }

3856                for (regs[rd] = notfound; addr != limit; addr += inc) {
3857                        if (dtrace_strncmp(addr, substr, sublen) == 0) {
3858                                if (subr != DIF_SUBR_STRSTR) {
3859                                        /*
3860                                         * As D index() and rindex() are
3861                                         * modeled on Perl (and not on awk),
3862                                         * we return a zero-based (and not a
3863                                         * one-based) index.  (For you Perl
3864                                         * weenies: no, we're not going to add
3865                                         * $[ -- and shouldn't you be at a con
3866                                         * or something?)
3867                                         */
3868                                        regs[rd] = (uintptr_t)(addr - orig);
3869                                        break;
3870                                }

3872                                ASSERT(subr == DIF_SUBR_STRSTR);
3873                                regs[rd] = (uintptr_t)addr;
3874                                break;
3875                        }
3876                }

3878                break;
3879        }

3881        case DIF_SUBR_STRTOK: {
3882                uintptr_t addr = tupregs[0].dttk_value;
3883                uintptr_t tokaddr = tupregs[1].dttk_value;
3884                uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
3885                uintptr_t limit, toklimit = tokaddr + size;
3886                uint8_t c, tokmap[32];   /* 256 / 8 */
3887                char *dest = (char *)mstate->dtms_scratch_ptr;
3888                int i;

3890                /*
3891                 * Check both the token buffer and (later) the input buffer,
3892                 * since both could be non-scratch addresses.
3893                 */
3894                if (!dtrace_strcanload(tokaddr, size, mstate, vstate)) {
3895                        regs[rd] = NULL;
3896                        break;
3897                }
```

```
3899                if (!DTRACE_INSCRATCH(mstate, size)) {
3900                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
3901                        regs[rd] = NULL;
3902                        break;
3903                }

3905                if (addr == NULL) {
3906                        /*
3907                         * If the address specified is NULL, we use our saved
3908                         * strtok pointer from the mstate.  Note that this
3909                         * means that the saved strtok pointer is _only_
3910                         * valid within multiple enablings of the same probe --
3911                         * it behaves like an implicit clause-local variable.
3912                         */
3913                        addr = mstate->dtms_strtok;
3914                } else {
3915                        /*
3916                         * If the user-specified address is non-NULL we must
3917                         * access check it.  This is the only time we have
3918                         * a chance to do so, since this address may reside
3919                         * in the string table of this clause-- future calls
3920                         * (when we fetch addr from mstate->dtms_strtok)
3921                         * would fail this access check.
3922                         */
3923                        if (!dtrace_strcanload(addr, size, mstate, vstate)) {
3924                                regs[rd] = NULL;
3925                                break;
3926                        }
3927                }

3929                /*
3930                 * First, zero the token map, and then process the token
3931                 * string -- setting a bit in the map for every character
3932                 * found in the token string.
3933                 */
3934                for (i = 0; i < sizeof (tokmap); i++)
3935                        tokmap[i] = 0;

3937                for (; tokaddr < toklimit; tokaddr++) {
3938                        if ((c = dtrace_load8(tokaddr)) == '\0')
3939                                break;

3941                        ASSERT((c >> 3) < sizeof (tokmap));
3942                        tokmap[c >> 3] |= (1 << (c & 0x7));
3943                }

3945                for (limit = addr + size; addr < limit; addr++) {
3946                        /*
3947                         * We're looking for a character that is _not_ contained
3948                         * in the token string.
3949                         */
3950                        if ((c = dtrace_load8(addr)) == '\0')
3951                                break;

3953                        if (!(tokmap[c >> 3] & (1 << (c & 0x7))))
3954                                break;
3955                }

3957                if (c == '\0') {
3958                        /*
3959                         * We reached the end of the string without finding
3960                         * any character that was not in the token string.
3961                         * We return NULL in this case, and we set the saved
3962                         * address to NULL as well.
3963                         */
3964                        regs[rd] = NULL;
```



```
3965                        mstate->dtms_strtok = NULL;
3966                        break;
3967                }

3969                /*
3970                 * From here on, we're copying into the destination string.
3971                 */
3972                for (i = 0; addr < limit && i < size - 1; addr++) {
3973                        if ((c = dtrace_load8(addr)) == '\0')
3974                                break;

3976                        if (tokmap[c >> 3] & (1 << (c & 0x7)))
3977                                break;

3979                        ASSERT(i < size);
3980                        dest[i++] = c;
3981                }

3983                ASSERT(i < size);
3984                dest[i] = '\0';
3985                regs[rd] = (uintptr_t)dest;
3986                mstate->dtms_scratch_ptr += size;
3987                mstate->dtms_strtok = addr;
3988                break;
3989        }

3991        case DIF_SUBR_SUBSTR: {
3992                uintptr_t s = tupregs[0].dttk_value;
3993                uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
3994                char *d = (char *)mstate->dtms_scratch_ptr;
3995                int64_t index = (int64_t)tupregs[1].dttk_value;
3996                int64_t remaining = (int64_t)tupregs[2].dttk_value;
3997                size_t len = dtrace_strlen((char *)s, size);
3998                int64_t i;

4000                if (!dtrace_canload(s, len + 1, mstate, vstate)) {
4001                        regs[rd] = NULL;
4002                        break;
4003                }

4005                if (!DTRACE_INSCRATCH(mstate, size)) {
4006                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4007                        regs[rd] = NULL;
4008                        break;
4009                }

4011                if (nargs <= 2)
4012                        remaining = (int64_t)size;

4014                if (index < 0) {
4015                        index += len;

4017                        if (index < 0 && index + remaining > 0) {
4018                                remaining += index;
4019                                index = 0;
4020                        }
4021                }

4023                if (index >= len || index < 0) {
4024                        remaining = 0;
4025                } else if (remaining < 0) {
4026                        remaining += len - index;
4027                } else if (index + remaining > size) {
4028                        remaining = size - index;
4029                }
```

```
4031                 for (i = 0; i < remaining; i++) {
4032                         if ((d[i] = dtrace_load8(s + index + i)) == '\0')
4033                                 break;
4034                 }

4036                 d[i] = '\0';

4038                 mstate->dtms_scratch_ptr += size;
4039                 regs[rd] = (uintptr_t)d;
4040                 break;
4041         }

4043         case DIF_SUBR_TOUPPER:
4044         case DIF_SUBR_TOLOWER: {
4045                 uintptr_t s = tupregs[0].dttk_value;
4046                 uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
4047                 char *dest = (char *)mstate->dtms_scratch_ptr, c;
4048                 size_t len = dtrace_strlen((char *)s, size);
4049                 char lower, upper, convert;
4050                 int64_t i;

4052                 if (subr == DIF_SUBR_TOUPPER) {
4053                         lower = 'a';
4054                         upper = 'z';
4055                         convert = 'A';
4056                 } else {
4057                         lower = 'A';
4058                         upper = 'Z';
4059                         convert = 'a';
4060                 }

4062                 if (!dtrace_canload(s, len + 1, mstate, vstate)) {
4063                         regs[rd] = NULL;
4064                         break;
4065                 }

4067                 if (!DTRACE_INSCRATCH(mstate, size)) {
4068                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4069                         regs[rd] = NULL;
4070                         break;
4071                 }

4073                 for (i = 0; i < size - 1; i++) {
4074                         if ((c = dtrace_load8(s + i)) == '\0')
4075                                 break;

4077                         if (c >= lower && c <= upper)
4078                                 c = convert + (c - lower);

4080                         dest[i] = c;
4081                 }

4083                 ASSERT(i < size);
4084                 dest[i] = '\0';
4085                 regs[rd] = (uintptr_t)dest;
4086                 mstate->dtms_scratch_ptr += size;
4087                 break;
4088         }

4090 case DIF_SUBR_GETMAJOR:
4091 #ifdef _LP64
4092                 regs[rd] = (tupregs[0].dttk_value >> NBITSMINOR64) & MAXMAJ64;
4093 #else
4094                 regs[rd] = (tupregs[0].dttk_value >> NBITSMINOR) & MAXMAJ;
4095 #endif
4096                 break;
```



```
4098         case DIF_SUBR_GETMINOR:
4099 #ifdef _LP64
4100                 regs[rd] = tupregs[0].dttk_value & MAXMIN64;
4101 #else
4102                 regs[rd] = tupregs[0].dttk_value & MAXMIN;
4103 #endif
4104                 break;

4106         case DIF_SUBR_DDI_PATHNAME: {
4107                 /*
4108                  * This one is a galactic mess.  We are going to roughly
4109                  * emulate ddi_pathname(), but it's made more complicated
4110                  * by the fact that we (a) want to include the minor name and
4111                  * (b) must proceed iteratively instead of recursively.
4112                  */
4113                 uintptr_t dest = mstate->dtms_scratch_ptr;
4114                 uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
4115                 char *start = (char *)dest, *end = start + size - 1;
4116                 uintptr_t daddr = tupregs[0].dttk_value;
4117                 int64_t minor = (int64_t)tupregs[1].dttk_value;
4118                 char *s;
4119                 int i, len, depth = 0;

4121                 /*
4122                  * Due to all the pointer jumping we do and context we must
4123                  * rely upon, we just mandate that the user must have kernel
4124                  * read privileges to use this routine.
4125                  */
4126                 if ((mstate->dtms_access & DTRACE_ACCESS_KERNEL) == 0) {
4127                         *flags |= CPU_DTRACE_KPRIV;
4128                         *illval = daddr;
4129                         regs[rd] = NULL;
4130                 }

4132                 if (!DTRACE_INSCRATCH(mstate, size)) {
4133                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4134                         regs[rd] = NULL;
4135                         break;
4136                 }

4138                 *end = '\0';

4140                 /*
4141                  * We want to have a name for the minor.  In order to do this,
4142                  * we need to walk the minor list from the devinfo.  We want
4143                  * to be sure that we don't infinitely walk a circular list,
4144                  * so we check for circularity by sending a scout pointer
4145                  * ahead two elements for every element that we iterate over;
4146                  * if the list is circular, these will ultimately point to the
4147                  * same element.  You may recognize this little trick as the
4148                  * answer to a stupid interview question -- one that always
4149                  * seems to be asked by those who had to have it laboriously
4150                  * explained to them, and who can't even concisely describe
4151                  * the conditions under which one would be forced to resort to
4152                  * this technique.  Needless to say, those conditions are
4153                  * found here -- and probably only here.  Is this the only use
4154                  * of this infamous trick in shipping, production code?  If it
4155                  * isn't, it probably should be...
4156                  */
4157                 if (minor != -1) {
4158                         uintptr_t maddr = dtrace_loadptr(daddr +
4159                             offsetof(struct dev_info, devi_minor));

4161                         uintptr_t next = offsetof(struct ddi_minor_data, next);
4162                         uintptr_t name = offsetof(struct ddi_minor_data,
```

```
4163                            d_minor) + offsetof(struct ddi_minor, name);
4164                        uintptr_t dev = offsetof(struct ddi_minor_data,
4165                            d_minor) + offsetof(struct ddi_minor, dev);
4166                        uintptr_t scout;
4167
4168                        if (maddr != NULL)
4169                                scout = dtrace_loadptr(maddr + next);
4170
4171                        while (maddr != NULL && !(*flags & CPU_DTRACE_FAULT)) {
4172                                uint64_t m;
4173 #ifdef _LP64
4174                                m = dtrace_load64(maddr + dev) & MAXMIN64;
4175 #else
4176                                m = dtrace_load32(maddr + dev) & MAXMIN;
4177 #endif
4178                                if (m != minor) {
4179                                        maddr = dtrace_loadptr(maddr + next);
4180
4181                                        if (scout == NULL)
4182                                                continue;
4183
4184                                        scout = dtrace_loadptr(scout + next);
4185
4186                                        if (scout == NULL)
4187                                                continue;
4188
4189                                        scout = dtrace_loadptr(scout + next);
4190
4191                                        if (scout == NULL)
4192                                                continue;
4193
4194                                        if (scout == maddr) {
4195                                                *flags |= CPU_DTRACE_ILLOP;
4196                                                break;
4197                                        }
4198
4199                                        continue;
4200                                }
4201
4202                                /*
4203                                 * We have the minor data.  Now we need to
4204                                 * copy the minor's name into the end of the
4205                                 * pathname.
4206                                 */
4207                                s = (char *)dtrace_loadptr(maddr + name);
4208                                len = dtrace_strlen(s, size);
4209
4210                                if (*flags & CPU_DTRACE_FAULT)
4211                                        break;
4212
4213                                if (len != 0) {
4214                                        if ((end -= (len + 1)) < start)
4215                                                break;
4216
4217                                        *end = ':';
4218                                }
4219
4220                                for (i = 1; i <= len; i++)
4221                                        end[i] = dtrace_load8((uintptr_t)s++);
4222                                break;
4223                        }
4224                }
4225
4226                while (daddr != NULL && !(*flags & CPU_DTRACE_FAULT)) {
4227                        ddi_node_state_t devi_state;
```



```
4228
4229                        devi_state = dtrace_load32(daddr +
4230                            offsetof(struct dev_info, devi_node_state));
4231
4232                        if (*flags & CPU_DTRACE_FAULT)
4233                                break;
4234
4235                        if (devi_state >= DS_INITIALIZED) {
4236                                s = (char *)dtrace_loadptr(daddr +
4237                                    offsetof(struct dev_info, devi_addr));
4238                                len = dtrace_strlen(s, size);
4239
4240                                if (*flags & CPU_DTRACE_FAULT)
4241                                        break;
4242
4243                                if (len != 0) {
4244                                        if ((end -= (len + 1)) < start)
4245                                                break;
4246
4247                                        *end = '@';
4248                                }
4249
4250                                for (i = 1; i <= len; i++)
4251                                        end[i] = dtrace_load8((uintptr_t)s++);
4252                        }
4253
4254                        /*
4255                         * Now for the node name...
4256                         */
4257                        s = (char *)dtrace_loadptr(daddr +
4258                            offsetof(struct dev_info, devi_node_name));
4259
4260                        daddr = dtrace_loadptr(daddr +
4261                            offsetof(struct dev_info, devi_parent));
4262
4263                        /*
4264                         * If our parent is NULL (that is, if we're the root
4265                         * node), we're going to use the special path
4266                         * "devices".
4267                         */
4268                        if (daddr == NULL)
4269                                s = "devices";
4270
4271                        len = dtrace_strlen(s, size);
4272                        if (*flags & CPU_DTRACE_FAULT)
4273                                break;
4274
4275                        if ((end -= (len + 1)) < start)
4276                                break;
4277
4278                        for (i = 1; i <= len; i++)
4279                                end[i] = dtrace_load8((uintptr_t)s++);
4280                        *end = '/';
4281
4282                        if (depth++ > dtrace_devdepth_max) {
4283                                *flags |= CPU_DTRACE_ILLOP;
4284                                break;
4285                        }
4286                }
4287
4288                if (end < start)
4289                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4290
4291                if (daddr == NULL) {
4292                        regs[rd] = (uintptr_t)end;
4293                        mstate->dtms_scratch_ptr += size;
4294                }
```

```
4296                 break;
4297         }

4299         case DIF_SUBR_STRJOIN: {
4300                 char *d = (char *)mstate->dtms_scratch_ptr;
4301                 uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
4302                 uintptr_t s1 = tupregs[0].dttk_value;
4303                 uintptr_t s2 = tupregs[1].dttk_value;
4304                 int i = 0;

4306                 if (!dtrace_strcanload(s1, size, mstate, vstate) ||
4307                     !dtrace_strcanload(s2, size, mstate, vstate)) {
4308                         regs[rd] = NULL;
4309                         break;
4310                 }

4312                 if (!DTRACE_INSCRATCH(mstate, size)) {
4313                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4314                         regs[rd] = NULL;
4315                         break;
4316                 }

4318                 for (;;) {
4319                         if (i >= size) {
4320                                 DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4321                                 regs[rd] = NULL;
4322                                 break;
4323                         }

4325                         if ((d[i++] = dtrace_load8(s1++)) == '\0') {
4326                                 i--;
4327                                 break;
4328                         }
4329                 }

4331                 for (;;) {
4332                         if (i >= size) {
4333                                 DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4334                                 regs[rd] = NULL;
4335                                 break;
4336                         }

4338                         if ((d[i++] = dtrace_load8(s2++)) == '\0')
4339                                 break;
4340                 }

4342                 if (i < size) {
4343                         mstate->dtms_scratch_ptr += i;
4344                         regs[rd] = (uintptr_t)d;
4345                 }

4347                 break;
4348         }

4350         case DIF_SUBR_LLTOSTR: {
4351                 int64_t i = (int64_t)tupregs[0].dttk_value;
4352                 uint64_t val, digit;
4353                 uint64_t size = 65;     /* enough room for 2^64 in binary */
4354                 char *end = (char *)mstate->dtms_scratch_ptr + size - 1;
4355                 int base = 10;

4357                 if (nargs > 1) {
4358                         if ((base = tupregs[1].dttk_value) <= 1 ||
4359                             base > ('z' - 'a' + 1) + ('9' - '0' + 1)) {
4360                                 *flags |= CPU_DTRACE_ILLOP;
```



```
4361                                 break;
4362                         }
4363                 }

4365                 val = (base == 10 && i < 0) ? i * -1 : i;

4367                 if (!DTRACE_INSCRATCH(mstate, size)) {
4368                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4369                         regs[rd] = NULL;
4370                         break;
4371                 }

4373                 for (*end-- = '\0'; val; val /= base) {
4374                         if ((digit = val % base) <= '9' - '0') {
4375                                 *end-- = '0' + digit;
4376                         } else {
4377                                 *end-- = 'a' + (digit - ('9' - '0') - 1);
4378                         }
4379                 }

4381                 if (i == 0 && base == 16)
4382                         *end-- = '0';

4384                 if (base == 16)
4385                         *end-- = 'x';

4387                 if (i == 0 || base == 8 || base == 16)
4388                         *end-- = '0';

4390                 if (i < 0 && base == 10)
4391                         *end-- = '-';

4393                 regs[rd] = (uintptr_t)end + 1;
4394                 mstate->dtms_scratch_ptr += size;
4395                 break;
4396         }

4398         case DIF_SUBR_HTONS:
4399         case DIF_SUBR_NTOHS:
4400 #ifdef _BIG_ENDIAN
4401                 regs[rd] = (uint16_t)tupregs[0].dttk_value;
4402 #else
4403                 regs[rd] = DT_BSWAP_16((uint16_t)tupregs[0].dttk_value);
4404 #endif
4405                 break;


4408         case DIF_SUBR_HTONL:
4409         case DIF_SUBR_NTOHL:
4410 #ifdef _BIG_ENDIAN
4411                 regs[rd] = (uint32_t)tupregs[0].dttk_value;
4412 #else
4413                 regs[rd] = DT_BSWAP_32((uint32_t)tupregs[0].dttk_value);
4414 #endif
4415                 break;


4418         case DIF_SUBR_HTONLL:
4419         case DIF_SUBR_NTOHLL:
4420 #ifdef _BIG_ENDIAN
4421                 regs[rd] = (uint64_t)tupregs[0].dttk_value;
4422 #else
4423                 regs[rd] = DT_BSWAP_64((uint64_t)tupregs[0].dttk_value);
4424 #endif
4425                 break;
```

```
4428         case DIF_SUBR_DIRNAME:
4429         case DIF_SUBR_BASENAME: {
4430                 char *dest = (char *)mstate->dtms_scratch_ptr;
4431                 uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
4432                 uintptr_t src = tupregs[0].dttk_value;
4433                 int i, j, len = dtrace_strlen((char *)src, size);
4434                 int lastbase = -1, firstbase = -1, lastdir = -1;
4435                 int start, end;

4437                 if (!dtrace_canload(src, len + 1, mstate, vstate)) {
4438                         regs[rd] = NULL;
4439                         break;
4440                 }

4442                 if (!DTRACE_INSCRATCH(mstate, size)) {
4443                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4444                         regs[rd] = NULL;
4445                         break;
4446                 }

4448                 /*
4449                  * The basename and dirname for a zero-length string is
4450                  * defined to be "."
4451                  */
4452                 if (len == 0) {
4453                         len = 1;
4454                         src = (uintptr_t)".";
4455                 }

4457                 /*
4458                  * Start from the back of the string, moving back toward the
4459                  * front until we see a character that isn't a slash.  That
4460                  * character is the last character in the basename.
4461                  */
4462                 for (i = len - 1; i >= 0; i--) {
4463                         if (dtrace_load8(src + i) != '/')
4464                                 break;
4465                 }

4467                 if (i >= 0)
4468                         lastbase = i;

4470                 /*
4471                  * Starting from the last character in the basename, move
4472                  * towards the front until we find a slash.  The character
4473                  * that we processed immediately before that is the first
4474                  * character in the basename.
4475                  */
4476                 for (; i >= 0; i--) {
4477                         if (dtrace_load8(src + i) == '/')
4478                                 break;
4479                 }

4481                 if (i >= 0)
4482                         firstbase = i + 1;

4484                 /*
4485                  * Now keep going until we find a non-slash character.  That
4486                  * character is the last character in the dirname.
4487                  */
4488                 for (; i >= 0; i--) {
4489                         if (dtrace_load8(src + i) != '/')
4490                                 break;
4491                 }
```



```
4493                 if (i >= 0)
4494                         lastdir = i;

4496                 ASSERT(!(lastbase == -1 && firstbase != -1));
4497                 ASSERT(!(firstbase == -1 && lastdir != -1));

4499                 if (lastbase == -1) {
4500                         /*
4501                          * We didn't find a non-slash character.  We know that
4502                          * the length is non-zero, so the whole string must be
4503                          * slashes.  In either the dirname or the basename
4504                          * case, we return '/'.
4505                          */
4506                         ASSERT(firstbase == -1);
4507                         firstbase = lastbase = lastdir = 0;
4508                 }

4510                 if (firstbase == -1) {
4511                         /*
4512                          * The entire string consists only of a basename
4513                          * component.  If we're looking for dirname, we need
4514                          * to change our string to be just "."; if we're
4515                          * looking for a basename, we'll just set the first
4516                          * character of the basename to be 0.
4517                          */
4518                         if (subr == DIF_SUBR_DIRNAME) {
4519                                 ASSERT(lastdir == -1);
4520                                 src = (uintptr_t)".";
4521                                 lastdir = 0;
4522                         } else {
4523                                 firstbase = 0;
4524                         }
4525                 }

4527                 if (subr == DIF_SUBR_DIRNAME) {
4528                         if (lastdir == -1) {
4529                                 /*
4530                                  * We know that we have a slash in the name --
4531                                  * or lastdir would be set to 0, above.  And
4532                                  * because lastdir is -1, we know that this
4533                                  * slash must be the first character.  (That
4534                                  * is, the full string must be of the form
4535                                  * "/basename".)  In this case, the last
4536                                  * character of the directory name is 0.
4537                                  */
4538                                 lastdir = 0;
4539                         }

4541                         start = 0;
4542                         end = lastdir;
4543                 } else {
4544                         ASSERT(subr == DIF_SUBR_BASENAME);
4545                         ASSERT(firstbase != -1 && lastbase != -1);
4546                         start = firstbase;
4547                         end = lastbase;
4548                 }

4550                 for (i = start, j = 0; i <= end && j < size - 1; i++, j++)
4551                         dest[j] = dtrace_load8(src + i);

4553                 dest[j] = '\0';
4554                 regs[rd] = (uintptr_t)dest;
4555                 mstate->dtms_scratch_ptr += size;
4556                 break;
4557         }
```

```
4559         case DIF_SUBR_GETF: {
4560                 uintptr_t fd = tupregs[0].dttk_value;
4561                 uf_info_t *finfo = &curthread->t_procp->p_user.u_finfo;
4562                 file_t *fp;
4563
4564                 if (!dtrace_priv_proc(state, mstate)) {
4565                         regs[rd] = NULL;
4566                         break;
4567                 }
4568
4569                 /*
4570                  * This is safe because fi_nfiles only increases, and the
4571                  * fi_list array is not freed when the array size doubles.
4572                  * (See the comment in flist_grow() for details on the
4573                  * management of the u_finfo structure.)
4574                  */
4575                 fp = fd < finfo->fi_nfiles ? finfo->fi_list[fd].uf_file : NULL;
4576
4577                 mstate->dtms_getf = fp;
4578                 regs[rd] = (uintptr_t)fp;
4579                 break;
4580         }
4581
4582 #endif /* ! codereview */
4583         case DIF_SUBR_CLEANPATH: {
4584                 char *dest = (char *)mstate->dtms_scratch_ptr, c;
4585                 uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
4586                 uintptr_t src = tupregs[0].dttk_value;
4587                 int i = 0, j = 0;
4588                 zone_t *z;
4589 #endif /* ! codereview */
4590
4591                 if (!dtrace_strcanload(src, size, mstate, vstate)) {
4592                         regs[rd] = NULL;
4593                         break;
4594                 }
4595
4596                 if (!DTRACE_INSCRATCH(mstate, size)) {
4597                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4598                         regs[rd] = NULL;
4599                         break;
4600                 }
4601
4602                 /*
4603                  * Move forward, loading each character.
4604                  */
4605                 do {
4606                         c = dtrace_load8(src + i++);
4607 next:
4608                         if (j + 5 >= size)      /* 5 = strlen("/..c\0") */
4609                                 break;
4610
4611                         if (c != '/') {
4612                                 dest[j++] = c;
4613                                 continue;
4614                         }
4615
4616                         c = dtrace_load8(src + i++);
4617
4618                         if (c == '/') {
4619                                 /*
4620                                  * We have two slashes -- we can just advance
4621                                  * to the next character.
4622                                  */
4623                                 goto next;
4624                         }
```



```
4625
4626                         if (c != '.') {
4627                                 /*
4628                                  * This is not "." and it's not ".." -- we can
4629                                  * just store the "/" and this character and
4630                                  * drive on.
4631                                  */
4632                                 dest[j++] = '/';
4633                                 dest[j++] = c;
4634                                 continue;
4635                         }
4636
4637                         c = dtrace_load8(src + i++);
4638
4639                         if (c == '/') {
4640                                 /*
4641                                  * This is a "/./" component.  We're not going
4642                                  * to store anything in the destination buffer;
4643                                  * we're just going to go to the next component.
4644                                  */
4645                                 goto next;
4646                         }
4647
4648                         if (c != '.') {
4649                                 /*
4650                                  * This is not ".." -- we can just store the
4651                                  * "/." and this character and continue
4652                                  * processing.
4653                                  */
4654                                 dest[j++] = '/';
4655                                 dest[j++] = '.';
4656                                 dest[j++] = c;
4657                                 continue;
4658                         }
4659
4660                         c = dtrace_load8(src + i++);
4661
4662                         if (c != '/' && c != '\0') {
4663                                 /*
4664                                  * This is not ".." -- it's "..[mumble]".
4665                                  * We'll store the "/.." and this character
4666                                  * and continue processing.
4667                                  */
4668                                 dest[j++] = '/';
4669                                 dest[j++] = '.';
4670                                 dest[j++] = '.';
4671                                 dest[j++] = c;
4672                                 continue;
4673                         }
4674
4675                         /*
4676                          * This is "/../" or "/..\0".  We need to back up
4677                          * our destination pointer until we find a "/".
4678                          */
4679                         i--;
4680                         while (j != 0 && dest[--j] != '/')
4681                                 continue;
4682
4683                         if (c == '\0')
4684                                 dest[++j] = '/';
4685                 } while (c != '\0');
4686
4687                 dest[j] = '\0';
4688
4689                 if (mstate->dtms_getf != NULL &&
4690                     !(mstate->dtms_access & DTRACE_ACCESS_KERNEL) &&
```

```
4691                    (z = state->dts_cred.dcr_cred->cr_zone) != kcred->cr_zone) {
4692                        /*
4693                         * If we've done a getf() as a part of this ECB and we
4694                         * don't have kernel access (and we're not in the global
4695                         * zone), check if the path we cleaned up begins with
4696                         * the zone's root path, and trim it off if so.  Note
4697                         * that this is an output cleanliness issue, not a
4698                         * security issue: knowing one's zone root path does
4699                         * not enable privilege escalation.
4700                         */
4701                        if (strstr(dest, z->zone_rootpath) == dest)
4702                                dest += strlen(z->zone_rootpath) - 1;
4703                }

4705 #endif /* ! codereview */
4706                regs[rd] = (uintptr_t)dest;
4707                mstate->dtms_scratch_ptr += size;
4708                break;
4709        }

4711        case DIF_SUBR_INET_NTOA:
4712        case DIF_SUBR_INET_NTOA6:
4713        case DIF_SUBR_INET_NTOP: {
4714                size_t size;
4715                int af, argi, i;
4716                char *base, *end;

4718                if (subr == DIF_SUBR_INET_NTOP) {
4719                        af = (int)tupregs[0].dttk_value;
4720                        argi = 1;
4721                } else {
4722                        af = subr == DIF_SUBR_INET_NTOA ? AF_INET: AF_INET6;
4723                        argi = 0;
4724                }

4726                if (af == AF_INET) {
4727                        ipaddr_t ip4;
4728                        uint8_t *ptr8, val;

4730                        /*
4731                         * Safely load the IPv4 address.
4732                         */
4733                        ip4 = dtrace_load32(tupregs[argi].dttk_value);

4735                        /*
4736                         * Check an IPv4 string will fit in scratch.
4737                         */
4738                        size = INET_ADDRSTRLEN;
4739                        if (!DTRACE_INSCRATCH(mstate, size)) {
4740                                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4741                                regs[rd] = NULL;
4742                                break;
4743                        }
4744                        base = (char *)mstate->dtms_scratch_ptr;
4745                        end = (char *)mstate->dtms_scratch_ptr + size - 1;

4747                        /*
4748                         * Stringify as a dotted decimal quad.
4749                         */
4750                        *end-- = '\0';
4751                        ptr8 = (uint8_t *)&ip4;
4752                        for (i = 3; i >= 0; i--) {
4753                                val = ptr8[i];

4755                                if (val == 0) {
4756                                        *end-- = '0';
```



```
4757                                } else {
4758                                        for (; val; val /= 10) {
4759                                                *end-- = '0' + (val % 10);
4760                                        }
4761                                }

4763                                if (i > 0)
4764                                        *end-- = '.';
4765                        }
4766                        ASSERT(end + 1 >= base);

4768                } else if (af == AF_INET6) {
4769                        struct in6_addr ip6;
4770                        int firstzero, tryzero, numzero, v6end;
4771                        uint16_t val;
4772                        const char digits[] = "0123456789abcdef";

4774                        /*
4775                         * Stringify using RFC 1884 convention 2 - 16 bit
4776                         * hexadecimal values with a zero-run compression.
4777                         * Lower case hexadecimal digits are used.
4778                         *      eg, fe80::214:4fff:fe0b:76c8.
4779                         * The IPv4 embedded form is returned for inet_ntop,
4780                         * just the IPv4 string is returned for inet_ntoa6.
4781                         */

4783                        /*
4784                         * Safely load the IPv6 address.
4785                         */
4786                        dtrace_bcopy(
4787                            (void *)(uintptr_t)tupregs[argi].dttk_value,
4788                            (void *)(uintptr_t)&ip6, sizeof (struct in6_addr));

4790                        /*
4791                         * Check an IPv6 string will fit in scratch.
4792                         */
4793                        size = INET6_ADDRSTRLEN;
4794                        if (!DTRACE_INSCRATCH(mstate, size)) {
4795                                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4796                                regs[rd] = NULL;
4797                                break;
4798                        }
4799                        base = (char *)mstate->dtms_scratch_ptr;
4800                        end = (char *)mstate->dtms_scratch_ptr + size - 1;
4801                        *end-- = '\0';

4803                        /*
4804                         * Find the longest run of 16 bit zero values
4805                         * for the single allowed zero compression - "::".
4806                         */
4807                        firstzero = -1;
4808                        tryzero = -1;
4809                        numzero = 1;
4810                        for (i = 0; i < sizeof (struct in6_addr); i++) {
4811                                if (ip6._S6_un._S6_u8[i] == 0 &&
4812                                    tryzero == -1 && i % 2 == 0) {
4813                                        tryzero = i;
4814                                        continue;
4815                                }

4817                                if (tryzero != -1 &&
4818                                    (ip6._S6_un._S6_u8[i] != 0 ||
4819                                    i == sizeof (struct in6_addr) - 1)) {

4821                                        if (i - tryzero <= numzero) {
4822                                                tryzero = -1;
```

```
4823                                        continue;
4824                                }
4825
4826                                firstzero = tryzero;
4827                                numzero = i - i % 2 - tryzero;
4828                                tryzero = -1;
4829
4830                                if (ip6._S6_un._S6_u8[i] == 0 &&
4831                                    i == sizeof (struct in6_addr) - 1)
4832                                        numzero += 2;
4833                        }
4834                }
4835                ASSERT(firstzero + numzero <= sizeof (struct in6_addr));
4836
4837                /*
4838                 * Check for an IPv4 embedded address.
4839                 */
4840                v6end = sizeof (struct in6_addr) - 2;
4841                if (IN6_IS_ADDR_V4MAPPED(&ip6) ||
4842                    IN6_IS_ADDR_V4COMPAT(&ip6)) {
4843                        for (i = sizeof (struct in6_addr) - 1;
4844                            i >= DTRACE_V4MAPPED_OFFSET; i--) {
4845                                ASSERT(end >= base);
4846
4847                                val = ip6._S6_un._S6_u8[i];
4848
4849                                if (val == 0) {
4850                                        *end-- = '0';
4851                                } else {
4852                                        for (; val; val /= 10) {
4853                                                *end-- = '0' + val % 10;
4854                                        }
4855                                }
4856
4857                                if (i > DTRACE_V4MAPPED_OFFSET)
4858                                        *end-- = '.';
4859                        }
4860
4861                        if (subr == DIF_SUBR_INET_NTOA6)
4862                                goto inetout;
4863
4864                        /*
4865                         * Set v6end to skip the IPv4 address that
4866                         * we have already stringified.
4867                         */
4868                        v6end = 10;
4869                }
4870
4871                /*
4872                 * Build the IPv6 string by working through the
4873                 * address in reverse.
4874                 */
4875                for (i = v6end; i >= 0; i -= 2) {
4876                        ASSERT(end >= base);
4877
4878                        if (i == firstzero + numzero - 2) {
4879                                *end-- = ':';
4880                                *end-- = ':';
4881                                i -= numzero - 2;
4882                                continue;
4883                        }
4884
4885                        if (i < 14 && i != firstzero - 2)
4886                                *end-- = ':';
4887
4888                        val = (ip6._S6_un._S6_u8[i] << 8) +
```



```
4889                                    ip6._S6_un._S6_u8[i + 1];
4890
4891                                if (val == 0) {
4892                                        *end-- = '0';
4893                                } else {
4894                                        for (; val; val /= 16) {
4895                                                *end-- = digits[val % 16];
4896                                        }
4897                                }
4898                        }
4899                        ASSERT(end + 1 >= base);
4900
4901                } else {
4902                        /*
4903                         * The user didn't use AH_INET or AH_INET6.
4904                         */
4905                        DTRACE_CPUFLAG_SET(CPU_DTRACE_ILLOP);
4906                        regs[rd] = NULL;
4907                        break;
4908                }
4909
4910 inetout:       regs[rd] = (uintptr_t)end + 1;
4911                mstate->dtms_scratch_ptr += size;
4912                break;
4913        }
4914
4915        }
4916 }
4917
4918 /*
4919  * Emulate the execution of DTrace IR instructions specified by the given
4920  * DIF object.  This function is deliberately void of assertions as all of
4921  * the necessary checks are handled by a call to dtrace_difo_validate().
4922  */
4923 static uint64_t
4924 dtrace_dif_emulate(dtrace_difo_t *difo, dtrace_mstate_t *mstate,
4925     dtrace_vstate_t *vstate, dtrace_state_t *state)
4926 {
4927        const dif_instr_t *text = difo->dtdo_buf;
4928        const uint_t textlen = difo->dtdo_len;
4929        const char *strtab = difo->dtdo_strtab;
4930        const uint64_t *inttab = difo->dtdo_inttab;
4931
4932        uint64_t rval = 0;
4933        dtrace_statvar_t *svar;
4934        dtrace_dstate_t *dstate = &vstate->dtvs_dynvars;
4935        dtrace_difv_t *v;
4936        volatile uint16_t *flags = &cpu_core[CPU->cpu_id].cpuc_dtrace_flags;
4937        volatile uintptr_t *illval = &cpu_core[CPU->cpu_id].cpuc_dtrace_illval;
4938
4939        dtrace_key_t tupregs[DIF_DTR_NREGS + 2]; /* +2 for thread and id */
4940        uint64_t regs[DIF_DIR_NREGS];
4941        uint64_t *tmp;
4942
4943        uint8_t cc_n = 0, cc_z = 0, cc_v = 0, cc_c = 0;
4944        int64_t cc_r;
4945        uint_t pc = 0, id, opc;
4946        uint8_t ttop = 0;
4947        dif_instr_t instr;
4948        uint_t r1, r2, rd;
4949
4950        /*
4951         * We stash the current DIF object into the machine state: we need it
4952         * for subsequent access checking.
4953         */
4954        mstate->dtms_difo = difo;
```

```
4956         regs[DIF_REG_R0] = 0;           /* %r0 is fixed at zero */

4958         while (pc < textlen && !(*flags & CPU_DTRACE_FAULT)) {
4959                 opc = pc;

4961                 instr = text[pc++];
4962                 r1 = DIF_INSTR_R1(instr);
4963                 r2 = DIF_INSTR_R2(instr);
4964                 rd = DIF_INSTR_RD(instr);

4966                 switch (DIF_INSTR_OP(instr)) {
4967                 case DIF_OP_OR:
4968                         regs[rd] = regs[r1] | regs[r2];
4969                         break;
4970                 case DIF_OP_XOR:
4971                         regs[rd] = regs[r1] ^ regs[r2];
4972                         break;
4973                 case DIF_OP_AND:
4974                         regs[rd] = regs[r1] & regs[r2];
4975                         break;
4976                 case DIF_OP_SLL:
4977                         regs[rd] = regs[r1] << regs[r2];
4978                         break;
4979                 case DIF_OP_SRL:
4980                         regs[rd] = regs[r1] >> regs[r2];
4981                         break;
4982                 case DIF_OP_SUB:
4983                         regs[rd] = regs[r1] - regs[r2];
4984                         break;
4985                 case DIF_OP_ADD:
4986                         regs[rd] = regs[r1] + regs[r2];
4987                         break;
4988                 case DIF_OP_MUL:
4989                         regs[rd] = regs[r1] * regs[r2];
4990                         break;
4991                 case DIF_OP_SDIV:
4992                         if (regs[r2] == 0) {
4993                                 regs[rd] = 0;
4994                                 *flags |= CPU_DTRACE_DIVZERO;
4995                         } else {
4996                                 regs[rd] = (int64_t)regs[r1] /
4997                                     (int64_t)regs[r2];
4998                         }
4999                         break;

5001                 case DIF_OP_UDIV:
5002                         if (regs[r2] == 0) {
5003                                 regs[rd] = 0;
5004                                 *flags |= CPU_DTRACE_DIVZERO;
5005                         } else {
5006                                 regs[rd] = regs[r1] / regs[r2];
5007                         }
5008                         break;

5010                 case DIF_OP_SREM:
5011                         if (regs[r2] == 0) {
5012                                 regs[rd] = 0;
5013                                 *flags |= CPU_DTRACE_DIVZERO;
5014                         } else {
5015                                 regs[rd] = (int64_t)regs[r1] %
5016                                     (int64_t)regs[r2];
5017                         }
5018                         break;

5020                 case DIF_OP_UREM:
```



```
5021                         if (regs[r2] == 0) {
5022                                 regs[rd] = 0;
5023                                 *flags |= CPU_DTRACE_DIVZERO;
5024                         } else {
5025                                 regs[rd] = regs[r1] % regs[r2];
5026                         }
5027                         break;

5029                 case DIF_OP_NOT:
5030                         regs[rd] = ~regs[r1];
5031                         break;
5032                 case DIF_OP_MOV:
5033                         regs[rd] = regs[r1];
5034                         break;
5035                 case DIF_OP_CMP:
5036                         cc_r = regs[r1] - regs[r2];
5037                         cc_n = cc_r < 0;
5038                         cc_z = cc_r == 0;
5039                         cc_v = 0;
5040                         cc_c = regs[r1] < regs[r2];
5041                         break;
5042                 case DIF_OP_TST:
5043                         cc_n = cc_v = cc_c = 0;
5044                         cc_z = regs[r1] == 0;
5045                         break;
5046                 case DIF_OP_BA:
5047                         pc = DIF_INSTR_LABEL(instr);
5048                         break;
5049                 case DIF_OP_BE:
5050                         if (cc_z)
5051                                 pc = DIF_INSTR_LABEL(instr);
5052                         break;
5053                 case DIF_OP_BNE:
5054                         if (cc_z == 0)
5055                                 pc = DIF_INSTR_LABEL(instr);
5056                         break;
5057                 case DIF_OP_BG:
5058                         if ((cc_z | (cc_n ^ cc_v)) == 0)
5059                                 pc = DIF_INSTR_LABEL(instr);
5060                         break;
5061                 case DIF_OP_BGU:
5062                         if ((cc_c | cc_z) == 0)
5063                                 pc = DIF_INSTR_LABEL(instr);
5064                         break;
5065                 case DIF_OP_BGE:
5066                         if ((cc_n ^ cc_v) == 0)
5067                                 pc = DIF_INSTR_LABEL(instr);
5068                         break;
5069                 case DIF_OP_BGEU:
5070                         if (cc_c == 0)
5071                                 pc = DIF_INSTR_LABEL(instr);
5072                         break;
5073                 case DIF_OP_BL:
5074                         if (cc_n ^ cc_v)
5075                                 pc = DIF_INSTR_LABEL(instr);
5076                         break;
5077                 case DIF_OP_BLU:
5078                         if (cc_c)
5079                                 pc = DIF_INSTR_LABEL(instr);
5080                         break;
5081                 case DIF_OP_BLE:
5082                         if (cc_z | (cc_n ^ cc_v))
5083                                 pc = DIF_INSTR_LABEL(instr);
5084                         break;
5085                 case DIF_OP_BLEU:
5086                         if (cc_c | cc_z)
```

```
5087                                    pc = DIF_INSTR_LABEL(instr);
5088                            break;
5089                    case DIF_OP_RLDSB:
5090                            if (!dtrace_canload(regs[r1], 1, mstate, vstate))
2078                            if (!dtrace_canstore(regs[r1], 1, mstate, vstate)) {
2079                                    *flags |= CPU_DTRACE_KPRIV;
2080                                    *illval = regs[r1];
5091                                    break;
2082                            }
5092                            /*FALLTHROUGH*/
5093                    case DIF_OP_LDSB:
5094                            regs[rd] = (int8_t)dtrace_load8(regs[r1]);
5095                            break;
5096                    case DIF_OP_RLDSH:
5097                            if (!dtrace_canload(regs[r1], 2, mstate, vstate))
2088                            if (!dtrace_canstore(regs[r1], 2, mstate, vstate)) {
2089                                    *flags |= CPU_DTRACE_KPRIV;
2090                                    *illval = regs[r1];
5098                                    break;
2092                            }
5099                            /*FALLTHROUGH*/
5100                    case DIF_OP_LDSH:
5101                            regs[rd] = (int16_t)dtrace_load16(regs[r1]);
5102                            break;
5103                    case DIF_OP_RLDSW:
5104                            if (!dtrace_canload(regs[r1], 4, mstate, vstate))
2098                            if (!dtrace_canstore(regs[r1], 4, mstate, vstate)) {
2099                                    *flags |= CPU_DTRACE_KPRIV;
2100                                    *illval = regs[r1];
5105                                    break;
5102                            }
5106                            /*FALLTHROUGH*/
5107                    case DIF_OP_LDSW:
5108                            regs[rd] = (int32_t)dtrace_load32(regs[r1]);
5109                            break;
5110                    case DIF_OP_RLDUB:
5111                            if (!dtrace_canload(regs[r1], 1, mstate, vstate))
2108                            if (!dtrace_canstore(regs[r1], 1, mstate, vstate)) {
2109                                    *flags |= CPU_DTRACE_KPRIV;
2110                                    *illval = regs[r1];
5112                                    break;
2112                            }
5113                            /*FALLTHROUGH*/
5114                    case DIF_OP_LDUB:
5115                            regs[rd] = dtrace_load8(regs[r1]);
5116                            break;
5117                    case DIF_OP_RLDUH:
5118                            if (!dtrace_canload(regs[r1], 2, mstate, vstate))
2118                            if (!dtrace_canstore(regs[r1], 2, mstate, vstate)) {
2119                                    *flags |= CPU_DTRACE_KPRIV;
2120                                    *illval = regs[r1];
5119                                    break;
2122                            }
5120                            /*FALLTHROUGH*/
5121                    case DIF_OP_LDUH:
5122                            regs[rd] = dtrace_load16(regs[r1]);
5123                            break;
5124                    case DIF_OP_RLDUW:
5125                            if (!dtrace_canload(regs[r1], 4, mstate, vstate))
2128                            if (!dtrace_canstore(regs[r1], 4, mstate, vstate)) {
2129                                    *flags |= CPU_DTRACE_KPRIV;
2130                                    *illval = regs[r1];
5126                                    break;
2132                            }
5127                            /*FALLTHROUGH*/
5128                    case DIF_OP_LDUW:
```



```
5129                            regs[rd] = dtrace_load32(regs[r1]);
5130                            break;
5131                    case DIF_OP_RLDX:
5132                            if (!dtrace_canload(regs[r1], 8, mstate, vstate))
2138                            if (!dtrace_canstore(regs[r1], 8, mstate, vstate)) {
2139                                    *flags |= CPU_DTRACE_KPRIV;
2140                                    *illval = regs[r1];
5133                                    break;
2142                            }
5134                            /*FALLTHROUGH*/
5135                    case DIF_OP_LDX:
5136                            regs[rd] = dtrace_load64(regs[r1]);
5137                            break;
5138                    case DIF_OP_ULDSB:
5139                            regs[rd] = (int8_t)
5140                                dtrace_fuword8((void *)(uintptr_t)regs[r1]);
5141                            break;
5142                    case DIF_OP_ULDSH:
5143                            regs[rd] = (int16_t)
5144                                dtrace_fuword16((void *)(uintptr_t)regs[r1]);
5145                            break;
5146                    case DIF_OP_ULDSW:
5147                            regs[rd] = (int32_t)
5148                                dtrace_fuword32((void *)(uintptr_t)regs[r1]);
5149                            break;
5150                    case DIF_OP_ULDUB:
5151                            regs[rd] =
5152                                dtrace_fuword8((void *)(uintptr_t)regs[r1]);
5153                            break;
5154                    case DIF_OP_ULDUH:
5155                            regs[rd] =
5156                                dtrace_fuword16((void *)(uintptr_t)regs[r1]);
5157                            break;
5158                    case DIF_OP_ULDUW:
5159                            regs[rd] =
5160                                dtrace_fuword32((void *)(uintptr_t)regs[r1]);
5161                            break;
5162                    case DIF_OP_ULDX:
5163                            regs[rd] =
5164                                dtrace_fuword64((void *)(uintptr_t)regs[r1]);
5165                            break;
5166                    case DIF_OP_RET:
5167                            rval = regs[rd];
5168                            pc = textlen;
5169                            break;
5170                    case DIF_OP_NOP:
5171                            break;
5172                    case DIF_OP_SETX:
5173                            regs[rd] = inttab[DIF_INSTR_INTEGER(instr)];
5174                            break;
5175                    case DIF_OP_SETS:
5176                            regs[rd] = (uint64_t)(uintptr_t)
5177                                (strtab + DIF_INSTR_STRING(instr));
5178                            break;
5179                    case DIF_OP_SCMP: {
5180                            size_t sz = state->dts_options[DTRACEOPT_STRSIZE];
5181                            uintptr_t s1 = regs[r1];
5182                            uintptr_t s2 = regs[r2];

5184                            if (s1 != NULL &&
5185                                !dtrace_strcanload(s1, sz, mstate, vstate))
5186                                    break;
5187                            if (s2 != NULL &&
5188                                !dtrace_strcanload(s2, sz, mstate, vstate))
5189                                    break;
```

```
5191                        cc_r = dtrace_strncmp((char *)s1, (char *)s2, sz);

5193                        cc_n = cc_r < 0;
5194                        cc_z = cc_r == 0;
5195                        cc_v = cc_c = 0;
5196                        break;
5197                }
5198                case DIF_OP_LDGA:
5199                        regs[rd] = dtrace_dif_variable(mstate, state,
5200                            r1, regs[r2]);
5201                        break;
5202                case DIF_OP_LDGS:
5203                        id = DIF_INSTR_VAR(instr);

5205                        if (id >= DIF_VAR_OTHER_UBASE) {
5206                                uintptr_t a;

5208                                id -= DIF_VAR_OTHER_UBASE;
5209                                svar = vstate->dtvs_globals[id];
5210                                ASSERT(svar != NULL);
5211                                v = &svar->dtsv_var;

5213                                if (!(v->dtdv_type.dtdt_flags & DIF_TF_BYREF)) {
5214                                        regs[rd] = svar->dtsv_data;
5215                                        break;
5216                                }

5218                                a = (uintptr_t)svar->dtsv_data;

5220                                if (*(uint8_t *)a == UINT8_MAX) {
5221                                        /*
5222                                         * If the 0th byte is set to UINT8_MAX
5223                                         * then this is to be treated as a
5224                                         * reference to a NULL variable.
5225                                         */
5226                                        regs[rd] = NULL;
5227                                } else {
5228                                        regs[rd] = a + sizeof (uint64_t);
5229                                }

5231                                break;
5232                        }

5234                        regs[rd] = dtrace_dif_variable(mstate, state, id, 0);
5235                        break;

5237                case DIF_OP_STGS:
5238                        id = DIF_INSTR_VAR(instr);

5240                        ASSERT(id >= DIF_VAR_OTHER_UBASE);
5241                        id -= DIF_VAR_OTHER_UBASE;

5243                        svar = vstate->dtvs_globals[id];
5244                        ASSERT(svar != NULL);
5245                        v = &svar->dtsv_var;

5247                        if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
5248                                uintptr_t a = (uintptr_t)svar->dtsv_data;

5250                                ASSERT(a != NULL);
5251                                ASSERT(svar->dtsv_size != 0);

5253                                if (regs[rd] == NULL) {
5254                                        *(uint8_t *)a = UINT8_MAX;
5255                                        break;
5256                                } else {
```



```
5257                                        *(uint8_t *)a = 0;
5258                                        a += sizeof (uint64_t);
5259                                }
5260                                if (!dtrace_vcanload(
5261                                    (void *)(uintptr_t)regs[rd], &v->dtdv_type,
5262                                    mstate, vstate))
5263                                        break;

5265                                dtrace_vcopy((void *)(uintptr_t)regs[rd],
5266                                    (void *)a, &v->dtdv_type);
5267                                break;
5268                        }

5270                        svar->dtsv_data = regs[rd];
5271                        break;

5273                case DIF_OP_LDTA:
5274                        /*
5275                         * There are no DTrace built-in thread-local arrays at
5276                         * present.  This opcode is saved for future work.
5277                         */
5278                        *flags |= CPU_DTRACE_ILLOP;
5279                        regs[rd] = 0;
5280                        break;

5282                case DIF_OP_LDLS:
5283                        id = DIF_INSTR_VAR(instr);

5285                        if (id < DIF_VAR_OTHER_UBASE) {
5286                                /*
5287                                 * For now, this has no meaning.
5288                                 */
5289                                regs[rd] = 0;
5290                                break;
5291                        }

5293                        id -= DIF_VAR_OTHER_UBASE;

5295                        ASSERT(id < vstate->dtvs_nlocals);
5296                        ASSERT(vstate->dtvs_locals != NULL);

5298                        svar = vstate->dtvs_locals[id];
5299                        ASSERT(svar != NULL);
5300                        v = &svar->dtsv_var;

5302                        if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
5303                                uintptr_t a = (uintptr_t)svar->dtsv_data;
5304                                size_t sz = v->dtdv_type.dtdt_size;

5306                                sz += sizeof (uint64_t);
5307                                ASSERT(svar->dtsv_size == NCPU * sz);
5308                                a += CPU->cpu_id * sz;

5310                                if (*(uint8_t *)a == UINT8_MAX) {
5311                                        /*
5312                                         * If the 0th byte is set to UINT8_MAX
5313                                         * then this is to be treated as a
5314                                         * reference to a NULL variable.
5315                                         */
5316                                        regs[rd] = NULL;
5317                                } else {
5318                                        regs[rd] = a + sizeof (uint64_t);
5319                                }

5321                                break;
5322                        }
```

```
5324                        ASSERT(svar->dtsv_size == NCPU * sizeof (uint64_t));
5325                        tmp = (uint64_t *)(uintptr_t)svar->dtsv_data;
5326                        regs[rd] = tmp[CPU->cpu_id];
5327                        break;

5329                case DIF_OP_STLS:
5330                        id = DIF_INSTR_VAR(instr);

5332                        ASSERT(id >= DIF_VAR_OTHER_UBASE);
5333                        id -= DIF_VAR_OTHER_UBASE;
5334                        ASSERT(id < vstate->dtvs_nlocals);

5336                        ASSERT(vstate->dtvs_locals != NULL);
5337                        svar = vstate->dtvs_locals[id];
5338                        ASSERT(svar != NULL);
5339                        v = &svar->dtsv_var;

5341                        if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
5342                                uintptr_t a = (uintptr_t)svar->dtsv_data;
5343                                size_t sz = v->dtdv_type.dtdt_size;

5345                                sz += sizeof (uint64_t);
5346                                ASSERT(svar->dtsv_size == NCPU * sz);
5347                                a += CPU->cpu_id * sz;

5349                                if (regs[rd] == NULL) {
5350                                        *(uint8_t *)a = UINT8_MAX;
5351                                        break;
5352                                } else {
5353                                        *(uint8_t *)a = 0;
5354                                        a += sizeof (uint64_t);
5355                                }

5357                                if (!dtrace_vcanload(
5358                                    (void *)(uintptr_t)regs[rd], &v->dtdv_type,
5359                                    mstate, vstate))
5360                                        break;

5362                                dtrace_vcopy((void *)(uintptr_t)regs[rd],
5363                                    (void *)a, &v->dtdv_type);
5364                                break;
5365                        }

5367                        ASSERT(svar->dtsv_size == NCPU * sizeof (uint64_t));
5368                        tmp = (uint64_t *)(uintptr_t)svar->dtsv_data;
5369                        tmp[CPU->cpu_id] = regs[rd];
5370                        break;

5372                case DIF_OP_LDTS: {
5373                        dtrace_dynvar_t *dvar;
5374                        dtrace_key_t *key;

5376                        id = DIF_INSTR_VAR(instr);
5377                        ASSERT(id >= DIF_VAR_OTHER_UBASE);
5378                        id -= DIF_VAR_OTHER_UBASE;
5379                        v = &vstate->dtvs_tlocals[id];

5381                        key = &tupregs[DIF_DTR_NREGS];
5382                        key[0].dttk_value = (uint64_t)id;
5383                        key[0].dttk_size = 0;
5384                        DTRACE_TLS_THRKEY(key[1].dttk_value);
5385                        key[1].dttk_size = 0;

5387                        dvar = dtrace_dynvar(dstate, 2, key,
5388                            sizeof (uint64_t), DTRACE_DYNVAR_NOALLOC,
```



```
5389                            mstate, vstate);

5391                        if (dvar == NULL) {
5392                                regs[rd] = 0;
5393                                break;
5394                        }

5396                        if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
5397                                regs[rd] = (uint64_t)(uintptr_t)dvar->dtdv_data;
5398                        } else {
5399                                regs[rd] = *((uint64_t *)dvar->dtdv_data);
5400                        }

5402                        break;
5403                }

5405                case DIF_OP_STTS: {
5406                        dtrace_dynvar_t *dvar;
5407                        dtrace_key_t *key;

5409                        id = DIF_INSTR_VAR(instr);
5410                        ASSERT(id >= DIF_VAR_OTHER_UBASE);
5411                        id -= DIF_VAR_OTHER_UBASE;

5413                        key = &tupregs[DIF_DTR_NREGS];
5414                        key[0].dttk_value = (uint64_t)id;
5415                        key[0].dttk_size = 0;
5416                        DTRACE_TLS_THRKEY(key[1].dttk_value);
5417                        key[1].dttk_size = 0;
5418                        v = &vstate->dtvs_tlocals[id];

5420                        dvar = dtrace_dynvar(dstate, 2, key,
5421                            v->dtdv_type.dtdt_size > sizeof (uint64_t) ?
5422                            v->dtdv_type.dtdt_size : sizeof (uint64_t),
5423                            regs[rd] ? DTRACE_DYNVAR_ALLOC :
5424                            DTRACE_DYNVAR_DEALLOC, mstate, vstate);

5426                        /*
5427                         * Given that we're storing to thread-local data,
5428                         * we need to flush our predicate cache.
5429                         */
5430                        curthread->t_predcache = NULL;

5432                        if (dvar == NULL)
5433                                break;

5435                        if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
5436                                if (!dtrace_vcanload(
5437                                    (void *)(uintptr_t)regs[rd],
5438                                    &v->dtdv_type, mstate, vstate))
5439                                        break;

5441                                dtrace_vcopy((void *)(uintptr_t)regs[rd],
5442                                    dvar->dtdv_data, &v->dtdv_type);
5443                        } else {
5444                                *((uint64_t *)dvar->dtdv_data) = regs[rd];
5445                        }

5447                        break;
5448                }

5450                case DIF_OP_SRA:
5451                        regs[rd] = (int64_t)regs[r1] >> regs[r2];
5452                        break;

5454                case DIF_OP_CALL:
```

```
5455                        dtrace_dif_subr(DIF_INSTR_SUBR(instr), rd,
5456                            regs, tupregs, ttop, mstate, state);
5457                        break;

5459                case DIF_OP_PUSHTR:
5460                        if (ttop == DIF_DTR_NREGS) {
5461                                *flags |= CPU_DTRACE_TUPOFLOW;
5462                                break;
5463                        }

5465                        if (r1 == DIF_TYPE_STRING) {
5466                                /*
5467                                 * If this is a string type and the size is 0,
5468                                 * we'll use the system-wide default string
5469                                 * size.  Note that we are _not_ looking at
5470                                 * the value of the DTRACEOPT_STRSIZE option;
5471                                 * had this been set, we would expect to have
5472                                 * a non-zero size value in the "pushtr".
5473                                 */
5474                                tupregs[ttop].dttk_size =
5475                                    dtrace_strlen((char *)(uintptr_t)regs[rd],
5476                                    regs[r2] ? regs[r2] :
5477                                    dtrace_strsize_default) + 1;
5478                        } else {
5479                                tupregs[ttop].dttk_size = regs[r2];
5480                        }

5482                        tupregs[ttop++].dttk_value = regs[rd];
5483                        break;

5485                case DIF_OP_PUSHTV:
5486                        if (ttop == DIF_DTR_NREGS) {
5487                                *flags |= CPU_DTRACE_TUPOFLOW;
5488                                break;
5489                        }

5491                        tupregs[ttop].dttk_value = regs[rd];
5492                        tupregs[ttop++].dttk_size = 0;
5493                        break;

5495                case DIF_OP_POPTS:
5496                        if (ttop != 0)
5497                                ttop--;
5498                        break;

5500                case DIF_OP_FLUSHTS:
5501                        ttop = 0;
5502                        break;

5504                case DIF_OP_LDGAA:
5505                case DIF_OP_LDTAA: {
5506                        dtrace_dynvar_t *dvar;
5507                        dtrace_key_t *key = tupregs;
5508                        uint_t nkeys = ttop;

5510                        id = DIF_INSTR_VAR(instr);
5511                        ASSERT(id >= DIF_VAR_OTHER_UBASE);
5512                        id -= DIF_VAR_OTHER_UBASE;

5514                        key[nkeys].dttk_value = (uint64_t)id;
5515                        key[nkeys++].dttk_size = 0;

5517                        if (DIF_INSTR_OP(instr) == DIF_OP_LDTAA) {
5518                                DTRACE_TLS_THRKEY(key[nkeys].dttk_value);
5519                                key[nkeys++].dttk_size = 0;
5520                                v = &vstate->dtvs_tlocals[id];
```



```
5521                        } else {
5522                                v = &vstate->dtvs_globals[id]->dtsv_var;
5523                        }

5525                        dvar = dtrace_dynvar(dstate, nkeys, key,
5526                            v->dtdv_type.dtdt_size > sizeof (uint64_t) ?
5527                            v->dtdv_type.dtdt_size : sizeof (uint64_t),
5528                            DTRACE_DYNVAR_NOALLOC, mstate, vstate);

5530                        if (dvar == NULL) {
5531                                regs[rd] = 0;
5532                                break;
5533                        }

5535                        if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
5536                                regs[rd] = (uint64_t)(uintptr_t)dvar->dtdv_data;
5537                        } else {
5538                                regs[rd] = *((uint64_t *)dvar->dtdv_data);
5539                        }

5541                        break;
5542                }

5544                case DIF_OP_STGAA:
5545                case DIF_OP_STTAA: {
5546                        dtrace_dynvar_t *dvar;
5547                        dtrace_key_t *key = tupregs;
5548                        uint_t nkeys = ttop;

5550                        id = DIF_INSTR_VAR(instr);
5551                        ASSERT(id >= DIF_VAR_OTHER_UBASE);
5552                        id -= DIF_VAR_OTHER_UBASE;

5554                        key[nkeys].dttk_value = (uint64_t)id;
5555                        key[nkeys++].dttk_size = 0;

5557                        if (DIF_INSTR_OP(instr) == DIF_OP_STTAA) {
5558                                DTRACE_TLS_THRKEY(key[nkeys].dttk_value);
5559                                key[nkeys++].dttk_size = 0;
5560                                v = &vstate->dtvs_tlocals[id];
5561                        } else {
5562                                v = &vstate->dtvs_globals[id]->dtsv_var;
5563                        }

5565                        dvar = dtrace_dynvar(dstate, nkeys, key,
5566                            v->dtdv_type.dtdt_size > sizeof (uint64_t) ?
5567                            v->dtdv_type.dtdt_size : sizeof (uint64_t),
5568                            regs[rd] ? DTRACE_DYNVAR_ALLOC :
5569                            DTRACE_DYNVAR_DEALLOC, mstate, vstate);

5571                        if (dvar == NULL)
5572                                break;

5574                        if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF) {
5575                                if (!dtrace_vcanload(
5576                                    (void *)(uintptr_t)regs[rd], &v->dtdv_type,
5577                                    mstate, vstate))
5578                                        break;

5580                                dtrace_vcopy((void *)(uintptr_t)regs[rd],
5581                                    dvar->dtdv_data, &v->dtdv_type);
5582                        } else {
5583                                *((uint64_t *)dvar->dtdv_data) = regs[rd];
5584                        }

5586                        break;
```

```
5587                }

5589                case DIF_OP_ALLOCS: {
5590                        uintptr_t ptr = P2ROUNDUP(mstate->dtms_scratch_ptr, 8);
5591                        size_t size = ptr - mstate->dtms_scratch_ptr + regs[r1];

5593                        /*
5594                         * Rounding up the user allocation size could have
5595                         * overflowed large, bogus allocations (like -1ULL) to
5596                         * 0.
5597                         */
5598                        if (size < regs[r1] ||
5599                            !DTRACE_INSCRATCH(mstate, size)) {
5600                                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
5601                                regs[rd] = NULL;
5602                                break;
5603                        }

5605                        dtrace_bzero((void *) mstate->dtms_scratch_ptr, size);
5606                        mstate->dtms_scratch_ptr += size;
5607                        regs[rd] = ptr;
5608                        break;
5609                }

5611                case DIF_OP_COPYS:
5612                        if (!dtrace_canstore(regs[rd], regs[r2],
5613                            mstate, vstate)) {
5614                                *flags |= CPU_DTRACE_BADADDR;
5615                                *illval = regs[rd];
5616                                break;
5617                        }

5619                        if (!dtrace_canload(regs[r1], regs[r2], mstate, vstate))
5620                                break;

5622                        dtrace_bcopy((void *)(uintptr_t)regs[r1],
5623                            (void *)(uintptr_t)regs[rd], (size_t)regs[r2]);
5624                        break;

5626                case DIF_OP_STB:
5627                        if (!dtrace_canstore(regs[rd], 1, mstate, vstate)) {
5628                                *flags |= CPU_DTRACE_BADADDR;
5629                                *illval = regs[rd];
5630                                break;
5631                        }
5632                        *((uint8_t *)(uintptr_t)regs[rd]) = (uint8_t)regs[r1];
5633                        break;

5635                case DIF_OP_STH:
5636                        if (!dtrace_canstore(regs[rd], 2, mstate, vstate)) {
5637                                *flags |= CPU_DTRACE_BADADDR;
5638                                *illval = regs[rd];
5639                                break;
5640                        }
5641                        if (regs[rd] & 1) {
5642                                *flags |= CPU_DTRACE_BADALIGN;
5643                                *illval = regs[rd];
5644                                break;
5645                        }
5646                        *((uint16_t *)(uintptr_t)regs[rd]) = (uint16_t)regs[r1];
5647                        break;

5649                case DIF_OP_STW:
5650                        if (!dtrace_canstore(regs[rd], 4, mstate, vstate)) {
5651                                *flags |= CPU_DTRACE_BADADDR;
5652                                *illval = regs[rd];
```



```
5653                                break;
5654                        }
5655                        if (regs[rd] & 3) {
5656                                *flags |= CPU_DTRACE_BADALIGN;
5657                                *illval = regs[rd];
5658                                break;
5659                        }
5660                        *((uint32_t *)(uintptr_t)regs[rd]) = (uint32_t)regs[r1];
5661                        break;

5663                case DIF_OP_STX:
5664                        if (!dtrace_canstore(regs[rd], 8, mstate, vstate)) {
5665                                *flags |= CPU_DTRACE_BADADDR;
5666                                *illval = regs[rd];
5667                                break;
5668                        }
5669                        if (regs[rd] & 7) {
5670                                *flags |= CPU_DTRACE_BADALIGN;
5671                                *illval = regs[rd];
5672                                break;
5673                        }
5674                        *((uint64_t *)(uintptr_t)regs[rd]) = regs[r1];
5675                        break;
5676                }
5677        }

5679        if (!(*flags & CPU_DTRACE_FAULT))
5680                return (rval);

5682        mstate->dtms_fltoffs = opc * sizeof (dif_instr_t);
5683        mstate->dtms_present |= DTRACE_MSTATE_FLTOFFS;

5685        return (0);
5686 }
______unchanged_portion_omitted_

5972 /*
5973  * If you're looking for the epicenter of DTrace, you just found it.  This
5974  * is the function called by the provider to fire a probe -- from which all
5975  * subsequent probe-context DTrace activity emanates.
5976  */
5977 void
5978 dtrace_probe(dtrace_id_t id, uintptr_t arg0, uintptr_t arg1,
5979     uintptr_t arg2, uintptr_t arg3, uintptr_t arg4)
5980 {
5981        processorid_t cpuid;
5982        dtrace_icookie_t cookie;
5983        dtrace_probe_t *probe;
5984        dtrace_mstate_t mstate;
5985        dtrace_ecb_t *ecb;
5986        dtrace_action_t *act;
5987        intptr_t offs;
5988        size_t size;
5989        int vtime, onintr;
5990        volatile uint16_t *flags;
5991        hrtime_t now, end;

5993        /*
5994         * Kick out immediately if this CPU is still being born (in which case
5995         * curthread will be set to -1) or the current thread can't allow
5996         * probes in its current context.
5997         */
5998        if (((uintptr_t)curthread & 1) || (curthread->t_flag & T_DONTDTRACE))
5999                return;

6001        cookie = dtrace_interrupt_disable();
```

```
6002         probe = dtrace_probes[id - 1];
6003         cpuid = CPU->cpu_id;
6004         onintr = CPU_ON_INTR(CPU);

6006         CPU->cpu_dtrace_probes++;

6008         if (!onintr && probe->dtpr_predcache != DTRACE_CACHEIDNONE &&
6009             probe->dtpr_predcache == curthread->t_predcache) {
6010                 /*
6011                  * We have hit in the predicate cache; we know that
6012                  * this predicate would evaluate to be false.
6013                  */
6014                 dtrace_interrupt_enable(cookie);
6015                 return;
6016         }

6018         if (panic_quiesce) {
6019                 /*
6020                  * We don't trace anything if we're panicking.
6021                  */
6022                 dtrace_interrupt_enable(cookie);
6023                 return;
6024         }

6026         now = dtrace_gethrtime();
6027         vtime = dtrace_vtime_references != 0;

6029         if (vtime && curthread->t_dtrace_start)
6030                 curthread->t_dtrace_vtime += now - curthread->t_dtrace_start;

6032         mstate.dtms_difo = NULL;
6033         mstate.dtms_probe = probe;
6034         mstate.dtms_strtok = NULL;
6035         mstate.dtms_arg[0] = arg0;
6036         mstate.dtms_arg[1] = arg1;
6037         mstate.dtms_arg[2] = arg2;
6038         mstate.dtms_arg[3] = arg3;
6039         mstate.dtms_arg[4] = arg4;

6041         flags = (volatile uint16_t *)&cpu_core[cpuid].cpuc_dtrace_flags;

6043         for (ecb = probe->dtpr_ecb; ecb != NULL; ecb = ecb->dte_next) {
6044                 dtrace_predicate_t *pred = ecb->dte_predicate;
6045                 dtrace_state_t *state = ecb->dte_state;
6046                 dtrace_buffer_t *buf = &state->dts_buffer[cpuid];
6047                 dtrace_buffer_t *aggbuf = &state->dts_aggbuffer[cpuid];
6048                 dtrace_vstate_t *vstate = &state->dts_vstate;
6049                 dtrace_provider_t *prov = probe->dtpr_provider;
6050                 uint64_t tracememsize = 0;
6051                 int committed = 0;
6052                 caddr_t tomax;

6054                 /*
6055                  * A little subtlety with the following (seemingly innocuous)
6056                  * declaration of the automatic 'val':  by looking at the
6057                  * code, you might think that it could be declared in the
6058                  * action processing loop, below.  (That is, it's only used in
6059                  * the action processing loop.)  However, it must be declared
6060                  * out of that scope because in the case of DIF expression
6061                  * arguments to aggregating actions, one iteration of the
6062                  * action loop will use the last iteration's value.
6063                  */
6064 #ifdef lint
6065                 uint64_t val = 0;
6066 #else
6067                 uint64_t val;
```



```
6068 #endif

6070                 mstate.dtms_present = DTRACE_MSTATE_ARGS | DTRACE_MSTATE_PROBE;
6071                 mstate.dtms_access = DTRACE_ACCESS_ARGS | DTRACE_ACCESS_PROC;
6072                 mstate.dtms_getf = NULL;

6074 #endif /* ! codereview */
6075                 *flags &= ~CPU_DTRACE_ERROR;

6077                 if (prov == dtrace_provider) {
6078                         /*
6079                          * If dtrace itself is the provider of this probe,
6080                          * we're only going to continue processing the ECB if
6081                          * arg0 (the dtrace_state_t) is equal to the ECB's
6082                          * creating state.  (This prevents disjoint consumers
6083                          * from seeing one another's metaprobes.)
6084                          */
6085                         if (arg0 != (uint64_t)(uintptr_t)state)
6086                                 continue;
6087                 }

6089                 if (state->dts_activity != DTRACE_ACTIVITY_ACTIVE) {
6090                         /*
6091                          * We're not currently active.  If our provider isn't
6092                          * the dtrace pseudo provider, we're not interested.
6093                          */
6094                         if (prov != dtrace_provider)
6095                                 continue;

6097                         /*
6098                          * Now we must further check if we are in the BEGIN
6099                          * probe.  If we are, we will only continue processing
6100                          * if we're still in WARMUP -- if one BEGIN enabling
6101                          * has invoked the exit() action, we don't want to
6102                          * evaluate subsequent BEGIN enablings.
6103                          */
6104                         if (probe->dtpr_id == dtrace_probeid_begin &&
6105                             state->dts_activity != DTRACE_ACTIVITY_WARMUP) {
6106                                 ASSERT(state->dts_activity ==
6107                                     DTRACE_ACTIVITY_DRAINING);
6108                                 continue;
6109                         }
6110                 }

6112                 if (ecb->dte_cond && !dtrace_priv_probe(state, &mstate, ecb))
6113                         continue;

6115                 if (now - state->dts_alive > dtrace_deadman_timeout) {
6116                         /*
6117                          * We seem to be dead.  Unless we (a) have kernel
6118                          * destructive permissions (b) have explicitly enabled
6119                          * destructive actions and (c) destructive actions have
6120                          * not been disabled, we're going to transition into
6121                          * the KILLED state, from which no further processing
6122                          * on this state will be performed.
6123                          */
6124                         if (!dtrace_priv_kernel_destructive(state) ||
6125                             !state->dts_cred.dcr_destructive ||
6126                             dtrace_destructive_disallow) {
6127                                 void *activity = &state->dts_activity;
6128                                 dtrace_activity_t current;

6130                                 do {
6131                                         current = state->dts_activity;
6132                                 } while (dtrace_cas32(activity, current,
6133                                     DTRACE_ACTIVITY_KILLED) != current);
```

```
6135                                    continue;
6136                            }
6137                    }
6139                    if ((offs = dtrace_buffer_reserve(buf, ecb->dte_needed,
6140                        ecb->dte_alignment, state, &mstate)) < 0)
6141                            continue;
6143                    tomax = buf->dtb_tomax;
6144                    ASSERT(tomax != NULL);
6146                    if (ecb->dte_size != 0) {
6147                            dtrace_rechdr_t dtrh;
6148                            if (!(mstate.dtms_present & DTRACE_MSTATE_TIMESTAMP)) {
6149                                    mstate.dtms_timestamp = dtrace_gethrtime();
6150                                    mstate.dtms_present |= DTRACE_MSTATE_TIMESTAMP;
6151                            }
6152                            ASSERT3U(ecb->dte_size, >=, sizeof (dtrace_rechdr_t));
6153                            dtrh.dtrh_epid = ecb->dte_epid;
6154                            DTRACE_RECORD_STORE_TIMESTAMP(&dtrh,
6155                                mstate.dtms_timestamp);
6156                            *((dtrace_rechdr_t *)(tomax + offs)) = dtrh;
6157                    }
6159                    mstate.dtms_epid = ecb->dte_epid;
6160                    mstate.dtms_present |= DTRACE_MSTATE_EPID;
6162                    if (state->dts_cred.dcr_visible & DTRACE_CRV_KERNEL)
6163                            mstate.dtms_access |= DTRACE_ACCESS_KERNEL;
6165                    if (pred != NULL) {
6166                            dtrace_difo_t *dp = pred->dtp_difo;
6167                            int rval;
6169                            rval = dtrace_dif_emulate(dp, &mstate, vstate, state);
6171                            if (!(*flags & CPU_DTRACE_ERROR) && !rval) {
6172                                    dtrace_cacheid_t cid = probe->dtpr_predcache;
6174                                    if (cid != DTRACE_CACHEIDNONE && !onintr) {
6175                                            /*
6176                                             * Update the predicate cache...
6177                                             */
6178                                            ASSERT(cid == pred->dtp_cacheid);
6179                                            curthread->t_predcache = cid;
6180                                    }
6182                                    continue;
6183                            }
6184                    }
6186                    for (act = ecb->dte_action; !(*flags & CPU_DTRACE_ERROR) &&
6187                        act != NULL; act = act->dta_next) {
6188                            size_t valoffs;
6189                            dtrace_difo_t *dp;
6190                            dtrace_recdesc_t *rec = &act->dta_rec;
6192                            size = rec->dtrd_size;
6193                            valoffs = offs + rec->dtrd_offset;
6195                            if (DTRACEACT_ISAGG(act->dta_kind)) {
6196                                    uint64_t v = 0xbad;
6197                                    dtrace_aggregation_t *agg;
6199                                    agg = (dtrace_aggregation_t *)act;
```



```
6201                                    if ((dp = act->dta_difo) != NULL)
6202                                            v = dtrace_dif_emulate(dp,
6203                                                &mstate, vstate, state);
6205                                    if (*flags & CPU_DTRACE_ERROR)
6206                                            continue;
6208                                    /*
6209                                     * Note that we always pass the expression
6210                                     * value from the previous iteration of the
6211                                     * action loop.  This value will only be used
6212                                     * if there is an expression argument to the
6213                                     * aggregating action, denoted by the
6214                                     * dtag_hasarg field.
6215                                     */
6216                                    dtrace_aggregate(agg, buf,
6217                                        offs, aggbuf, v, val);
6218                                    continue;
6219                            }
6221                            switch (act->dta_kind) {
6222                            case DTRACEACT_STOP:
6223                                    if (dtrace_priv_proc_destructive(state,
6224                                        &mstate))
6225                                            dtrace_action_stop();
6226                                    continue;
6228                            case DTRACEACT_BREAKPOINT:
6229                                    if (dtrace_priv_kernel_destructive(state))
6230                                            dtrace_action_breakpoint(ecb);
6231                                    continue;
6233                            case DTRACEACT_PANIC:
6234                                    if (dtrace_priv_kernel_destructive(state))
6235                                            dtrace_action_panic(ecb);
6236                                    continue;
6238                            case DTRACEACT_STACK:
6239                                    if (!dtrace_priv_kernel(state))
6240                                            continue;
6242                                    dtrace_getpcstack((pc_t *)(tomax + valoffs),
6243                                        size / sizeof (pc_t), probe->dtpr_aframes,
6244                                        DTRACE_ANCHORED(probe) ? NULL :
6245                                        (uint32_t *)arg0);
6247                                    continue;
6249                            case DTRACEACT_JSTACK:
6250                            case DTRACEACT_USTACK:
6251                                    if (!dtrace_priv_proc(state, &mstate))
6252                                            continue;
6254                                    /*
6255                                     * See comment in DIF_VAR_PID.
6256                                     */
6257                                    if (DTRACE_ANCHORED(mstate.dtms_probe) &&
6258                                        CPU_ON_INTR(CPU)) {
6259                                            int depth = DTRACE_USTACK_NFRAMES(
6260                                                rec->dtrd_arg) + 1;
6262                                            dtrace_bzero((void *)(tomax + valoffs),
6263                                                DTRACE_USTACK_STRSIZE(rec->dtrd_arg)
6264                                                + depth * sizeof (uint64_t));
```

```
6266                                    continue;
6267                            }

6269                            if (DTRACE_USTACK_STRSIZE(rec->dtrd_arg) != 0 &&
6270                                curproc->p_dtrace_helpers != NULL) {
6271                                    /*
6272                                     * This is the slow path -- we have
6273                                     * allocated string space, and we're
6274                                     * getting the stack of a process that
6275                                     * has helpers.  Call into a separate
6276                                     * routine to perform this processing.
6277                                     */
6278                                    dtrace_action_ustack(&mstate, state,
6279                                        (uint64_t *)(tomax + valoffs),
6280                                        rec->dtrd_arg);
6281                                    continue;
6282                            }

6284                            /*
6285                             * Clear the string space, since there's no
6286                             * helper to do it for us.
6287                             */
6288                            if (DTRACE_USTACK_STRSIZE(rec->dtrd_arg) != 0) {
6289                                    int depth = DTRACE_USTACK_NFRAMES(
6290                                        rec->dtrd_arg);
6291                                    size_t strsize = DTRACE_USTACK_STRSIZE(
6292                                        rec->dtrd_arg);
6293                                    uint64_t *buf = (uint64_t *)(tomax +
6294                                        valoffs);
6295                                    void *strspace = &buf[depth + 1];

6297                                    dtrace_bzero(strspace,
6298                                        MIN(depth, strsize));
6299                            }

6301                            DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
6302                            dtrace_getupcstack((uint64_t *)
6303                                (tomax + valoffs),
6304                                DTRACE_USTACK_NFRAMES(rec->dtrd_arg) + 1);
6305                            DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
6306                            continue;

6308                    default:
6309                            break;
6310                    }

6312                    dp = act->dta_difo;
6313                    ASSERT(dp != NULL);

6315                    val = dtrace_dif_emulate(dp, &mstate, vstate, state);

6317                    if (*flags & CPU_DTRACE_ERROR)
6318                            continue;

6320                    switch (act->dta_kind) {
6321                    case DTRACEACT_SPECULATE: {
6322                            dtrace_rechdr_t *dtrh;

6324                            ASSERT(buf == &state->dts_buffer[cpuid]);
6325                            buf = dtrace_speculation_buffer(state,
6326                                cpuid, val);

6328                            if (buf == NULL) {
6329                                    *flags |= CPU_DTRACE_DROP;
6330                                    continue;
6331                            }
```



```
6333                            offs = dtrace_buffer_reserve(buf,
6334                                ecb->dte_needed, ecb->dte_alignment,
6335                                state, NULL);

6337                            if (offs < 0) {
6338                                    *flags |= CPU_DTRACE_DROP;
6339                                    continue;
6340                            }

6342                            tomax = buf->dtb_tomax;
6343                            ASSERT(tomax != NULL);

6345                            if (ecb->dte_size == 0)
6346                                    continue;

6348                            ASSERT3U(ecb->dte_size, >=,
6349                                sizeof (dtrace_rechdr_t));
6350                            dtrh = ((void *)(tomax + offs));
6351                            dtrh->dtrh_epid = ecb->dte_epid;
6352                            /*
6353                             * When the speculation is committed, all of
6354                             * the records in the speculative buffer will
6355                             * have their timestamps set to the commit
6356                             * time.  Until then, it is set to a sentinel
6357                             * value, for debugability.
6358                             */
6359                            DTRACE_RECORD_STORE_TIMESTAMP(dtrh, UINT64_MAX);
6360                            continue;
6361                    }

6363                    case DTRACEACT_CHILL:
6364                            if (dtrace_priv_kernel_destructive(state))
6365                                    dtrace_action_chill(&mstate, val);
6366                            continue;

6368                    case DTRACEACT_RAISE:
6369                            if (dtrace_priv_proc_destructive(state,
6370                                &mstate))
6371                                    dtrace_action_raise(val);
6372                            continue;

6374                    case DTRACEACT_COMMIT:
6375                            ASSERT(!committed);

6377                            /*
6378                             * We need to commit our buffer state.
6379                             */
6380                            if (ecb->dte_size)
6381                                    buf->dtb_offset = offs + ecb->dte_size;
6382                            buf = &state->dts_buffer[cpuid];
6383                            dtrace_speculation_commit(state, cpuid, val);
6384                            committed = 1;
6385                            continue;

6387                    case DTRACEACT_DISCARD:
6388                            dtrace_speculation_discard(state, cpuid, val);
6389                            continue;

6391                    case DTRACEACT_DIFEXPR:
6392                    case DTRACEACT_LIBACT:
6393                    case DTRACEACT_PRINTF:
6394                    case DTRACEACT_PRINTA:
6395                    case DTRACEACT_SYSTEM:
6396                    case DTRACEACT_FREOPEN:
6397                    case DTRACEACT_TRACEMEM:
```

```
6398                                    break;
6399
6400                            case DTRACEACT_TRACEMEM_DYNSIZE:
6401                                    tracememsize = val;
6402                                    break;
6403
6404                            case DTRACEACT_SYM:
6405                            case DTRACEACT_MOD:
6406                                    if (!dtrace_priv_kernel(state))
6407                                            continue;
6408                                    break;
6409
6410                            case DTRACEACT_USYM:
6411                            case DTRACEACT_UMOD:
6412                            case DTRACEACT_UADDR: {
6413                                    struct pid *pid = curthread->t_procp->p_pidp;
6414
6415                                    if (!dtrace_priv_proc(state, &mstate))
6416                                            continue;
6417
6418                                    DTRACE_STORE(uint64_t, tomax,
6419                                        valoffs, (uint64_t)pid->pid_id);
6420                                    DTRACE_STORE(uint64_t, tomax,
6421                                        valoffs + sizeof (uint64_t), val);
6422
6423                                    continue;
6424                            }
6425
6426                            case DTRACEACT_EXIT: {
6427                                    /*
6428                                     * For the exit action, we are going to attempt
6429                                     * to atomically set our activity to be
6430                                     * draining.  If this fails (either because
6431                                     * another CPU has beat us to the exit action,
6432                                     * or because our current activity is something
6433                                     * other than ACTIVE or WARMUP), we will
6434                                     * continue.  This assures that the exit action
6435                                     * can be successfully recorded at most once
6436                                     * when we're in the ACTIVE state.  If we're
6437                                     * encountering the exit() action while in
6438                                     * COOLDOWN, however, we want to honor the new
6439                                     * status code.  (We know that we're the only
6440                                     * thread in COOLDOWN, so there is no race.)
6441                                     */
6442                                    void *activity = &state->dts_activity;
6443                                    dtrace_activity_t current = state->dts_activity;
6444
6445                                    if (current == DTRACE_ACTIVITY_COOLDOWN)
6446                                            break;
6447
6448                                    if (current != DTRACE_ACTIVITY_WARMUP)
6449                                            current = DTRACE_ACTIVITY_ACTIVE;
6450
6451                                    if (dtrace_cas32(activity, current,
6452                                        DTRACE_ACTIVITY_DRAINING) != current) {
6453                                            *flags |= CPU_DTRACE_DROP;
6454                                            continue;
6455                                    }
6456
6457                                    break;
6458                            }
6459
6460                            default:
6461                                    ASSERT(0);
6462                            }
```



```
6463
6464                             if (dp->dtdo_rtype.dtdt_flags & DIF_TF_BYREF) {
6465                                     uintptr_t end = valoffs + size;
6466
6467                                     if (tracememsize != 0 &&
6468                                         valoffs + tracememsize < end) {
6469                                             end = valoffs + tracememsize;
6470                                             tracememsize = 0;
6471                                     }
6472
6473                                     if (!dtrace_vcanload((void *)(uintptr_t)val,
6474                                         &dp->dtdo_rtype, &mstate, vstate))
6475                                             continue;
6476
6477                                     /*
6478                                      * If this is a string, we're going to only
6479                                      * load until we find the zero byte -- after
6480                                      * which we'll store zero bytes.
6481                                      */
6482                                     if (dp->dtdo_rtype.dtdt_kind ==
6483                                         DIF_TYPE_STRING) {
6484                                             char c = '\0' + 1;
6485                                             int intuple = act->dta_intuple;
6486                                             size_t s;
6487
6488                                             for (s = 0; s < size; s++) {
6489                                                     if (c != '\0')
6490                                                             c = dtrace_load8(val++);
6491
6492                                                     DTRACE_STORE(uint8_t, tomax,
6493                                                         valoffs++, c);
6494
6495                                                     if (c == '\0' && intuple)
6496                                                             break;
6497                                             }
6498
6499                                             continue;
6500                                     }
6501
6502                                     while (valoffs < end) {
6503                                             DTRACE_STORE(uint8_t, tomax, valoffs++,
6504                                                 dtrace_load8(val++));
6505                                     }
6506
6507                                     continue;
6508                             }
6509
6510                             switch (size) {
6511                             case 0:
6512                                     break;
6513
6514                             case sizeof (uint8_t):
6515                                     DTRACE_STORE(uint8_t, tomax, valoffs, val);
6516                                     break;
6517                             case sizeof (uint16_t):
6518                                     DTRACE_STORE(uint16_t, tomax, valoffs, val);
6519                                     break;
6520                             case sizeof (uint32_t):
6521                                     DTRACE_STORE(uint32_t, tomax, valoffs, val);
6522                                     break;
6523                             case sizeof (uint64_t):
6524                                     DTRACE_STORE(uint64_t, tomax, valoffs, val);
6525                                     break;
6526                             default:
6527                                     /*
6528                                      * Any other size should have been returned by
6529                                      * reference, not by value.
```

```
6530                                 */
6531                                ASSERT(0);
6532                                break;
6533                        }
6534                }
6535
6536                if (*flags & CPU_DTRACE_DROP)
6537                        continue;
6538
6539                if (*flags & CPU_DTRACE_FAULT) {
6540                        int ndx;
6541                        dtrace_action_t *err;
6542
6543                        buf->dtb_errors++;
6544
6545                        if (probe->dtpr_id == dtrace_probeid_error) {
6546                                /*
6547                                 * There's nothing we can do -- we had an
6548                                 * error on the error probe.  We bump an
6549                                 * error counter to at least indicate that
6550                                 * this condition happened.
6551                                 */
6552                                dtrace_error(&state->dts_dblerrors);
6553                                continue;
6554                        }
6555
6556                        if (vtime) {
6557                                /*
6558                                 * Before recursing on dtrace_probe(), we
6559                                 * need to explicitly clear out our start
6560                                 * time to prevent it from being accumulated
6561                                 * into t_dtrace_vtime.
6562                                 */
6563                                curthread->t_dtrace_start = 0;
6564                        }
6565
6566                        /*
6567                         * Iterate over the actions to figure out which action
6568                         * we were processing when we experienced the error.
6569                         * Note that act points _past_ the faulting action; if
6570                         * act is ecb->dte_action, the fault was in the
6571                         * predicate, if it's ecb->dte_action->dta_next it's
6572                         * in action #1, and so on.
6573                         */
6574                        for (err = ecb->dte_action, ndx = 0;
6575                            err != act; err = err->dta_next, ndx++)
6576                                continue;
6577
6578                        dtrace_probe_error(state, ecb->dte_epid, ndx,
6579                            (mstate.dtms_present & DTRACE_MSTATE_FLTOFFS) ?
6580                            mstate.dtms_fltoffs : -1, DTRACE_FLAGS2FLT(*flags),
6581                            cpu_core[cpuid].cpuc_dtrace_illval);
6582
6583                        continue;
6584                }
6585
6586                if (!committed)
6587                        buf->dtb_offset = offs + ecb->dte_size;
6588        }
6589
6590        end = dtrace_gethrtime();
6591        if (vtime)
6592                curthread->t_dtrace_start = end;
6593
6594        CPU->cpu_dtrace_nsec += end - now;
```



```
6596        dtrace_interrupt_enable(cookie);
6597 }
6598
6599 /*
6600  * DTrace Probe Hashing Functions
6601  *
6602  * The functions in this section (and indeed, the functions in remaining
6603  * sections) are not _called_ from probe context.  (Any exceptions to this are
6604  * marked with a "Note:".)  Rather, they are called from elsewhere in the
6605  * DTrace framework to look-up probes in, add probes to and remove probes from
6606  * the DTrace probe hashes.  (Each probe is hashed by each element of the
6607  * probe tuple -- allowing for fast lookups, regardless of what was
6608  * specified.)
6609  */
6610 static uint_t
6611 dtrace_hash_str(char *p)
6612 {
6613        unsigned int g;
6614        uint_t hval = 0;
6615
6616        while (*p) {
6617                hval = (hval << 4) + *p++;
6618                if ((g = (hval & 0xf0000000)) != 0)
6619                        hval ^= g >> 24;
6620                hval &= ~g;
6621        }
6622        return (hval);
6623 }
6624
6625 static dtrace_hash_t *
6626 dtrace_hash_create(uintptr_t stroffs, uintptr_t nextoffs, uintptr_t prevoffs)
6627 {
6628        dtrace_hash_t *hash = kmem_zalloc(sizeof (dtrace_hash_t), KM_SLEEP);
6629
6630        hash->dth_stroffs = stroffs;
6631        hash->dth_nextoffs = nextoffs;
6632        hash->dth_prevoffs = prevoffs;
6633
6634        hash->dth_size = 1;
6635        hash->dth_mask = hash->dth_size - 1;
6636
6637        hash->dth_tab = kmem_zalloc(hash->dth_size *
6638            sizeof (dtrace_hashbucket_t *), KM_SLEEP);
6639
6640        return (hash);
6641 }
6642
6643 static void
6644 dtrace_hash_destroy(dtrace_hash_t *hash)
6645 {
6646 #ifdef DEBUG
6647        int i;
6648
6649        for (i = 0; i < hash->dth_size; i++)
6650                ASSERT(hash->dth_tab[i] == NULL);
6651 #endif
6652
6653        kmem_free(hash->dth_tab,
6654            hash->dth_size * sizeof (dtrace_hashbucket_t *));
6655        kmem_free(hash, sizeof (dtrace_hash_t));
6656 }
6657
6658 static void
6659 dtrace_hash_resize(dtrace_hash_t *hash)
6660 {
6661        int size = hash->dth_size, i, ndx;
```

```
6662         int new_size = hash->dth_size << 1;
6663         int new_mask = new_size - 1;
6664         dtrace_hashbucket_t **new_tab, *bucket, *next;

6666         ASSERT((new_size & new_mask) == 0);

6668         new_tab = kmem_zalloc(new_size * sizeof (void *), KM_SLEEP);

6670         for (i = 0; i < size; i++) {
6671                 for (bucket = hash->dth_tab[i]; bucket != NULL; bucket = next) {
6672                         dtrace_probe_t *probe = bucket->dthb_chain;

6674                         ASSERT(probe != NULL);
6675                         ndx = DTRACE_HASHSTR(hash, probe) & new_mask;

6677                         next = bucket->dthb_next;
6678                         bucket->dthb_next = new_tab[ndx];
6679                         new_tab[ndx] = bucket;
6680                 }
6681         }

6683         kmem_free(hash->dth_tab, hash->dth_size * sizeof (void *));
6684         hash->dth_tab = new_tab;
6685         hash->dth_size = new_size;
6686         hash->dth_mask = new_mask;
6687 }

6689 static void
6690 dtrace_hash_add(dtrace_hash_t *hash, dtrace_probe_t *new)
6691 {
6692         int hashval = DTRACE_HASHSTR(hash, new);
6693         int ndx = hashval & hash->dth_mask;
6694         dtrace_hashbucket_t *bucket = hash->dth_tab[ndx];
6695         dtrace_probe_t **nextp, **prevp;

6697         for (; bucket != NULL; bucket = bucket->dthb_next) {
6698                 if (DTRACE_HASHEQ(hash, bucket->dthb_chain, new))
6699                         goto add;
6700         }

6702         if ((hash->dth_nbuckets >> 1) > hash->dth_size) {
6703                 dtrace_hash_resize(hash);
6704                 dtrace_hash_add(hash, new);
6705                 return;
6706         }

6708         bucket = kmem_zalloc(sizeof (dtrace_hashbucket_t), KM_SLEEP);
6709         bucket->dthb_next = hash->dth_tab[ndx];
6710         hash->dth_tab[ndx] = bucket;
6711         hash->dth_nbuckets++;

6713 add:
6714         nextp = DTRACE_HASHNEXT(hash, new);
6715         ASSERT(*nextp == NULL && *(DTRACE_HASHPREV(hash, new)) == NULL);
6716         *nextp = bucket->dthb_chain;

6718         if (bucket->dthb_chain != NULL) {
6719                 prevp = DTRACE_HASHPREV(hash, bucket->dthb_chain);
6720                 ASSERT(*prevp == NULL);
6721                 *prevp = new;
6722         }

6724         bucket->dthb_chain = new;
6725         bucket->dthb_len++;
6726 }
```



```
6728 static dtrace_probe_t *
6729 dtrace_hash_lookup(dtrace_hash_t *hash, dtrace_probe_t *template)
6730 {
6731         int hashval = DTRACE_HASHSTR(hash, template);
6732         int ndx = hashval & hash->dth_mask;
6733         dtrace_hashbucket_t *bucket = hash->dth_tab[ndx];

6735         for (; bucket != NULL; bucket = bucket->dthb_next) {
6736                 if (DTRACE_HASHEQ(hash, bucket->dthb_chain, template))
6737                         return (bucket->dthb_chain);
6738         }

6740         return (NULL);
6741 }

6743 static int
6744 dtrace_hash_collisions(dtrace_hash_t *hash, dtrace_probe_t *template)
6745 {
6746         int hashval = DTRACE_HASHSTR(hash, template);
6747         int ndx = hashval & hash->dth_mask;
6748         dtrace_hashbucket_t *bucket = hash->dth_tab[ndx];

6750         for (; bucket != NULL; bucket = bucket->dthb_next) {
6751                 if (DTRACE_HASHEQ(hash, bucket->dthb_chain, template))
6752                         return (bucket->dthb_len);
6753         }

6755         return (NULL);
6756 }

6758 static void
6759 dtrace_hash_remove(dtrace_hash_t *hash, dtrace_probe_t *probe)
6760 {
6761         int ndx = DTRACE_HASHSTR(hash, probe) & hash->dth_mask;
6762         dtrace_hashbucket_t *bucket = hash->dth_tab[ndx];

6764         dtrace_probe_t **prevp = DTRACE_HASHPREV(hash, probe);
6765         dtrace_probe_t **nextp = DTRACE_HASHNEXT(hash, probe);

6767         /*
6768          * Find the bucket that we're removing this probe from.
6769          */
6770         for (; bucket != NULL; bucket = bucket->dthb_next) {
6771                 if (DTRACE_HASHEQ(hash, bucket->dthb_chain, probe))
6772                         break;
6773         }

6775         ASSERT(bucket != NULL);

6777         if (*prevp == NULL) {
6778                 if (*nextp == NULL) {
6779                         /*
6780                          * The removed probe was the only probe on this
6781                          * bucket; we need to remove the bucket.
6782                          */
6783                         dtrace_hashbucket_t *b = hash->dth_tab[ndx];

6785                         ASSERT(bucket->dthb_chain == probe);
6786                         ASSERT(b != NULL);

6788                         if (b == bucket) {
6789                                 hash->dth_tab[ndx] = bucket->dthb_next;
6790                         } else {
6791                                 while (b->dthb_next != bucket)
6792                                         b = b->dthb_next;
6793                                 b->dthb_next = bucket->dthb_next;
```

```
6794                        }

6796                        ASSERT(hash->dth_nbuckets > 0);
6797                        hash->dth_nbuckets--;
6798                        kmem_free(bucket, sizeof (dtrace_hashbucket_t));
6799                        return;
6800                }

6802                bucket->dthb_chain = *nextp;
6803        } else {
6804                *(DTRACE_HASHNEXT(hash, *prevp)) = *nextp;
6805        }

6807        if (*nextp != NULL)
6808                *(DTRACE_HASHPREV(hash, *nextp)) = *prevp;
6809 }

6811 /*
6812  * DTrace Utility Functions
6813  *
6814  * These are random utility functions that are _not_ called from probe context.
6815  */
6816 static int
6817 dtrace_badattr(const dtrace_attribute_t *a)
6818 {
6819        return (a->dtat_name > DTRACE_STABILITY_MAX ||
6820            a->dtat_data > DTRACE_STABILITY_MAX ||
6821            a->dtat_class > DTRACE_CLASS_MAX);
6822 }

6824 /*
6825  * Return a duplicate copy of a string.  If the specified string is NULL,
6826  * this function returns a zero-length string.
6827  */
6828 static char *
6829 dtrace_strdup(const char *str)
6830 {
6831        char *new = kmem_zalloc((str != NULL ? strlen(str) : 0) + 1, KM_SLEEP);

6833        if (str != NULL)
6834                (void) strcpy(new, str);

6836        return (new);
6837 }

6839 #define DTRACE_ISALPHA(c)       \
6840        (((c) >= 'a' && (c) <= 'z') || ((c) >= 'A' && (c) <= 'Z'))

6842 static int
6843 dtrace_badname(const char *s)
6844 {
6845        char c;

6847        if (s == NULL || (c = *s++) == '\0')
6848                return (0);

6850        if (!DTRACE_ISALPHA(c) && c != '-' && c != '_' && c != '.')
6851                return (1);

6853        while ((c = *s++) != '\0') {
6854                if (!DTRACE_ISALPHA(c) && (c < '0' || c > '9') &&
6855                    c != '-' && c != '_' && c != '.' && c != '`')
6856                        return (1);
6857        }

6859        return (0);
```



```
6860 }

6862 static void
6863 dtrace_cred2priv(cred_t *cr, uint32_t *privp, uid_t *uidp, zoneid_t *zoneidp)
6864 {
6865        uint32_t priv;

6867        if (cr == NULL || PRIV_POLICY_ONLY(cr, PRIV_ALL, B_FALSE)) {
6868                /*
6869                 * For DTRACE_PRIV_ALL, the uid and zoneid don't matter.
6870                 */
6871                priv = DTRACE_PRIV_ALL;
6872        } else {
6873                *uidp = crgetuid(cr);
6874                *zoneidp = crgetzoneid(cr);

6876                priv = 0;
6877                if (PRIV_POLICY_ONLY(cr, PRIV_DTRACE_KERNEL, B_FALSE))
6878                        priv |= DTRACE_PRIV_KERNEL | DTRACE_PRIV_USER;
6879                else if (PRIV_POLICY_ONLY(cr, PRIV_DTRACE_USER, B_FALSE))
6880                        priv |= DTRACE_PRIV_USER;
6881                if (PRIV_POLICY_ONLY(cr, PRIV_DTRACE_PROC, B_FALSE))
6882                        priv |= DTRACE_PRIV_PROC;
6883                if (PRIV_POLICY_ONLY(cr, PRIV_PROC_OWNER, B_FALSE))
6884                        priv |= DTRACE_PRIV_OWNER;
6885                if (PRIV_POLICY_ONLY(cr, PRIV_PROC_ZONE, B_FALSE))
6886                        priv |= DTRACE_PRIV_ZONEOWNER;
6887        }

6889        *privp = priv;
6890 }

6892 #ifdef DTRACE_ERRDEBUG
6893 static void
6894 dtrace_errdebug(const char *str)
6895 {
6896        int hval = dtrace_hash_str((char *)str) % DTRACE_ERRHASHSZ;
6897        int occupied = 0;

6899        mutex_enter(&dtrace_errlock);
6900        dtrace_errlast = str;
6901        dtrace_errthread = curthread;

6903        while (occupied++ < DTRACE_ERRHASHSZ) {
6904                if (dtrace_errhash[hval].dter_msg == str) {
6905                        dtrace_errhash[hval].dter_count++;
6906                        goto out;
6907                }

6909                if (dtrace_errhash[hval].dter_msg != NULL) {
6910                        hval = (hval + 1) % DTRACE_ERRHASHSZ;
6911                        continue;
6912                }

6914                dtrace_errhash[hval].dter_msg = str;
6915                dtrace_errhash[hval].dter_count = 1;
6916                goto out;
6917        }

6919        panic("dtrace: undersized error hash");
6920 out:
6921        mutex_exit(&dtrace_errlock);
6922 }
6923 #endif

6925 /*
```

```
 * DTrace Matching Functions
 *
 * These functions are used to match groups of probes, given some elements of
 * a probe tuple, or some globbed expressions for elements of a probe tuple.
 */
static int
dtrace_match_priv(const dtrace_probe_t *prp, uint32_t priv, uid_t uid,
    zoneid_t zoneid)
{
        if (priv != DTRACE_PRIV_ALL) {
                uint32_t ppriv = prp->dtpr_provider->dtpv_priv.dtpp_flags;
                uint32_t match = priv & ppriv;

                /*
                 * No PRIV_DTRACE_* privileges...
                 */
                if ((priv & (DTRACE_PRIV_PROC | DTRACE_PRIV_USER |
                    DTRACE_PRIV_KERNEL)) == 0)
                        return (0);

                /*
                 * No matching bits, but there were bits to match...
                 */
                if (match == 0 && ppriv != 0)
                        return (0);

                /*
                 * Need to have permissions to the process, but don't...
                 */
                if (((ppriv & ~match) & DTRACE_PRIV_OWNER) != 0 &&
                    uid != prp->dtpr_provider->dtpv_priv.dtpp_uid) {
                        return (0);
                }

                /*
                 * Need to be in the same zone unless we possess the
                 * privilege to examine all zones.
                 */
                if (((ppriv & ~match) & DTRACE_PRIV_ZONEOWNER) != 0 &&
                    zoneid != prp->dtpr_provider->dtpv_priv.dtpp_zoneid) {
                        return (0);
                }
        }

        return (1);
}

/*
 * dtrace_match_probe compares a dtrace_probe_t to a pre-compiled key, which
 * consists of input pattern strings and an ops-vector to evaluate them.
 * This function returns >0 for match, 0 for no match, and <0 for error.
 */
static int
dtrace_match_probe(const dtrace_probe_t *prp, const dtrace_probekey_t *pkp,
    uint32_t priv, uid_t uid, zoneid_t zoneid)
{
        dtrace_provider_t *pvp = prp->dtpr_provider;
        int rv;

        if (pvp->dtpv_defunct)
                return (0);

        if ((rv = pkp->dtpk_pmatch(pvp->dtpv_name, pkp->dtpk_prov, 0)) <= 0)
                return (rv);

        if ((rv = pkp->dtpk_mmatch(prp->dtpr_mod, pkp->dtpk_mod, 0)) <= 0)
```



```
                return (rv);

        if ((rv = pkp->dtpk_fmatch(prp->dtpr_func, pkp->dtpk_func, 0)) <= 0)
                return (rv);

        if ((rv = pkp->dtpk_nmatch(prp->dtpr_name, pkp->dtpk_name, 0)) <= 0)
                return (rv);

        if (dtrace_match_priv(prp, priv, uid, zoneid) == 0)
                return (0);

        return (rv);
}

/*
 * dtrace_match_glob() is a safe kernel implementation of the gmatch(3GEN)
 * interface for matching a glob pattern 'p' to an input string 's'.  Unlike
 * libc's version, the kernel version only applies to 8-bit ASCII strings.
 * In addition, all of the recursion cases except for '*' matching have been
 * unwound.  For '*', we still implement recursive evaluation, but a depth
 * counter is maintained and matching is aborted if we recurse too deep.
 * The function returns 0 if no match, >0 if match, and <0 if recursion error.
 */
static int
dtrace_match_glob(const char *s, const char *p, int depth)
{
        const char *olds;
        char s1, c;
        int gs;

        if (depth > DTRACE_PROBEKEY_MAXDEPTH)
                return (-1);

        if (s == NULL)
                s = ""; /* treat NULL as empty string */

top:
        olds = s;
        s1 = *s++;

        if (p == NULL)
                return (0);

        if ((c = *p++) == '\0')
                return (s1 == '\0');

        switch (c) {
        case '[': {
                int ok = 0, notflag = 0;
                char lc = '\0';

                if (s1 == '\0')
                        return (0);

                if (*p == '!') {
                        notflag = 1;
                        p++;
                }

                if ((c = *p++) == '\0')
                        return (0);

                do {
                        if (c == '-' && lc != '\0' && *p != ']') {
                                if ((c = *p++) == '\0')
                                        return (0);
```

```
7058                                if (c == '\\' && (c = *p++) == '\0')
7059                                        return (0);
7060
7061                                if (notflag) {
7062                                        if (s1 < lc || s1 > c)
7063                                                ok++;
7064                                        else
7065                                                return (0);
7066                                } else if (lc <= s1 && s1 <= c)
7067                                        ok++;
7068
7069                        } else if (c == '\\' && (c = *p++) == '\0')
7070                                return (0);
7071
7072                        lc = c; /* save left-hand 'c' for next iteration */
7073
7074                        if (notflag) {
7075                                if (s1 != c)
7076                                        ok++;
7077                                else
7078                                        return (0);
7079                        } else if (s1 == c)
7080                                ok++;
7081
7082                        if ((c = *p++) == '\0')
7083                                return (0);
7084
7085                } while (c != ']');
7086
7087                if (ok)
7088                        goto top;
7089
7090                return (0);
7091        }
7092
7093        case '\\':
7094                if ((c = *p++) == '\0')
7095                        return (0);
7096                /*FALLTHRU*/
7097
7098        default:
7099                if (c != s1)
7100                        return (0);
7101                /*FALLTHRU*/
7102
7103        case '?':
7104                if (s1 != '\0')
7105                        goto top;
7106                return (0);
7107
7108        case '*':
7109                while (*p == '*')
7110                        p++; /* consecutive *'s are identical to a single one */
7111
7112                if (*p == '\0')
7113                        return (1);
7114
7115                for (s = olds; *s != '\0'; s++) {
7116                        if ((gs = dtrace_match_glob(s, p, depth + 1)) != 0)
7117                                return (gs);
7118                }
7119
7120                return (0);
7121        }
7122 }
```



```
7124 /*ARGSUSED*/
7125 static int
7126 dtrace_match_string(const char *s, const char *p, int depth)
7127 {
7128         return (s != NULL && strcmp(s, p) == 0);
7129 }
7130
7131 /*ARGSUSED*/
7132 static int
7133 dtrace_match_nul(const char *s, const char *p, int depth)
7134 {
7135         return (1); /* always match the empty pattern */
7136 }
7137
7138 /*ARGSUSED*/
7139 static int
7140 dtrace_match_nonzero(const char *s, const char *p, int depth)
7141 {
7142         return (s != NULL && s[0] != '\0');
7143 }
7144
7145 static int
7146 dtrace_match(const dtrace_probekey_t *pkp, uint32_t priv, uid_t uid,
7147     zoneid_t zoneid, int (*matched)(dtrace_probe_t *, void *), void *arg)
7148 {
7149         dtrace_probe_t template, *probe;
7150         dtrace_hash_t *hash = NULL;
7151         int len, rc, best = INT_MAX, nmatched = 0;
7152         dtrace_id_t i;
7153
7154         ASSERT(MUTEX_HELD(&dtrace_lock));
7155
7156         /*
7157          * If the probe ID is specified in the key, just lookup by ID and
7158          * invoke the match callback once if a matching probe is found.
7159          */
7160         if (pkp->dtpk_id != DTRACE_IDNONE) {
7161                 if ((probe = dtrace_probe_lookup_id(pkp->dtpk_id)) != NULL &&
7162                     dtrace_match_probe(probe, pkp, priv, uid, zoneid) > 0) {
7163                         if ((*matched)(probe, arg) == DTRACE_MATCH_FAIL)
7164                                 return (DTRACE_MATCH_FAIL);
7165                         nmatched++;
7166                 }
7167                 return (nmatched);
7168         }
7169
7170         template.dtpr_mod = (char *)pkp->dtpk_mod;
7171         template.dtpr_func = (char *)pkp->dtpk_func;
7172         template.dtpr_name = (char *)pkp->dtpk_name;
7173
7174         /*
7175          * We want to find the most distinct of the module name, function
7176          * name, and name.  So for each one that is not a glob pattern or
7177          * empty string, we perform a lookup in the corresponding hash and
7178          * use the hash table with the fewest collisions to do our search.
7179          */
7180         if (pkp->dtpk_mmatch == &dtrace_match_string &&
7181             (len = dtrace_hash_collisions(dtrace_bymod, &template)) < best) {
7182                 best = len;
7183                 hash = dtrace_bymod;
7184         }
7185
7186         if (pkp->dtpk_fmatch == &dtrace_match_string &&
7187             (len = dtrace_hash_collisions(dtrace_byfunc, &template)) < best) {
7188                 best = len;
7189                 hash = dtrace_byfunc;
```

```
7190         }

7192         if (pkp->dtpk_nmatch == &dtrace_match_string &&
7193             (len = dtrace_hash_collisions(dtrace_byname, &template)) < best) {
7194                 best = len;
7195                 hash = dtrace_byname;
7196         }

7198         /*
7199          * If we did not select a hash table, iterate over every probe and
7200          * invoke our callback for each one that matches our input probe key.
7201          */
7202         if (hash == NULL) {
7203                 for (i = 0; i < dtrace_nprobes; i++) {
7204                         if ((probe = dtrace_probes[i]) == NULL ||
7205                             dtrace_match_probe(probe, pkp, priv, uid,
7206                             zoneid) <= 0)
7207                                 continue;

7209                         nmatched++;

7211                         if ((rc = (*matched)(probe, arg)) !=
7212                             DTRACE_MATCH_NEXT) {
7213                                 if (rc == DTRACE_MATCH_FAIL)
7214                                         return (DTRACE_MATCH_FAIL);
7215                                 break;
7216                         }
7217                 }

7219                 return (nmatched);
7220         }

7222         /*
7223          * If we selected a hash table, iterate over each probe of the same key
7224          * name and invoke the callback for every probe that matches the other
7225          * attributes of our input probe key.
7226          */
7227         for (probe = dtrace_hash_lookup(hash, &template); probe != NULL;
7228             probe = *(DTRACE_HASHNEXT(hash, probe))) {

7230                 if (dtrace_match_probe(probe, pkp, priv, uid, zoneid) <= 0)
7231                         continue;

7233                 nmatched++;

7235                 if ((rc = (*matched)(probe, arg)) != DTRACE_MATCH_NEXT) {
7236                         if (rc == DTRACE_MATCH_FAIL)
7237                                 return (DTRACE_MATCH_FAIL);
7238                         break;
7239                 }
7240         }

7242         return (nmatched);
7243 }

7245 /*
7246  * Return the function pointer dtrace_probecmp() should use to compare the
7247  * specified pattern with a string.  For NULL or empty patterns, we select
7248  * dtrace_match_nul().  For glob pattern strings, we use dtrace_match_glob().
7249  * For non-empty non-glob strings, we use dtrace_match_string().
7250  */
7251 static dtrace_probekey_f *
7252 dtrace_probekey_func(const char *p)
7253 {
7254         char c;
```



```
7256         if (p == NULL || *p == '\0')
7257                 return (&dtrace_match_nul);

7259         while ((c = *p++) != '\0') {
7260                 if (c == '[' || c == '?' || c == '*' || c == '\\')
7261                         return (&dtrace_match_glob);
7262         }

7264         return (&dtrace_match_string);
7265 }

7267 /*
7268  * Build a probe comparison key for use with dtrace_match_probe() from the
7269  * given probe description.  By convention, a null key only matches anchored
7270  * probes: if each field is the empty string, reset dtpk_fmatch to
7271  * dtrace_match_nonzero().
7272  */
7273 static void
7274 dtrace_probekey(const dtrace_probedesc_t *pdp, dtrace_probekey_t *pkp)
7275 {
7276         pkp->dtpk_prov = pdp->dtpd_provider;
7277         pkp->dtpk_pmatch = dtrace_probekey_func(pdp->dtpd_provider);

7279         pkp->dtpk_mod = pdp->dtpd_mod;
7280         pkp->dtpk_mmatch = dtrace_probekey_func(pdp->dtpd_mod);

7282         pkp->dtpk_func = pdp->dtpd_func;
7283         pkp->dtpk_fmatch = dtrace_probekey_func(pdp->dtpd_func);

7285         pkp->dtpk_name = pdp->dtpd_name;
7286         pkp->dtpk_nmatch = dtrace_probekey_func(pdp->dtpd_name);

7288         pkp->dtpk_id = pdp->dtpd_id;

7290         if (pkp->dtpk_id == DTRACE_IDNONE &&
7291             pkp->dtpk_pmatch == &dtrace_match_nul &&
7292             pkp->dtpk_mmatch == &dtrace_match_nul &&
7293             pkp->dtpk_fmatch == &dtrace_match_nul &&
7294             pkp->dtpk_nmatch == &dtrace_match_nul)
7295                 pkp->dtpk_fmatch = &dtrace_match_nonzero;
7296 }

7298 /*
7299  * DTrace Provider-to-Framework API Functions
7300  *
7301  * These functions implement much of the Provider-to-Framework API, as
7302  * described in <sys/dtrace.h>.  The parts of the API not in this section are
7303  * the functions in the API for probe management (found below), and
7304  * dtrace_probe() itself (found above).
7305  */

7307 /*
7308  * Register the calling provider with the DTrace framework.  This should
7309  * generally be called by DTrace providers in their attach(9E) entry point.
7310  */
7311 int
7312 dtrace_register(const char *name, const dtrace_pattr_t *pap, uint32_t priv,
7313     cred_t *cr, const dtrace_pops_t *pops, void *arg, dtrace_provider_id_t *idp)
7314 {
7315         dtrace_provider_t *provider;

7317         if (name == NULL || pap == NULL || pops == NULL || idp == NULL) {
7318                 cmn_err(CE_WARN, "failed to register provider '%s': invalid "
7319                     "arguments", name ? name : "<NULL>");
7320                 return (EINVAL);
7321         }
```

```
7323         if (name[0] == '\0' || dtrace_badname(name)) {
7324                 cmn_err(CE_WARN, "failed to register provider '%s': invalid "
7325                     "provider name", name);
7326                 return (EINVAL);
7327         }

7329         if ((pops->dtps_provide == NULL && pops->dtps_provide_module == NULL) ||
7330             pops->dtps_enable == NULL || pops->dtps_disable == NULL ||
7331             pops->dtps_destroy == NULL ||
7332             ((pops->dtps_resume == NULL) != (pops->dtps_suspend == NULL))) {
7333                 cmn_err(CE_WARN, "failed to register provider '%s': invalid "
7334                     "provider ops", name);
7335                 return (EINVAL);
7336         }

7338         if (dtrace_badattr(&pap->dtpa_provider) ||
7339             dtrace_badattr(&pap->dtpa_mod) ||
7340             dtrace_badattr(&pap->dtpa_func) ||
7341             dtrace_badattr(&pap->dtpa_name) ||
7342             dtrace_badattr(&pap->dtpa_args)) {
7343                 cmn_err(CE_WARN, "failed to register provider '%s': invalid "
7344                     "provider attributes", name);
7345                 return (EINVAL);
7346         }

7348         if (priv & ~DTRACE_PRIV_ALL) {
7349                 cmn_err(CE_WARN, "failed to register provider '%s': invalid "
7350                     "privilege attributes", name);
7351                 return (EINVAL);
7352         }

7354         if ((priv & DTRACE_PRIV_KERNEL) &&
7355             (priv & (DTRACE_PRIV_USER | DTRACE_PRIV_OWNER)) &&
7356             pops->dtps_mode == NULL) {
7357                 cmn_err(CE_WARN, "failed to register provider '%s': need "
7358                     "dtps_mode() op for given privilege attributes", name);
7359                 return (EINVAL);
7360         }

7362         provider = kmem_zalloc(sizeof (dtrace_provider_t), KM_SLEEP);
7363         provider->dtpv_name = kmem_alloc(strlen(name) + 1, KM_SLEEP);
7364         (void) strcpy(provider->dtpv_name, name);

7366         provider->dtpv_attr = *pap;
7367         provider->dtpv_priv.dtpp_flags = priv;
7368         if (cr != NULL) {
7369                 provider->dtpv_priv.dtpp_uid = crgetuid(cr);
7370                 provider->dtpv_priv.dtpp_zoneid = crgetzoneid(cr);
7371         }
7372         provider->dtpv_pops = *pops;

7374         if (pops->dtps_provide == NULL) {
7375                 ASSERT(pops->dtps_provide_module != NULL);
7376                 provider->dtpv_pops.dtps_provide =
7377                     (void (*)(void *, const dtrace_probedesc_t *))dtrace_nullop;
7378         }

7380         if (pops->dtps_provide_module == NULL) {
7381                 ASSERT(pops->dtps_provide != NULL);
7382                 provider->dtpv_pops.dtps_provide_module =
7383                     (void (*)(void *, struct modctl *))dtrace_nullop;
7384         }

7386         if (pops->dtps_suspend == NULL) {
7387                 ASSERT(pops->dtps_resume == NULL);
```



```
7388                 provider->dtpv_pops.dtps_suspend =
7389                     (void (*)(void *, dtrace_id_t, void *))dtrace_nullop;
7390                 provider->dtpv_pops.dtps_resume =
7391                     (void (*)(void *, dtrace_id_t, void *))dtrace_nullop;
7392         }

7394         provider->dtpv_arg = arg;
7395         *idp = (dtrace_provider_id_t)provider;

7397         if (pops == &dtrace_provider_ops) {
7398                 ASSERT(MUTEX_HELD(&dtrace_provider_lock));
7399                 ASSERT(MUTEX_HELD(&dtrace_lock));
7400                 ASSERT(dtrace_anon.dta_enabling == NULL);

7402                 /*
7403                  * We make sure that the DTrace provider is at the head of
7404                  * the provider chain.
7405                  */
7406                 provider->dtpv_next = dtrace_provider;
7407                 dtrace_provider = provider;
7408                 return (0);
7409         }

7411         mutex_enter(&dtrace_provider_lock);
7412         mutex_enter(&dtrace_lock);

7414         /*
7415          * If there is at least one provider registered, we'll add this
7416          * provider after the first provider.
7417          */
7418         if (dtrace_provider != NULL) {
7419                 provider->dtpv_next = dtrace_provider->dtpv_next;
7420                 dtrace_provider->dtpv_next = provider;
7421         } else {
7422                 dtrace_provider = provider;
7423         }

7425         if (dtrace_retained != NULL) {
7426                 dtrace_enabling_provide(provider);

7428                 /*
7429                  * Now we need to call dtrace_enabling_matchall() -- which
7430                  * will acquire cpu_lock and dtrace_lock.  We therefore need
7431                  * to drop all of our locks before calling into it...
7432                  */
7433                 mutex_exit(&dtrace_lock);
7434                 mutex_exit(&dtrace_provider_lock);
7435                 dtrace_enabling_matchall();

7437                 return (0);
7438         }

7440         mutex_exit(&dtrace_lock);
7441         mutex_exit(&dtrace_provider_lock);

7443         return (0);
7444 }

7446 /*
7447  * Unregister the specified provider from the DTrace framework.  This should
7448  * generally be called by DTrace providers in their detach(9E) entry point.
7449  */
7450 int
7451 dtrace_unregister(dtrace_provider_id_t id)
7452 {
7453         dtrace_provider_t *old = (dtrace_provider_t *)id;
```

```
7454         dtrace_provider_t *prev = NULL;
7455         int i, self = 0, noreap = 0;
7456         dtrace_probe_t *probe, *first = NULL;

7458         if (old->dtpv_pops.dtps_enable ==
7459             (int (*)(void *, dtrace_id_t, void *))dtrace_enable_nullop) {
7460                 /*
7461                  * If DTrace itself is the provider, we're called with locks
7462                  * already held.
7463                  */
7464                 ASSERT(old == dtrace_provider);
7465                 ASSERT(dtrace_devi != NULL);
7466                 ASSERT(MUTEX_HELD(&dtrace_provider_lock));
7467                 ASSERT(MUTEX_HELD(&dtrace_lock));
7468                 self = 1;

7470                 if (dtrace_provider->dtpv_next != NULL) {
7471                         /*
7472                          * There's another provider here; return failure.
7473                          */
7474                         return (EBUSY);
7475                 }
7476         } else {
7477                 mutex_enter(&dtrace_provider_lock);
7478                 mutex_enter(&mod_lock);
7479                 mutex_enter(&dtrace_lock);
7480         }

7482         /*
7483          * If anyone has /dev/dtrace open, or if there are anonymous enabled
7484          * probes, we refuse to let providers slither away, unless this
7485          * provider has already been explicitly invalidated.
7486          */
7487         if (!old->dtpv_defunct &&
7488             (dtrace_opens || (dtrace_anon.dta_state != NULL &&
7489             dtrace_anon.dta_state->dts_necbs > 0))) {
7490                 if (!self) {
7491                         mutex_exit(&dtrace_lock);
7492                         mutex_exit(&mod_lock);
7493                         mutex_exit(&dtrace_provider_lock);
7494                 }
7495                 return (EBUSY);
7496         }

7498         /*
7499          * Attempt to destroy the probes associated with this provider.
7500          */
7501         for (i = 0; i < dtrace_nprobes; i++) {
7502                 if ((probe = dtrace_probes[i]) == NULL)
7503                         continue;

7505                 if (probe->dtpr_provider != old)
7506                         continue;

7508                 if (probe->dtpr_ecb == NULL)
7509                         continue;

7511                 /*
7512                  * If we are trying to unregister a defunct provider, and the
7513                  * provider was made defunct within the interval dictated by
7514                  * dtrace_unregister_defunct_reap, we'll (asynchronously)
7515                  * attempt to reap our enablings.  To denote that the provider
7516                  * should reattempt to unregister itself at some point in the
7517                  * future, we will return a differentiable error code (EAGAIN
7518                  * instead of EBUSY) in this case.
7519                  */
```



```
7520                 if (dtrace_gethrtime() - old->dtpv_defunct >
7521                     dtrace_unregister_defunct_reap)
7522                         noreap = 1;

7524                 if (!self) {
7525                         mutex_exit(&dtrace_lock);
7526                         mutex_exit(&mod_lock);
7527                         mutex_exit(&dtrace_provider_lock);
7528                 }

7530                 if (noreap)
7531                         return (EBUSY);

7533                 (void) taskq_dispatch(dtrace_taskq,
7534                     (task_func_t *)dtrace_enabling_reap, NULL, TQ_SLEEP);

7536                 return (EAGAIN);
7537         }

7539         /*
7540          * All of the probes for this provider are disabled; we can safely
7541          * remove all of them from their hash chains and from the probe array.
7542          */
7543         for (i = 0; i < dtrace_nprobes; i++) {
7544                 if ((probe = dtrace_probes[i]) == NULL)
7545                         continue;

7547                 if (probe->dtpr_provider != old)
7548                         continue;

7550                 dtrace_probes[i] = NULL;

7552                 dtrace_hash_remove(dtrace_bymod, probe);
7553                 dtrace_hash_remove(dtrace_byfunc, probe);
7554                 dtrace_hash_remove(dtrace_byname, probe);

7556                 if (first == NULL) {
7557                         first = probe;
7558                         probe->dtpr_nextmod = NULL;
7559                 } else {
7560                         probe->dtpr_nextmod = first;
7561                         first = probe;
7562                 }
7563         }

7565         /*
7566          * The provider's probes have been removed from the hash chains and
7567          * from the probe array.  Now issue a dtrace_sync() to be sure that
7568          * everyone has cleared out from any probe array processing.
7569          */
7570         dtrace_sync();

7572         for (probe = first; probe != NULL; probe = first) {
7573                 first = probe->dtpr_nextmod;

7575                 old->dtpv_pops.dtps_destroy(old->dtpv_arg, probe->dtpr_id,
7576                     probe->dtpr_arg);
7577                 kmem_free(probe->dtpr_mod, strlen(probe->dtpr_mod) + 1);
7578                 kmem_free(probe->dtpr_func, strlen(probe->dtpr_func) + 1);
7579                 kmem_free(probe->dtpr_name, strlen(probe->dtpr_name) + 1);
7580                 vmem_free(dtrace_arena, (void *)(uintptr_t)(probe->dtpr_id), 1);
7581                 kmem_free(probe, sizeof (dtrace_probe_t));
7582         }

7584         if ((prev = dtrace_provider) == old) {
7585                 ASSERT(self || dtrace_devi == NULL);
```

```
7586                 ASSERT(old->dtpv_next == NULL || dtrace_devi == NULL);
7587                 dtrace_provider = old->dtpv_next;
7588         } else {
7589                 while (prev != NULL && prev->dtpv_next != old)
7590                         prev = prev->dtpv_next;
7591
7592                 if (prev == NULL) {
7593                         panic("attempt to unregister non-existent "
7594                             "dtrace provider %p\n", (void *)id);
7595                 }
7596
7597                 prev->dtpv_next = old->dtpv_next;
7598         }
7599
7600         if (!self) {
7601                 mutex_exit(&dtrace_lock);
7602                 mutex_exit(&mod_lock);
7603                 mutex_exit(&dtrace_provider_lock);
7604         }
7605
7606         kmem_free(old->dtpv_name, strlen(old->dtpv_name) + 1);
7607         kmem_free(old, sizeof (dtrace_provider_t));
7608
7609         return (0);
7610 }
7611
7612 /*
7613  * Invalidate the specified provider.  All subsequent probe lookups for the
7614  * specified provider will fail, but its probes will not be removed.
7615  */
7616 void
7617 dtrace_invalidate(dtrace_provider_id_t id)
7618 {
7619         dtrace_provider_t *pvp = (dtrace_provider_t *)id;
7620
7621         ASSERT(pvp->dtpv_pops.dtps_enable !=
7622             (int (*)(void *, dtrace_id_t, void *))dtrace_enable_nullop);
7623
7624         mutex_enter(&dtrace_provider_lock);
7625         mutex_enter(&dtrace_lock);
7626
7627         pvp->dtpv_defunct = dtrace_gethrtime();
7628
7629         mutex_exit(&dtrace_lock);
7630         mutex_exit(&dtrace_provider_lock);
7631 }
7632
7633 /*
7634  * Indicate whether or not DTrace has attached.
7635  */
7636 int
7637 dtrace_attached(void)
7638 {
7639         /*
7640          * dtrace_provider will be non-NULL iff the DTrace driver has
7641          * attached.  (It's non-NULL because DTrace is always itself a
7642          * provider.)
7643          */
7644         return (dtrace_provider != NULL);
7645 }
7646
7647 /*
7648  * Remove all the unenabled probes for the given provider.  This function is
7649  * not unlike dtrace_unregister(), except that it doesn't remove the provider
7650  * -- just as many of its associated probes as it can.
7651  */
```



```
7652 int
7653 dtrace_condense(dtrace_provider_id_t id)
7654 {
7655         dtrace_provider_t *prov = (dtrace_provider_t *)id;
7656         int i;
7657         dtrace_probe_t *probe;
7658
7659         /*
7660          * Make sure this isn't the dtrace provider itself.
7661          */
7662         ASSERT(prov->dtpv_pops.dtps_enable !=
7663             (int (*)(void *, dtrace_id_t, void *))dtrace_enable_nullop);
7664
7665         mutex_enter(&dtrace_provider_lock);
7666         mutex_enter(&dtrace_lock);
7667
7668         /*
7669          * Attempt to destroy the probes associated with this provider.
7670          */
7671         for (i = 0; i < dtrace_nprobes; i++) {
7672                 if ((probe = dtrace_probes[i]) == NULL)
7673                         continue;
7674
7675                 if (probe->dtpr_provider != prov)
7676                         continue;
7677
7678                 if (probe->dtpr_ecb != NULL)
7679                         continue;
7680
7681                 dtrace_probes[i] = NULL;
7682
7683                 dtrace_hash_remove(dtrace_bymod, probe);
7684                 dtrace_hash_remove(dtrace_byfunc, probe);
7685                 dtrace_hash_remove(dtrace_byname, probe);
7686
7687                 prov->dtpv_pops.dtps_destroy(prov->dtpv_arg, i + 1,
7688                     probe->dtpr_arg);
7689                 kmem_free(probe->dtpr_mod, strlen(probe->dtpr_mod) + 1);
7690                 kmem_free(probe->dtpr_func, strlen(probe->dtpr_func) + 1);
7691                 kmem_free(probe->dtpr_name, strlen(probe->dtpr_name) + 1);
7692                 kmem_free(probe, sizeof (dtrace_probe_t));
7693                 vmem_free(dtrace_arena, (void *)((uintptr_t)i + 1), 1);
7694         }
7695
7696         mutex_exit(&dtrace_lock);
7697         mutex_exit(&dtrace_provider_lock);
7698
7699         return (0);
7700 }
7701
7702 /*
7703  * DTrace Probe Management Functions
7704  *
7705  * The functions in this section perform the DTrace probe management,
7706  * including functions to create probes, look-up probes, and call into the
7707  * providers to request that probes be provided.  Some of these functions are
7708  * in the Provider-to-Framework API; these functions can be identified by the
7709  * fact that they are not declared "static".
7710  */
7711
7712 /*
7713  * Create a probe with the specified module name, function name, and name.
7714  */
7715 dtrace_id_t
7716 dtrace_probe_create(dtrace_provider_id_t prov, const char *mod,
7717     const char *func, const char *name, int aframes, void *arg)
```

```
7718 {
7719         dtrace_probe_t *probe, **probes;
7720         dtrace_provider_t *provider = (dtrace_provider_t *)prov;
7721         dtrace_id_t id;
7722
7723         if (provider == dtrace_provider) {
7724                 ASSERT(MUTEX_HELD(&dtrace_lock));
7725         } else {
7726                 mutex_enter(&dtrace_lock);
7727         }
7728
7729         id = (dtrace_id_t)(uintptr_t)vmem_alloc(dtrace_arena, 1,
7730             VM_BESTFIT | VM_SLEEP);
7731         probe = kmem_zalloc(sizeof (dtrace_probe_t), KM_SLEEP);
7732
7733         probe->dtpr_id = id;
7734         probe->dtpr_gen = dtrace_probegen++;
7735         probe->dtpr_mod = dtrace_strdup(mod);
7736         probe->dtpr_func = dtrace_strdup(func);
7737         probe->dtpr_name = dtrace_strdup(name);
7738         probe->dtpr_arg = arg;
7739         probe->dtpr_aframes = aframes;
7740         probe->dtpr_provider = provider;
7741
7742         dtrace_hash_add(dtrace_bymod, probe);
7743         dtrace_hash_add(dtrace_byfunc, probe);
7744         dtrace_hash_add(dtrace_byname, probe);
7745
7746         if (id - 1 >= dtrace_nprobes) {
7747                 size_t osize = dtrace_nprobes * sizeof (dtrace_probe_t *);
7748                 size_t nsize = osize << 1;
7749
7750                 if (nsize == 0) {
7751                         ASSERT(osize == 0);
7752                         ASSERT(dtrace_probes == NULL);
7753                         nsize = sizeof (dtrace_probe_t *);
7754                 }
7755
7756                 probes = kmem_zalloc(nsize, KM_SLEEP);
7757
7758                 if (dtrace_probes == NULL) {
7759                         ASSERT(osize == 0);
7760                         dtrace_probes = probes;
7761                         dtrace_nprobes = 1;
7762                 } else {
7763                         dtrace_probe_t **oprobes = dtrace_probes;
7764
7765                         bcopy(oprobes, probes, osize);
7766                         dtrace_membar_producer();
7767                         dtrace_probes = probes;
7768
7769                         dtrace_sync();
7770
7771                         /*
7772                          * All CPUs are now seeing the new probes array; we can
7773                          * safely free the old array.
7774                          */
7775                         kmem_free(oprobes, osize);
7776                         dtrace_nprobes <<= 1;
7777                 }
7778
7779                 ASSERT(id - 1 < dtrace_nprobes);
7780         }
7781
7782         ASSERT(dtrace_probes[id - 1] == NULL);
7783         dtrace_probes[id - 1] = probe;
```



```
7785         if (provider != dtrace_provider)
7786                 mutex_exit(&dtrace_lock);
7787
7788         return (id);
7789 }
7790
7791 static dtrace_probe_t *
7792 dtrace_probe_lookup_id(dtrace_id_t id)
7793 {
7794         ASSERT(MUTEX_HELD(&dtrace_lock));
7795
7796         if (id == 0 || id > dtrace_nprobes)
7797                 return (NULL);
7798
7799         return (dtrace_probes[id - 1]);
7800 }
7801
7802 static int
7803 dtrace_probe_lookup_match(dtrace_probe_t *probe, void *arg)
7804 {
7805         *((dtrace_id_t *)arg) = probe->dtpr_id;
7806
7807         return (DTRACE_MATCH_DONE);
7808 }
7809
7810 /*
7811  * Look up a probe based on provider and one or more of module name, function
7812  * name and probe name.
7813  */
7814 dtrace_id_t
7815 dtrace_probe_lookup(dtrace_provider_id_t prid, const char *mod,
7816     const char *func, const char *name)
7817 {
7818         dtrace_probekey_t pkey;
7819         dtrace_id_t id;
7820         int match;
7821
7822         pkey.dtpk_prov = ((dtrace_provider_t *)prid)->dtpv_name;
7823         pkey.dtpk_pmatch = &dtrace_match_string;
7824         pkey.dtpk_mod = mod;
7825         pkey.dtpk_mmatch = mod ? &dtrace_match_string : &dtrace_match_nul;
7826         pkey.dtpk_func = func;
7827         pkey.dtpk_fmatch = func ? &dtrace_match_string : &dtrace_match_nul;
7828         pkey.dtpk_name = name;
7829         pkey.dtpk_nmatch = name ? &dtrace_match_string : &dtrace_match_nul;
7830         pkey.dtpk_id = DTRACE_IDNONE;
7831
7832         mutex_enter(&dtrace_lock);
7833         match = dtrace_match(&pkey, DTRACE_PRIV_ALL, 0, 0,
7834             dtrace_probe_lookup_match, &id);
7835         mutex_exit(&dtrace_lock);
7836
7837         ASSERT(match == 1 || match == 0);
7838         return (match ? id : 0);
7839 }
7840
7841 /*
7842  * Returns the probe argument associated with the specified probe.
7843  */
7844 void *
7845 dtrace_probe_arg(dtrace_provider_id_t id, dtrace_id_t pid)
7846 {
7847         dtrace_probe_t *probe;
7848         void *rval = NULL;
```

```
7850         mutex_enter(&dtrace_lock);
7851
7852         if ((probe = dtrace_probe_lookup_id(pid)) != NULL &&
7853             probe->dtpr_provider == (dtrace_provider_t *)id)
7854                 rval = probe->dtpr_arg;
7855
7856         mutex_exit(&dtrace_lock);
7857
7858         return (rval);
7859 }
7860
7861 /*
7862  * Copy a probe into a probe description.
7863  */
7864 static void
7865 dtrace_probe_description(const dtrace_probe_t *prp, dtrace_probedesc_t *pdp)
7866 {
7867         bzero(pdp, sizeof (dtrace_probedesc_t));
7868         pdp->dtpd_id = prp->dtpr_id;
7869
7870         (void) strncpy(pdp->dtpd_provider,
7871             prp->dtpr_provider->dtpv_name, DTRACE_PROVNAMELEN - 1);
7872
7873         (void) strncpy(pdp->dtpd_mod, prp->dtpr_mod, DTRACE_MODNAMELEN - 1);
7874         (void) strncpy(pdp->dtpd_func, prp->dtpr_func, DTRACE_FUNCNAMELEN - 1);
7875         (void) strncpy(pdp->dtpd_name, prp->dtpr_name, DTRACE_NAMELEN - 1);
7876 }
7877
7878 /*
7879  * Called to indicate that a probe -- or probes -- should be provided by a
7880  * specfied provider.  If the specified description is NULL, the provider will
7881  * be told to provide all of its probes.  (This is done whenever a new
7882  * consumer comes along, or whenever a retained enabling is to be matched.) If
7883  * the specified description is non-NULL, the provider is given the
7884  * opportunity to dynamically provide the specified probe, allowing providers
7885  * to support the creation of probes on-the-fly.  (So-called _autocreated_
7886  * probes.)  If the provider is NULL, the operations will be applied to all
7887  * providers; if the provider is non-NULL the operations will only be applied
7888  * to the specified provider.  The dtrace_provider_lock must be held, and the
7889  * dtrace_lock must _not_ be held -- the provider's dtps_provide() operation
7890  * will need to grab the dtrace_lock when it reenters the framework through
7891  * dtrace_probe_lookup(), dtrace_probe_create(), etc.
7892  */
7893 static void
7894 dtrace_probe_provide(dtrace_probedesc_t *desc, dtrace_provider_t *prv)
7895 {
7896         struct modctl *ctl;
7897         int all = 0;
7898
7899         ASSERT(MUTEX_HELD(&dtrace_provider_lock));
7900
7901         if (prv == NULL) {
7902                 all = 1;
7903                 prv = dtrace_provider;
7904         }
7905
7906         do {
7907                 /*
7908                  * First, call the blanket provide operation.
7909                  */
7910                 prv->dtpv_pops.dtps_provide(prv->dtpv_arg, desc);
7911
7912                 /*
7913                  * Now call the per-module provide operation.  We will grab
7914                  * mod_lock to prevent the list from being modified.  Note
7915                  * that this also prevents the mod_busy bits from changing.
```



```
7916                  * (mod_busy can only be changed with mod_lock held.)
7917                  */
7918                 mutex_enter(&mod_lock);
7919
7920                 ctl = &modules;
7921                 do {
7922                         if (ctl->mod_busy || ctl->mod_mp == NULL)
7923                                 continue;
7924
7925                         prv->dtpv_pops.dtps_provide_module(prv->dtpv_arg, ctl);
7926
7927                 } while ((ctl = ctl->mod_next) != &modules);
7928
7929                 mutex_exit(&mod_lock);
7930         } while (all && (prv = prv->dtpv_next) != NULL);
7931 }
7932
7933 /*
7934  * Iterate over each probe, and call the Framework-to-Provider API function
7935  * denoted by offs.
7936  */
7937 static void
7938 dtrace_probe_foreach(uintptr_t offs)
7939 {
7940         dtrace_provider_t *prov;
7941         void (*func)(void *, dtrace_id_t, void *);
7942         dtrace_probe_t *probe;
7943         dtrace_icookie_t cookie;
7944         int i;
7945
7946         /*
7947          * We disable interrupts to walk through the probe array.  This is
7948          * safe -- the dtrace_sync() in dtrace_unregister() assures that we
7949          * won't see stale data.
7950          */
7951         cookie = dtrace_interrupt_disable();
7952
7953         for (i = 0; i < dtrace_nprobes; i++) {
7954                 if ((probe = dtrace_probes[i]) == NULL)
7955                         continue;
7956
7957                 if (probe->dtpr_ecb == NULL) {
7958                         /*
7959                          * This probe isn't enabled -- don't call the function.
7960                          */
7961                         continue;
7962                 }
7963
7964                 prov = probe->dtpr_provider;
7965                 func = *((void(**)(void *, dtrace_id_t, void *))
7966                     ((uintptr_t)&prov->dtpv_pops + offs));
7967
7968                 func(prov->dtpv_arg, i + 1, probe->dtpr_arg);
7969         }
7970
7971         dtrace_interrupt_enable(cookie);
7972 }
7973
7974 static int
7975 dtrace_probe_enable(const dtrace_probedesc_t *desc, dtrace_enabling_t *enab)
7976 {
7977         dtrace_probekey_t pkey;
7978         uint32_t priv;
7979         uid_t uid;
7980         zoneid_t zoneid;
```

```
7982         ASSERT(MUTEX_HELD(&dtrace_lock));
7983         dtrace_ecb_create_cache = NULL;
7984
7985         if (desc == NULL) {
7986                 /*
7987                  * If we're passed a NULL description, we're being asked to
7988                  * create an ECB with a NULL probe.
7989                  */
7990                 (void) dtrace_ecb_create_enable(NULL, enab);
7991                 return (0);
7992         }
7993
7994         dtrace_probekey(desc, &pkey);
7995         dtrace_cred2priv(enab->dten_vstate->dtvs_state->dts_cred.dcr_cred,
7996             &priv, &uid, &zoneid);
7997
7998         return (dtrace_match(&pkey, priv, uid, zoneid, dtrace_ecb_create_enable,
7999             enab));
8000 }
8001
8002 /*
8003  * DTrace Helper Provider Functions
8004  */
8005 static void
8006 dtrace_dofattr2attr(dtrace_attribute_t *attr, const dof_attr_t dofattr)
8007 {
8008         attr->dtat_name = DOF_ATTR_NAME(dofattr);
8009         attr->dtat_data = DOF_ATTR_DATA(dofattr);
8010         attr->dtat_class = DOF_ATTR_CLASS(dofattr);
8011 }
8012
8013 static void
8014 dtrace_dofprov2hprov(dtrace_helper_provdesc_t *hprov,
8015     const dof_provider_t *dofprov, char *strtab)
8016 {
8017         hprov->dthpv_provname = strtab + dofprov->dofpv_name;
8018         dtrace_dofattr2attr(&hprov->dthpv_pattr.dtpa_provider,
8019             dofprov->dofpv_provattr);
8020         dtrace_dofattr2attr(&hprov->dthpv_pattr.dtpa_mod,
8021             dofprov->dofpv_modattr);
8022         dtrace_dofattr2attr(&hprov->dthpv_pattr.dtpa_func,
8023             dofprov->dofpv_funcattr);
8024         dtrace_dofattr2attr(&hprov->dthpv_pattr.dtpa_name,
8025             dofprov->dofpv_nameattr);
8026         dtrace_dofattr2attr(&hprov->dthpv_pattr.dtpa_args,
8027             dofprov->dofpv_argsattr);
8028 }
8029
8030 static void
8031 dtrace_helper_provide_one(dof_helper_t *dhp, dof_sec_t *sec, pid_t pid)
8032 {
8033         uintptr_t daddr = (uintptr_t)dhp->dofhp_dof;
8034         dof_hdr_t *dof = (dof_hdr_t *)daddr;
8035         dof_sec_t *str_sec, *prb_sec, *arg_sec, *off_sec, *enoff_sec;
8036         dof_provider_t *provider;
8037         dof_probe_t *probe;
8038         uint32_t *off, *enoff;
8039         uint8_t *arg;
8040         char *strtab;
8041         uint_t i, nprobes;
8042         dtrace_helper_provdesc_t dhpv;
8043         dtrace_helper_probedesc_t dhpb;
8044         dtrace_meta_t *meta = dtrace_meta_pid;
8045         dtrace_mops_t *mops = &meta->dtm_mops;
8046         void *parg;
```



```
8048         provider = (dof_provider_t *)(uintptr_t)(daddr + sec->dofs_offset);
8049         str_sec = (dof_sec_t *)(uintptr_t)(daddr + dof->dofh_secoff +
8050             provider->dofpv_strtab * dof->dofh_secsize);
8051         prb_sec = (dof_sec_t *)(uintptr_t)(daddr + dof->dofh_secoff +
8052             provider->dofpv_probes * dof->dofh_secsize);
8053         arg_sec = (dof_sec_t *)(uintptr_t)(daddr + dof->dofh_secoff +
8054             provider->dofpv_prargs * dof->dofh_secsize);
8055         off_sec = (dof_sec_t *)(uintptr_t)(daddr + dof->dofh_secoff +
8056             provider->dofpv_proffs * dof->dofh_secsize);
8057
8058         strtab = (char *)(uintptr_t)(daddr + str_sec->dofs_offset);
8059         off = (uint32_t *)(uintptr_t)(daddr + off_sec->dofs_offset);
8060         arg = (uint8_t *)(uintptr_t)(daddr + arg_sec->dofs_offset);
8061         enoff = NULL;
8062
8063         /*
8064          * See dtrace_helper_provider_validate().
8065          */
8066         if (dof->dofh_ident[DOF_ID_VERSION] != DOF_VERSION_1 &&
8067             provider->dofpv_prenoffs != DOF_SECT_NONE) {
8068                 enoff_sec = (dof_sec_t *)(uintptr_t)(daddr + dof->dofh_secoff +
8069                     provider->dofpv_prenoffs * dof->dofh_secsize);
8070                 enoff = (uint32_t *)(uintptr_t)(daddr + enoff_sec->dofs_offset);
8071         }
8072
8073         nprobes = prb_sec->dofs_size / prb_sec->dofs_entsize;
8074
8075         /*
8076          * Create the provider.
8077          */
8078         dtrace_dofprov2hprov(&dhpv, provider, strtab);
8079
8080         if ((parg = mops->dtms_provide_pid(meta->dtm_arg, &dhpv, pid)) == NULL)
8081                 return;
8082
8083         meta->dtm_count++;
8084
8085         /*
8086          * Create the probes.
8087          */
8088         for (i = 0; i < nprobes; i++) {
8089                 probe = (dof_probe_t *)(uintptr_t)(daddr +
8090                     prb_sec->dofs_offset + i * prb_sec->dofs_entsize);
8091
8092                 dhpb.dthpb_mod = dhp->dofhp_mod;
8093                 dhpb.dthpb_func = strtab + probe->dofpr_func;
8094                 dhpb.dthpb_name = strtab + probe->dofpr_name;
8095                 dhpb.dthpb_base = probe->dofpr_addr;
8096                 dhpb.dthpb_offs = off + probe->dofpr_offidx;
8097                 dhpb.dthpb_noffs = probe->dofpr_noffs;
8098                 if (enoff != NULL) {
8099                         dhpb.dthpb_enoffs = enoff + probe->dofpr_enoffidx;
8100                         dhpb.dthpb_nenoffs = probe->dofpr_nenoffs;
8101                 } else {
8102                         dhpb.dthpb_enoffs = NULL;
8103                         dhpb.dthpb_nenoffs = 0;
8104                 }
8105                 dhpb.dthpb_args = arg + probe->dofpr_argidx;
8106                 dhpb.dthpb_nargc = probe->dofpr_nargc;
8107                 dhpb.dthpb_xargc = probe->dofpr_xargc;
8108                 dhpb.dthpb_ntypes = strtab + probe->dofpr_nargv;
8109                 dhpb.dthpb_xtypes = strtab + probe->dofpr_xargv;
8110
8111                 mops->dtms_create_probe(meta->dtm_arg, parg, &dhpb);
8112         }
8113 }
```

```
8115 static void
8116 dtrace_helper_provide(dof_helper_t *dhp, pid_t pid)
8117 {
8118         uintptr_t daddr = (uintptr_t)dhp->dofhp_dof;
8119         dof_hdr_t *dof = (dof_hdr_t *)daddr;
8120         int i;

8122         ASSERT(MUTEX_HELD(&dtrace_meta_lock));

8124         for (i = 0; i < dof->dofh_secnum; i++) {
8125                 dof_sec_t *sec = (dof_sec_t *)(uintptr_t)(daddr +
8126                     dof->dofh_secoff + i * dof->dofh_secsize);

8128                 if (sec->dofs_type != DOF_SECT_PROVIDER)
8129                         continue;

8131                 dtrace_helper_provide_one(dhp, sec, pid);
8132         }

8134         /*
8135          * We may have just created probes, so we must now rematch against
8136          * any retained enablings.  Note that this call will acquire both
8137          * cpu_lock and dtrace_lock; the fact that we are holding
8138          * dtrace_meta_lock now is what defines the ordering with respect to
8139          * these three locks.
8140          */
8141         dtrace_enabling_matchall();
8142 }

8144 static void
8145 dtrace_helper_provider_remove_one(dof_helper_t *dhp, dof_sec_t *sec, pid_t pid)
8146 {
8147         uintptr_t daddr = (uintptr_t)dhp->dofhp_dof;
8148         dof_hdr_t *dof = (dof_hdr_t *)daddr;
8149         dof_sec_t *str_sec;
8150         dof_provider_t *provider;
8151         char *strtab;
8152         dtrace_helper_provdesc_t dhpv;
8153         dtrace_meta_t *meta = dtrace_meta_pid;
8154         dtrace_mops_t *mops = &meta->dtm_mops;

8156         provider = (dof_provider_t *)(uintptr_t)(daddr + sec->dofs_offset);
8157         str_sec = (dof_sec_t *)(uintptr_t)(daddr + dof->dofh_secoff +
8158             provider->dofpv_strtab * dof->dofh_secsize);

8160         strtab = (char *)(uintptr_t)(daddr + str_sec->dofs_offset);

8162         /*
8163          * Create the provider.
8164          */
8165         dtrace_dofprov2hprov(&dhpv, provider, strtab);

8167         mops->dtms_remove_pid(meta->dtm_arg, &dhpv, pid);

8169         meta->dtm_count--;
8170 }

8172 static void
8173 dtrace_helper_provider_remove(dof_helper_t *dhp, pid_t pid)
8174 {
8175         uintptr_t daddr = (uintptr_t)dhp->dofhp_dof;
8176         dof_hdr_t *dof = (dof_hdr_t *)daddr;
8177         int i;

8179         ASSERT(MUTEX_HELD(&dtrace_meta_lock));
```



```
8181         for (i = 0; i < dof->dofh_secnum; i++) {
8182                 dof_sec_t *sec = (dof_sec_t *)(uintptr_t)(daddr +
8183                     dof->dofh_secoff + i * dof->dofh_secsize);

8185                 if (sec->dofs_type != DOF_SECT_PROVIDER)
8186                         continue;

8188                 dtrace_helper_provider_remove_one(dhp, sec, pid);
8189         }
8190 }

8192 /*
8193  * DTrace Meta Provider-to-Framework API Functions
8194  *
8195  * These functions implement the Meta Provider-to-Framework API, as described
8196  * in <sys/dtrace.h>.
8197  */
8198 int
8199 dtrace_meta_register(const char *name, const dtrace_mops_t *mops, void *arg,
8200     dtrace_meta_provider_id_t *idp)
8201 {
8202         dtrace_meta_t *meta;
8203         dtrace_helpers_t *help, *next;
8204         int i;

8206         *idp = DTRACE_METAPROVNONE;

8208         /*
8209          * We strictly don't need the name, but we hold onto it for
8210          * debuggability. All hail error queues!
8211          */
8212         if (name == NULL) {
8213                 cmn_err(CE_WARN, "failed to register meta-provider: "
8214                     "invalid name");
8215                 return (EINVAL);
8216         }

8218         if (mops == NULL ||
8219             mops->dtms_create_probe == NULL ||
8220             mops->dtms_provide_pid == NULL ||
8221             mops->dtms_remove_pid == NULL) {
8222                 cmn_err(CE_WARN, "failed to register meta-register %s: "
8223                     "invalid ops", name);
8224                 return (EINVAL);
8225         }

8227         meta = kmem_zalloc(sizeof (dtrace_meta_t), KM_SLEEP);
8228         meta->dtm_mops = *mops;
8229         meta->dtm_name = kmem_alloc(strlen(name) + 1, KM_SLEEP);
8230         (void) strcpy(meta->dtm_name, name);
8231         meta->dtm_arg = arg;

8233         mutex_enter(&dtrace_meta_lock);
8234         mutex_enter(&dtrace_lock);

8236         if (dtrace_meta_pid != NULL) {
8237                 mutex_exit(&dtrace_lock);
8238                 mutex_exit(&dtrace_meta_lock);
8239                 cmn_err(CE_WARN, "failed to register meta-register %s: "
8240                     "user-land meta-provider exists", name);
8241                 kmem_free(meta->dtm_name, strlen(meta->dtm_name) + 1);
8242                 kmem_free(meta, sizeof (dtrace_meta_t));
8243                 return (EINVAL);
8244         }
```

```
8246         dtrace_meta_pid = meta;
8247         *idp = (dtrace_meta_provider_id_t)meta;

8249         /*
8250          * If there are providers and probes ready to go, pass them
8251          * off to the new meta provider now.
8252          */

8254         help = dtrace_deferred_pid;
8255         dtrace_deferred_pid = NULL;

8257         mutex_exit(&dtrace_lock);

8259         while (help != NULL) {
8260                 for (i = 0; i < help->dthps_nprovs; i++) {
8261                         dtrace_helper_provide(&help->dthps_provs[i]->dthp_prov,
8262                             help->dthps_pid);
8263                 }

8265                 next = help->dthps_next;
8266                 help->dthps_next = NULL;
8267                 help->dthps_prev = NULL;
8268                 help->dthps_deferred = 0;
8269                 help = next;
8270         }

8272         mutex_exit(&dtrace_meta_lock);

8274         return (0);
8275 }

8277 int
8278 dtrace_meta_unregister(dtrace_meta_provider_id_t id)
8279 {
8280         dtrace_meta_t **pp, *old = (dtrace_meta_t *)id;

8282         mutex_enter(&dtrace_meta_lock);
8283         mutex_enter(&dtrace_lock);

8285         if (old == dtrace_meta_pid) {
8286                 pp = &dtrace_meta_pid;
8287         } else {
8288                 panic("attempt to unregister non-existent "
8289                     "dtrace meta-provider %p\n", (void *)old);
8290         }

8292         if (old->dtm_count != 0) {
8293                 mutex_exit(&dtrace_lock);
8294                 mutex_exit(&dtrace_meta_lock);
8295                 return (EBUSY);
8296         }

8298         *pp = NULL;

8300         mutex_exit(&dtrace_lock);
8301         mutex_exit(&dtrace_meta_lock);

8303         kmem_free(old->dtm_name, strlen(old->dtm_name) + 1);
8304         kmem_free(old, sizeof (dtrace_meta_t));

8306         return (0);
8307 }


8310 /*
8311  * DTrace DIF Object Functions
```



```
8312  */
8313 static int
8314 dtrace_difo_err(uint_t pc, const char *format, ...)
8315 {
8316         if (dtrace_err_verbose) {
8317                 va_list alist;

8319                 (void) uprintf("dtrace DIF object error: [%u]: ", pc);
8320                 va_start(alist, format);
8321                 (void) vuprintf(format, alist);
8322                 va_end(alist);
8323         }

8325 #ifdef DTRACE_ERRDEBUG
8326         dtrace_errdebug(format);
8327 #endif
8328         return (1);
8329 }

8331 /*
8332  * Validate a DTrace DIF object by checking the IR instructions.  The following
8333  * rules are currently enforced by dtrace_difo_validate():
8334  *
8335  * 1. Each instruction must have a valid opcode
8336  * 2. Each register, string, variable, or subroutine reference must be valid
8337  * 3. No instruction can modify register %r0 (must be zero)
8338  * 4. All instruction reserved bits must be set to zero
8339  * 5. The last instruction must be a "ret" instruction
8340  * 6. All branch targets must reference a valid instruction _after_ the branch
8341  */
8342 static int
8343 dtrace_difo_validate(dtrace_difo_t *dp, dtrace_vstate_t *vstate, uint_t nregs,
8344     cred_t *cr)
8345 {
8346         int err = 0, i;
8347         int (*efunc)(uint_t pc, const char *, ...) = dtrace_difo_err;
8348         int kcheckload;
8349         uint_t pc;

8351         kcheckload = cr == NULL ||
8352             (vstate->dtvs_state->dts_cred.dcr_visible & DTRACE_CRV_KERNEL) == 0;

8354         dp->dtdo_destructive = 0;

8356         for (pc = 0; pc < dp->dtdo_len && err == 0; pc++) {
8357                 dif_instr_t instr = dp->dtdo_buf[pc];

8359                 uint_t r1 = DIF_INSTR_R1(instr);
8360                 uint_t r2 = DIF_INSTR_R2(instr);
8361                 uint_t rd = DIF_INSTR_RD(instr);
8362                 uint_t rs = DIF_INSTR_RS(instr);
8363                 uint_t label = DIF_INSTR_LABEL(instr);
8364                 uint_t v = DIF_INSTR_VAR(instr);
8365                 uint_t subr = DIF_INSTR_SUBR(instr);
8366                 uint_t type = DIF_INSTR_TYPE(instr);
8367                 uint_t op = DIF_INSTR_OP(instr);

8369                 switch (op) {
8370                 case DIF_OP_OR:
8371                 case DIF_OP_XOR:
8372                 case DIF_OP_AND:
8373                 case DIF_OP_SLL:
8374                 case DIF_OP_SRL:
8375                 case DIF_OP_SRA:
8376                 case DIF_OP_SUB:
8377                 case DIF_OP_ADD:
```

```
8378                case DIF_OP_MUL:
8379                case DIF_OP_SDIV:
8380                case DIF_OP_UDIV:
8381                case DIF_OP_SREM:
8382                case DIF_OP_UREM:
8383                case DIF_OP_COPYS:
8384                        if (r1 >= nregs)
8385                                err += efunc(pc, "invalid register %u\n", r1);
8386                        if (r2 >= nregs)
8387                                err += efunc(pc, "invalid register %u\n", r2);
8388                        if (rd >= nregs)
8389                                err += efunc(pc, "invalid register %u\n", rd);
8390                        if (rd == 0)
8391                                err += efunc(pc, "cannot write to %r0\n");
8392                        break;
8393                case DIF_OP_NOT:
8394                case DIF_OP_MOV:
8395                case DIF_OP_ALLOCS:
8396                        if (r1 >= nregs)
8397                                err += efunc(pc, "invalid register %u\n", r1);
8398                        if (r2 != 0)
8399                                err += efunc(pc, "non-zero reserved bits\n");
8400                        if (rd >= nregs)
8401                                err += efunc(pc, "invalid register %u\n", rd);
8402                        if (rd == 0)
8403                                err += efunc(pc, "cannot write to %r0\n");
8404                        break;
8405                case DIF_OP_LDSB:
8406                case DIF_OP_LDSH:
8407                case DIF_OP_LDSW:
8408                case DIF_OP_LDUB:
8409                case DIF_OP_LDUH:
8410                case DIF_OP_LDUW:
8411                case DIF_OP_LDX:
8412                        if (r1 >= nregs)
8413                                err += efunc(pc, "invalid register %u\n", r1);
8414                        if (r2 != 0)
8415                                err += efunc(pc, "non-zero reserved bits\n");
8416                        if (rd >= nregs)
8417                                err += efunc(pc, "invalid register %u\n", rd);
8418                        if (rd == 0)
8419                                err += efunc(pc, "cannot write to %r0\n");
8420                        if (kcheckload)
8421                                dp->dtdo_buf[pc] = DIF_INSTR_LOAD(op +
8422                                    DIF_OP_RLDSB - DIF_OP_LDSB, r1, rd);
8423                        break;
8424                case DIF_OP_RLDSB:
8425                case DIF_OP_RLDSH:
8426                case DIF_OP_RLDSW:
8427                case DIF_OP_RLDUB:
8428                case DIF_OP_RLDUH:
8429                case DIF_OP_RLDUW:
8430                case DIF_OP_RLDX:
8431                        if (r1 >= nregs)
8432                                err += efunc(pc, "invalid register %u\n", r1);
8433                        if (r2 != 0)
8434                                err += efunc(pc, "non-zero reserved bits\n");
8435                        if (rd >= nregs)
8436                                err += efunc(pc, "invalid register %u\n", rd);
8437                        if (rd == 0)
8438                                err += efunc(pc, "cannot write to %r0\n");
8439                        break;
8440                case DIF_OP_ULDSB:
8441                case DIF_OP_ULDSH:
8442                case DIF_OP_ULDSW:
8443                case DIF_OP_ULDUB:
```



```
8444                case DIF_OP_ULDUH:
8445                case DIF_OP_ULDUW:
8446                case DIF_OP_ULDX:
8447                        if (r1 >= nregs)
8448                                err += efunc(pc, "invalid register %u\n", r1);
8449                        if (r2 != 0)
8450                                err += efunc(pc, "non-zero reserved bits\n");
8451                        if (rd >= nregs)
8452                                err += efunc(pc, "invalid register %u\n", rd);
8453                        if (rd == 0)
8454                                err += efunc(pc, "cannot write to %r0\n");
8455                        break;
8456                case DIF_OP_STB:
8457                case DIF_OP_STH:
8458                case DIF_OP_STW:
8459                case DIF_OP_STX:
8460                        if (r1 >= nregs)
8461                                err += efunc(pc, "invalid register %u\n", r1);
8462                        if (r2 != 0)
8463                                err += efunc(pc, "non-zero reserved bits\n");
8464                        if (rd >= nregs)
8465                                err += efunc(pc, "invalid register %u\n", rd);
8466                        if (rd == 0)
8467                                err += efunc(pc, "cannot write to 0 address\n");
8468                        break;
8469                case DIF_OP_CMP:
8470                case DIF_OP_SCMP:
8471                        if (r1 >= nregs)
8472                                err += efunc(pc, "invalid register %u\n", r1);
8473                        if (r2 >= nregs)
8474                                err += efunc(pc, "invalid register %u\n", r2);
8475                        if (rd != 0)
8476                                err += efunc(pc, "non-zero reserved bits\n");
8477                        break;
8478                case DIF_OP_TST:
8479                        if (r1 >= nregs)
8480                                err += efunc(pc, "invalid register %u\n", r1);
8481                        if (r2 != 0 || rd != 0)
8482                                err += efunc(pc, "non-zero reserved bits\n");
8483                        break;
8484                case DIF_OP_BA:
8485                case DIF_OP_BE:
8486                case DIF_OP_BNE:
8487                case DIF_OP_BG:
8488                case DIF_OP_BGU:
8489                case DIF_OP_BGE:
8490                case DIF_OP_BGEU:
8491                case DIF_OP_BL:
8492                case DIF_OP_BLU:
8493                case DIF_OP_BLE:
8494                case DIF_OP_BLEU:
8495                        if (label >= dp->dtdo_len) {
8496                                err += efunc(pc, "invalid branch target %u\n",
8497                                    label);
8498                        }
8499                        if (label <= pc) {
8500                                err += efunc(pc, "backward branch to %u\n",
8501                                    label);
8502                        }
8503                        break;
8504                case DIF_OP_RET:
8505                        if (r1 != 0 || r2 != 0)
8506                                err += efunc(pc, "non-zero reserved bits\n");
8507                        if (rd >= nregs)
8508                                err += efunc(pc, "invalid register %u\n", rd);
8509                        break;
```

```
8510                 case DIF_OP_NOP:
8511                 case DIF_OP_POPTS:
8512                 case DIF_OP_FLUSHTS:
8513                         if (r1 != 0 || r2 != 0 || rd != 0)
8514                                 err += efunc(pc, "non-zero reserved bits\n");
8515                         break;
8516                 case DIF_OP_SETX:
8517                         if (DIF_INSTR_INTEGER(instr) >= dp->dtdo_intlen) {
8518                                 err += efunc(pc, "invalid integer ref %u\n",
8519                                     DIF_INSTR_INTEGER(instr));
8520                         }
8521                         if (rd >= nregs)
8522                                 err += efunc(pc, "invalid register %u\n", rd);
8523                         if (rd == 0)
8524                                 err += efunc(pc, "cannot write to %r0\n");
8525                         break;
8526                 case DIF_OP_SETS:
8527                         if (DIF_INSTR_STRING(instr) >= dp->dtdo_strlen) {
8528                                 err += efunc(pc, "invalid string ref %u\n",
8529                                     DIF_INSTR_STRING(instr));
8530                         }
8531                         if (rd >= nregs)
8532                                 err += efunc(pc, "invalid register %u\n", rd);
8533                         if (rd == 0)
8534                                 err += efunc(pc, "cannot write to %r0\n");
8535                         break;
8536                 case DIF_OP_LDGA:
8537                 case DIF_OP_LDTA:
8538                         if (r1 > DIF_VAR_ARRAY_MAX)
8539                                 err += efunc(pc, "invalid array %u\n", r1);
8540                         if (r2 >= nregs)
8541                                 err += efunc(pc, "invalid register %u\n", r2);
8542                         if (rd >= nregs)
8543                                 err += efunc(pc, "invalid register %u\n", rd);
8544                         if (rd == 0)
8545                                 err += efunc(pc, "cannot write to %r0\n");
8546                         break;
8547                 case DIF_OP_LDGS:
8548                 case DIF_OP_LDTS:
8549                 case DIF_OP_LDLS:
8550                 case DIF_OP_LDGAA:
8551                 case DIF_OP_LDTAA:
8552                         if (v < DIF_VAR_OTHER_MIN || v > DIF_VAR_OTHER_MAX)
8553                                 err += efunc(pc, "invalid variable %u\n", v);
8554                         if (rd >= nregs)
8555                                 err += efunc(pc, "invalid register %u\n", rd);
8556                         if (rd == 0)
8557                                 err += efunc(pc, "cannot write to %r0\n");
8558                         break;
8559                 case DIF_OP_STGS:
8560                 case DIF_OP_STTS:
8561                 case DIF_OP_STLS:
8562                 case DIF_OP_STGAA:
8563                 case DIF_OP_STTAA:
8564                         if (v < DIF_VAR_OTHER_UBASE || v > DIF_VAR_OTHER_MAX)
8565                                 err += efunc(pc, "invalid variable %u\n", v);
8566                         if (rs >= nregs)
8567                                 err += efunc(pc, "invalid register %u\n", rd);
8568                         break;
8569                 case DIF_OP_CALL:
8570                         if (subr > DIF_SUBR_MAX)
8571                                 err += efunc(pc, "invalid subr %u\n", subr);
8572                         if (rd >= nregs)
8573                                 err += efunc(pc, "invalid register %u\n", rd);
8574                         if (rd == 0)
8575                                 err += efunc(pc, "cannot write to %r0\n");
```



```
8577                         if (subr == DIF_SUBR_COPYOUT ||
8578                             subr == DIF_SUBR_COPYOUTSTR) {
8579                                 dp->dtdo_destructive = 1;
8580                         }
8582                         if (subr == DIF_SUBR_GETF) {
8583                                 /*
8584                                  * If we have a getf() we need to record that
8585                                  * in our state.  Note that our state can be
8586                                  * NULL if this is a helper -- but in that
8587                                  * case, the call to getf() is itself illegal,
8588                                  * and will be caught (slightly later) when
8589                                  * the helper is validated.
8590                                  */
8591                                 if (vstate->dtvs_state != NULL)
8592                                         vstate->dtvs_state->dts_getf++;
8593                         }
8595 #endif /* ! codereview */
8596                         break;
8597                 case DIF_OP_PUSHTR:
8598                         if (type != DIF_TYPE_STRING && type != DIF_TYPE_CTF)
8599                                 err += efunc(pc, "invalid ref type %u\n", type);
8600                         if (r2 >= nregs)
8601                                 err += efunc(pc, "invalid register %u\n", r2);
8602                         if (rs >= nregs)
8603                                 err += efunc(pc, "invalid register %u\n", rs);
8604                         break;
8605                 case DIF_OP_PUSHTV:
8606                         if (type != DIF_TYPE_CTF)
8607                                 err += efunc(pc, "invalid val type %u\n", type);
8608                         if (r2 >= nregs)
8609                                 err += efunc(pc, "invalid register %u\n", r2);
8610                         if (rs >= nregs)
8611                                 err += efunc(pc, "invalid register %u\n", rs);
8612                         break;
8613                 default:
8614                         err += efunc(pc, "invalid opcode %u\n",
8615                             DIF_INSTR_OP(instr));
8616                 }
8617         }
8619         if (dp->dtdo_len != 0 &&
8620             DIF_INSTR_OP(dp->dtdo_buf[dp->dtdo_len - 1]) != DIF_OP_RET) {
8621                 err += efunc(dp->dtdo_len - 1,
8622                     "expected 'ret' as last DIF instruction\n");
8623         }
8625         if (!(dp->dtdo_rtype.dtdt_flags & DIF_TF_BYREF)) {
8626                 /*
8627                  * If we're not returning by reference, the size must be either
8628                  * 0 or the size of one of the base types.
8629                  */
8630                 switch (dp->dtdo_rtype.dtdt_size) {
8631                 case 0:
8632                 case sizeof (uint8_t):
8633                 case sizeof (uint16_t):
8634                 case sizeof (uint32_t):
8635                 case sizeof (uint64_t):
8636                         break;
8638                 default:
8639                         err += efunc(dp->dtdo_len - 1, "bad return size\n");
8640                 }
8641         }
```

```
8643         for (i = 0; i < dp->dtdo_varlen && err == 0; i++) {
8644                 dtrace_difv_t *v = &dp->dtdo_vartab[i], *existing = NULL;
8645                 dtrace_diftype_t *vt, *et;
8646                 uint_t id, ndx;
8647
8648                 if (v->dtdv_scope != DIFV_SCOPE_GLOBAL &&
8649                     v->dtdv_scope != DIFV_SCOPE_THREAD &&
8650                     v->dtdv_scope != DIFV_SCOPE_LOCAL) {
8651                         err += efunc(i, "unrecognized variable scope %d\n",
8652                             v->dtdv_scope);
8653                         break;
8654                 }
8655
8656                 if (v->dtdv_kind != DIFV_KIND_ARRAY &&
8657                     v->dtdv_kind != DIFV_KIND_SCALAR) {
8658                         err += efunc(i, "unrecognized variable type %d\n",
8659                             v->dtdv_kind);
8660                         break;
8661                 }
8662
8663                 if ((id = v->dtdv_id) > DIF_VARIABLE_MAX) {
8664                         err += efunc(i, "%d exceeds variable id limit\n", id);
8665                         break;
8666                 }
8667
8668                 if (id < DIF_VAR_OTHER_UBASE)
8669                         continue;
8670
8671                 /*
8672                  * For user-defined variables, we need to check that this
8673                  * definition is identical to any previous definition that we
8674                  * encountered.
8675                  */
8676                 ndx = id - DIF_VAR_OTHER_UBASE;
8677
8678                 switch (v->dtdv_scope) {
8679                 case DIFV_SCOPE_GLOBAL:
8680                         if (ndx < vstate->dtvs_nglobals) {
8681                                 dtrace_statvar_t *svar;
8682
8683                                 if ((svar = vstate->dtvs_globals[ndx]) != NULL)
8684                                         existing = &svar->dtsv_var;
8685                         }
8686
8687                         break;
8688
8689                 case DIFV_SCOPE_THREAD:
8690                         if (ndx < vstate->dtvs_ntlocals)
8691                                 existing = &vstate->dtvs_tlocals[ndx];
8692                         break;
8693
8694                 case DIFV_SCOPE_LOCAL:
8695                         if (ndx < vstate->dtvs_nlocals) {
8696                                 dtrace_statvar_t *svar;
8697
8698                                 if ((svar = vstate->dtvs_locals[ndx]) != NULL)
8699                                         existing = &svar->dtsv_var;
8700                         }
8701
8702                         break;
8703                 }
8704
8705                 vt = &v->dtdv_type;
8706
8707                 if (vt->dtdt_flags & DIF_TF_BYREF) {
```



```
8708                         if (vt->dtdt_size == 0) {
8709                                 err += efunc(i, "zero-sized variable\n");
8710                                 break;
8711                         }
8712
8713                         if (v->dtdv_scope == DIFV_SCOPE_GLOBAL &&
8714                             vt->dtdt_size > dtrace_global_maxsize) {
8715                                 err += efunc(i, "oversized by-ref global\n");
8716                                 break;
8717                         }
8718                 }
8719
8720                 if (existing == NULL || existing->dtdv_id == 0)
8721                         continue;
8722
8723                 ASSERT(existing->dtdv_id == v->dtdv_id);
8724                 ASSERT(existing->dtdv_scope == v->dtdv_scope);
8725
8726                 if (existing->dtdv_kind != v->dtdv_kind)
8727                         err += efunc(i, "%d changed variable kind\n", id);
8728
8729                 et = &existing->dtdv_type;
8730
8731                 if (vt->dtdt_flags != et->dtdt_flags) {
8732                         err += efunc(i, "%d changed variable type flags\n", id);
8733                         break;
8734                 }
8735
8736                 if (vt->dtdt_size != 0 && vt->dtdt_size != et->dtdt_size) {
8737                         err += efunc(i, "%d changed variable type size\n", id);
8738                         break;
8739                 }
8740         }
8741
8742         return (err);
8743 }
8744
8745 /*
8746  * Validate a DTrace DIF object that it is to be used as a helper.  Helpers
8747  * are much more constrained than normal DIFOs.  Specifically, they may
8748  * not:
8749  *
8750  * 1. Make calls to subroutines other than copyin(), copyinstr() or
8751  *    miscellaneous string routines
8752  * 2. Access DTrace variables other than the args[] array, and the
8753  *    curthread, pid, ppid, tid, execname, zonename, uid and gid variables.
8754  * 3. Have thread-local variables.
8755  * 4. Have dynamic variables.
8756  */
8757 static int
8758 dtrace_difo_validate_helper(dtrace_difo_t *dp)
8759 {
8760         int (*efunc)(uint_t pc, const char *, ...) = dtrace_difo_err;
8761         int err = 0;
8762         uint_t pc;
8763
8764         for (pc = 0; pc < dp->dtdo_len; pc++) {
8765                 dif_instr_t instr = dp->dtdo_buf[pc];
8766
8767                 uint_t v = DIF_INSTR_VAR(instr);
8768                 uint_t subr = DIF_INSTR_SUBR(instr);
8769                 uint_t op = DIF_INSTR_OP(instr);
8770
8771                 switch (op) {
8772                 case DIF_OP_OR:
8773                 case DIF_OP_XOR:
```

```
8774                 case DIF_OP_AND:
8775                 case DIF_OP_SLL:
8776                 case DIF_OP_SRL:
8777                 case DIF_OP_SRA:
8778                 case DIF_OP_SUB:
8779                 case DIF_OP_ADD:
8780                 case DIF_OP_MUL:
8781                 case DIF_OP_SDIV:
8782                 case DIF_OP_UDIV:
8783                 case DIF_OP_SREM:
8784                 case DIF_OP_UREM:
8785                 case DIF_OP_COPYS:
8786                 case DIF_OP_NOT:
8787                 case DIF_OP_MOV:
8788                 case DIF_OP_RLDSB:
8789                 case DIF_OP_RLDSH:
8790                 case DIF_OP_RLDSW:
8791                 case DIF_OP_RLDUB:
8792                 case DIF_OP_RLDUH:
8793                 case DIF_OP_RLDUW:
8794                 case DIF_OP_RLDX:
8795                 case DIF_OP_ULDSB:
8796                 case DIF_OP_ULDSH:
8797                 case DIF_OP_ULDSW:
8798                 case DIF_OP_ULDUB:
8799                 case DIF_OP_ULDUH:
8800                 case DIF_OP_ULDUW:
8801                 case DIF_OP_ULDX:
8802                 case DIF_OP_STB:
8803                 case DIF_OP_STH:
8804                 case DIF_OP_STW:
8805                 case DIF_OP_STX:
8806                 case DIF_OP_ALLOCS:
8807                 case DIF_OP_CMP:
8808                 case DIF_OP_SCMP:
8809                 case DIF_OP_TST:
8810                 case DIF_OP_BA:
8811                 case DIF_OP_BE:
8812                 case DIF_OP_BNE:
8813                 case DIF_OP_BG:
8814                 case DIF_OP_BGU:
8815                 case DIF_OP_BGE:
8816                 case DIF_OP_BGEU:
8817                 case DIF_OP_BL:
8818                 case DIF_OP_BLU:
8819                 case DIF_OP_BLE:
8820                 case DIF_OP_BLEU:
8821                 case DIF_OP_RET:
8822                 case DIF_OP_NOP:
8823                 case DIF_OP_POPTS:
8824                 case DIF_OP_FLUSHTS:
8825                 case DIF_OP_SETX:
8826                 case DIF_OP_SETS:
8827                 case DIF_OP_LDGA:
8828                 case DIF_OP_LDLS:
8829                 case DIF_OP_STGS:
8830                 case DIF_OP_STLS:
8831                 case DIF_OP_PUSHTR:
8832                 case DIF_OP_PUSHTV:
8833                         break;

8835                 case DIF_OP_LDGS:
8836                         if (v >= DIF_VAR_OTHER_UBASE)
8837                                 break;

8839                         if (v >= DIF_VAR_ARG0 && v <= DIF_VAR_ARG9)
```



```
8840                                 break;

8842                         if (v == DIF_VAR_CURTHREAD || v == DIF_VAR_PID ||
8843                             v == DIF_VAR_PPID || v == DIF_VAR_TID ||
8844                             v == DIF_VAR_EXECNAME || v == DIF_VAR_ZONENAME ||
8845                             v == DIF_VAR_UID || v == DIF_VAR_GID)
8846                                 break;

8848                         err += efunc(pc, "illegal variable %u\n", v);
8849                         break;

8851                 case DIF_OP_LDTA:
8852                 case DIF_OP_LDTS:
8853                 case DIF_OP_LDGAA:
8854                 case DIF_OP_LDTAA:
8855                         err += efunc(pc, "illegal dynamic variable load\n");
8856                         break;

8858                 case DIF_OP_STTS:
8859                 case DIF_OP_STGAA:
8860                 case DIF_OP_STTAA:
8861                         err += efunc(pc, "illegal dynamic variable store\n");
8862                         break;

8864                 case DIF_OP_CALL:
8865                         if (subr == DIF_SUBR_ALLOCA ||
8866                             subr == DIF_SUBR_BCOPY ||
8867                             subr == DIF_SUBR_COPYIN ||
8868                             subr == DIF_SUBR_COPYINTO ||
8869                             subr == DIF_SUBR_COPYINSTR ||
8870                             subr == DIF_SUBR_INDEX ||
8871                             subr == DIF_SUBR_INET_NTOA ||
8872                             subr == DIF_SUBR_INET_NTOA6 ||
8873                             subr == DIF_SUBR_INET_NTOP ||
8874                             subr == DIF_SUBR_LLTOSTR ||
8875                             subr == DIF_SUBR_RINDEX ||
8876                             subr == DIF_SUBR_STRCHR ||
8877                             subr == DIF_SUBR_STRJOIN ||
8878                             subr == DIF_SUBR_STRRCHR ||
8879                             subr == DIF_SUBR_STRSTR ||
8880                             subr == DIF_SUBR_HTONS ||
8881                             subr == DIF_SUBR_HTONL ||
8882                             subr == DIF_SUBR_HTONLL ||
8883                             subr == DIF_SUBR_NTOHS ||
8884                             subr == DIF_SUBR_NTOHL ||
8885                             subr == DIF_SUBR_NTOHLL)
8886                                 break;

8888                         err += efunc(pc, "invalid subr %u\n", subr);
8889                         break;

8891                 default:
8892                         err += efunc(pc, "invalid opcode %u\n",
8893                             DIF_INSTR_OP(instr));
8894                 }
8895         }

8897         return (err);
8898 }

8900 /*
8901  * Returns 1 if the expression in the DIF object can be cached on a per-thread
8902  * basis; 0 if not.
8903  */
8904 static int
8905 dtrace_difo_cacheable(dtrace_difo_t *dp)
```

```
8906 {
8907         int i;
8908 
8909         if (dp == NULL)
8910                 return (0);
8911 
8912         for (i = 0; i < dp->dtdo_varlen; i++) {
8913                 dtrace_difv_t *v = &dp->dtdo_vartab[i];
8914 
8915                 if (v->dtdv_scope != DIFV_SCOPE_GLOBAL)
8916                         continue;
8917 
8918                 switch (v->dtdv_id) {
8919                 case DIF_VAR_CURTHREAD:
8920                 case DIF_VAR_PID:
8921                 case DIF_VAR_TID:
8922                 case DIF_VAR_EXECNAME:
8923                 case DIF_VAR_ZONENAME:
8924                         break;
8925 
8926                 default:
8927                         return (0);
8928                 }
8929         }
8930 
8931         /*
8932          * This DIF object may be cacheable.  Now we need to look for any
8933          * array loading instructions, any memory loading instructions, or
8934          * any stores to thread-local variables.
8935          */
8936         for (i = 0; i < dp->dtdo_len; i++) {
8937                 uint_t op = DIF_INSTR_OP(dp->dtdo_buf[i]);
8938 
8939                 if ((op >= DIF_OP_LDSB && op <= DIF_OP_LDX) ||
8940                     (op >= DIF_OP_ULDSB && op <= DIF_OP_ULDX) ||
8941                     (op >= DIF_OP_RLDSB && op <= DIF_OP_RLDX) ||
8942                     op == DIF_OP_LDGA || op == DIF_OP_STTS)
8943                         return (0);
8944         }
8945 
8946         return (1);
8947 }
8948 
8949 static void
8950 dtrace_difo_hold(dtrace_difo_t *dp)
8951 {
8952         int i;
8953 
8954         ASSERT(MUTEX_HELD(&dtrace_lock));
8955 
8956         dp->dtdo_refcnt++;
8957         ASSERT(dp->dtdo_refcnt != 0);
8958 
8959         /*
8960          * We need to check this DIF object for references to the variable
8961          * DIF_VAR_VTIMESTAMP.
8962          */
8963         for (i = 0; i < dp->dtdo_varlen; i++) {
8964                 dtrace_difv_t *v = &dp->dtdo_vartab[i];
8965 
8966                 if (v->dtdv_id != DIF_VAR_VTIMESTAMP)
8967                         continue;
8968 
8969                 if (dtrace_vtime_references++ == 0)
8970                         dtrace_vtime_enable();
8971         }
```



```
8972 }
8973 
8974 /*
8975  * This routine calculates the dynamic variable chunksize for a given DIF
8976  * object.  The calculation is not fool-proof, and can probably be tricked by
8977  * malicious DIF -- but it works for all compiler-generated DIF.  Because this
8978  * calculation is likely imperfect, dtrace_dynvar() is able to gracefully fail
8979  * if a dynamic variable size exceeds the chunksize.
8980  */
8981 static void
8982 dtrace_difo_chunksize(dtrace_difo_t *dp, dtrace_vstate_t *vstate)
8983 {
8984         uint64_t sval;
8985         dtrace_key_t tupregs[DIF_DTR_NREGS + 2]; /* +2 for thread and id */
8986         const dif_instr_t *text = dp->dtdo_buf;
8987         uint_t pc, srd = 0;
8988         uint_t ttop = 0;
8989         size_t size, ksize;
8990         uint_t id, i;
8991 
8992         for (pc = 0; pc < dp->dtdo_len; pc++) {
8993                 dif_instr_t instr = text[pc];
8994                 uint_t op = DIF_INSTR_OP(instr);
8995                 uint_t rd = DIF_INSTR_RD(instr);
8996                 uint_t r1 = DIF_INSTR_R1(instr);
8997                 uint_t nkeys = 0;
8998                 uchar_t scope;
8999 
9000                 dtrace_key_t *key = tupregs;
9001 
9002                 switch (op) {
9003                 case DIF_OP_SETX:
9004                         sval = dp->dtdo_inttab[DIF_INSTR_INTEGER(instr)];
9005                         srd = rd;
9006                         continue;
9007 
9008                 case DIF_OP_STTS:
9009                         key = &tupregs[DIF_DTR_NREGS];
9010                         key[0].dttk_size = 0;
9011                         key[1].dttk_size = 0;
9012                         nkeys = 2;
9013                         scope = DIFV_SCOPE_THREAD;
9014                         break;
9015 
9016                 case DIF_OP_STGAA:
9017                 case DIF_OP_STTAA:
9018                         nkeys = ttop;
9019 
9020                         if (DIF_INSTR_OP(instr) == DIF_OP_STTAA)
9021                                 key[nkeys++].dttk_size = 0;
9022 
9023                         key[nkeys++].dttk_size = 0;
9024 
9025                         if (op == DIF_OP_STTAA) {
9026                                 scope = DIFV_SCOPE_THREAD;
9027                         } else {
9028                                 scope = DIFV_SCOPE_GLOBAL;
9029                         }
9030 
9031                         break;
9032 
9033                 case DIF_OP_PUSHTR:
9034                         if (ttop == DIF_DTR_NREGS)
9035                                 return;
9036 
9037                         if ((srd == 0 || sval == 0) && r1 == DIF_TYPE_STRING) {
```

```
9038                                /*
9039                                 * If the register for the size of the "pushtr"
9040                                 * is %r0 (or the value is 0) and the type is
9041                                 * a string, we'll use the system-wide default
9042                                 * string size.
9043                                 */
9044                                tupregs[ttop++].dttk_size =
9045                                    dtrace_strsize_default;
9046                        } else {
9047                                if (srd == 0)
9048                                        return;

9050                                tupregs[ttop++].dttk_size = sval;
9051                        }

9053                        break;

9055                case DIF_OP_PUSHTV:
9056                        if (ttop == DIF_DTR_NREGS)
9057                                return;

9059                        tupregs[ttop++].dttk_size = 0;
9060                        break;

9062                case DIF_OP_FLUSHTS:
9063                        ttop = 0;
9064                        break;

9066                case DIF_OP_POPTS:
9067                        if (ttop != 0)
9068                                ttop--;
9069                        break;
9070                }

9072                sval = 0;
9073                srd = 0;

9075                if (nkeys == 0)
9076                        continue;

9078                /*
9079                 * We have a dynamic variable allocation; calculate its size.
9080                 */
9081                for (ksize = 0, i = 0; i < nkeys; i++)
9082                        ksize += P2ROUNDUP(key[i].dttk_size, sizeof (uint64_t));

9084                size = sizeof (dtrace_dynvar_t);
9085                size += sizeof (dtrace_key_t) * (nkeys - 1);
9086                size += ksize;

9088                /*
9089                 * Now we need to determine the size of the stored data.
9090                 */
9091                id = DIF_INSTR_VAR(instr);

9093                for (i = 0; i < dp->dtdo_varlen; i++) {
9094                        dtrace_difv_t *v = &dp->dtdo_vartab[i];

9096                        if (v->dtdv_id == id && v->dtdv_scope == scope) {
9097                                size += v->dtdv_type.dtdt_size;
9098                                break;
9099                        }
9100                }

9102                if (i == dp->dtdo_varlen)
9103                        return;
```



```
9105                /*
9106                 * We have the size.  If this is larger than the chunk size
9107                 * for our dynamic variable state, reset the chunk size.
9108                 */
9109                size = P2ROUNDUP(size, sizeof (uint64_t));

9111                if (size > vstate->dtvs_dynvars.dtds_chunksize)
9112                        vstate->dtvs_dynvars.dtds_chunksize = size;
9113        }
9114 }

9116 static void
9117 dtrace_difo_init(dtrace_difo_t *dp, dtrace_vstate_t *vstate)
9118 {
9119        int i, oldsvars, osz, nsz, otlocals, ntlocals;
9120        uint_t id;

9122        ASSERT(MUTEX_HELD(&dtrace_lock));
9123        ASSERT(dp->dtdo_buf != NULL && dp->dtdo_len != 0);

9125        for (i = 0; i < dp->dtdo_varlen; i++) {
9126                dtrace_difv_t *v = &dp->dtdo_vartab[i];
9127                dtrace_statvar_t *svar, ***svarp;
9128                size_t dsize = 0;
9129                uint8_t scope = v->dtdv_scope;
9130                int *np;

9132                if ((id = v->dtdv_id) < DIF_VAR_OTHER_UBASE)
9133                        continue;

9135                id -= DIF_VAR_OTHER_UBASE;

9137                switch (scope) {
9138                case DIFV_SCOPE_THREAD:
9139                        while (id >= (otlocals = vstate->dtvs_ntlocals)) {
9140                                dtrace_difv_t *tlocals;

9142                                if ((ntlocals = (otlocals << 1)) == 0)
9143                                        ntlocals = 1;

9145                                osz = otlocals * sizeof (dtrace_difv_t);
9146                                nsz = ntlocals * sizeof (dtrace_difv_t);

9148                                tlocals = kmem_zalloc(nsz, KM_SLEEP);

9150                                if (osz != 0) {
9151                                        bcopy(vstate->dtvs_tlocals,
9152                                            tlocals, osz);
9153                                        kmem_free(vstate->dtvs_tlocals, osz);
9154                                }

9156                                vstate->dtvs_tlocals = tlocals;
9157                                vstate->dtvs_ntlocals = ntlocals;
9158                        }

9160                        vstate->dtvs_tlocals[id] = *v;
9161                        continue;

9163                case DIFV_SCOPE_LOCAL:
9164                        np = &vstate->dtvs_nlocals;
9165                        svarp = &vstate->dtvs_locals;

9167                        if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF)
9168                                dsize = NCPU * (v->dtdv_type.dtdt_size +
9169                                    sizeof (uint64_t));
```

```
9170                        else
9171                                dsize = NCPU * sizeof (uint64_t);
9173                        break;
9175                case DIFV_SCOPE_GLOBAL:
9176                        np = &vstate->dtvs_nglobals;
9177                        svarp = &vstate->dtvs_globals;
9179                        if (v->dtdv_type.dtdt_flags & DIF_TF_BYREF)
9180                                dsize = v->dtdv_type.dtdt_size +
9181                                    sizeof (uint64_t);
9183                        break;
9185                default:
9186                        ASSERT(0);
9187                }
9189                while (id >= (oldsvars = *np)) {
9190                        dtrace_statvar_t **statics;
9191                        int newsvars, oldsize, newsize;
9193                        if ((newsvars = (oldsvars << 1)) == 0)
9194                                newsvars = 1;
9196                        oldsize = oldsvars * sizeof (dtrace_statvar_t *);
9197                        newsize = newsvars * sizeof (dtrace_statvar_t *);
9199                        statics = kmem_zalloc(newsize, KM_SLEEP);
9201                        if (oldsize != 0) {
9202                                bcopy(*svarp, statics, oldsize);
9203                                kmem_free(*svarp, oldsize);
9204                        }
9206                        *svarp = statics;
9207                        *np = newsvars;
9208                }
9210                if ((svar = (*svarp)[id]) == NULL) {
9211                        svar = kmem_zalloc(sizeof (dtrace_statvar_t), KM_SLEEP);
9212                        svar->dtsv_var = *v;
9214                        if ((svar->dtsv_size = dsize) != 0) {
9215                                svar->dtsv_data = (uint64_t)(uintptr_t)
9216                                    kmem_zalloc(dsize, KM_SLEEP);
9217                        }
9219                        (*svarp)[id] = svar;
9220                }
9222                svar->dtsv_refcnt++;
9223        }
9225        dtrace_difo_chunksize(dp, vstate);
9226        dtrace_difo_hold(dp);
9227 }
9229 static dtrace_difo_t *
9230 dtrace_difo_duplicate(dtrace_difo_t *dp, dtrace_vstate_t *vstate)
9231 {
9232        dtrace_difo_t *new;
9233        size_t sz;
9235        ASSERT(dp->dtdo_buf != NULL);
```



```
9236        ASSERT(dp->dtdo_refcnt != 0);
9238        new = kmem_zalloc(sizeof (dtrace_difo_t), KM_SLEEP);
9240        ASSERT(dp->dtdo_buf != NULL);
9241        sz = dp->dtdo_len * sizeof (dif_instr_t);
9242        new->dtdo_buf = kmem_alloc(sz, KM_SLEEP);
9243        bcopy(dp->dtdo_buf, new->dtdo_buf, sz);
9244        new->dtdo_len = dp->dtdo_len;
9246        if (dp->dtdo_strtab != NULL) {
9247                ASSERT(dp->dtdo_strlen != 0);
9248                new->dtdo_strtab = kmem_alloc(dp->dtdo_strlen, KM_SLEEP);
9249                bcopy(dp->dtdo_strtab, new->dtdo_strtab, dp->dtdo_strlen);
9250                new->dtdo_strlen = dp->dtdo_strlen;
9251        }
9253        if (dp->dtdo_inttab != NULL) {
9254                ASSERT(dp->dtdo_intlen != 0);
9255                sz = dp->dtdo_intlen * sizeof (uint64_t);
9256                new->dtdo_inttab = kmem_alloc(sz, KM_SLEEP);
9257                bcopy(dp->dtdo_inttab, new->dtdo_inttab, sz);
9258                new->dtdo_intlen = dp->dtdo_intlen;
9259        }
9261        if (dp->dtdo_vartab != NULL) {
9262                ASSERT(dp->dtdo_varlen != 0);
9263                sz = dp->dtdo_varlen * sizeof (dtrace_difv_t);
9264                new->dtdo_vartab = kmem_alloc(sz, KM_SLEEP);
9265                bcopy(dp->dtdo_vartab, new->dtdo_vartab, sz);
9266                new->dtdo_varlen = dp->dtdo_varlen;
9267        }
9269        dtrace_difo_init(new, vstate);
9270        return (new);
9271 }
9273 static void
9274 dtrace_difo_destroy(dtrace_difo_t *dp, dtrace_vstate_t *vstate)
9275 {
9276        int i;
9278        ASSERT(dp->dtdo_refcnt == 0);
9280        for (i = 0; i < dp->dtdo_varlen; i++) {
9281                dtrace_difv_t *v = &dp->dtdo_vartab[i];
9282                dtrace_statvar_t *svar, **svarp;
9283                uint_t id;
9284                uint8_t scope = v->dtdv_scope;
9285                int *np;
9287                switch (scope) {
9288                case DIFV_SCOPE_THREAD:
9289                        continue;
9291                case DIFV_SCOPE_LOCAL:
9292                        np = &vstate->dtvs_nlocals;
9293                        svarp = vstate->dtvs_locals;
9294                        break;
9296                case DIFV_SCOPE_GLOBAL:
9297                        np = &vstate->dtvs_nglobals;
9298                        svarp = vstate->dtvs_globals;
9299                        break;
9301                default:
```

```
9302                        ASSERT(0);
9303                }

9305                if ((id = v->dtdv_id) < DIF_VAR_OTHER_UBASE)
9306                        continue;

9308                id -= DIF_VAR_OTHER_UBASE;
9309                ASSERT(id < *np);

9311                svar = svarp[id];
9312                ASSERT(svar != NULL);
9313                ASSERT(svar->dtsv_refcnt > 0);

9315                if (--svar->dtsv_refcnt > 0)
9316                        continue;

9318                if (svar->dtsv_size != 0) {
9319                        ASSERT(svar->dtsv_data != NULL);
9320                        kmem_free((void *)(uintptr_t)svar->dtsv_data,
9321                            svar->dtsv_size);
9322                }

9324                kmem_free(svar, sizeof (dtrace_statvar_t));
9325                svarp[id] = NULL;
9326        }

9328        kmem_free(dp->dtdo_buf, dp->dtdo_len * sizeof (dif_instr_t));
9329        kmem_free(dp->dtdo_inttab, dp->dtdo_intlen * sizeof (uint64_t));
9330        kmem_free(dp->dtdo_strtab, dp->dtdo_strlen);
9331        kmem_free(dp->dtdo_vartab, dp->dtdo_varlen * sizeof (dtrace_difv_t));

9333        kmem_free(dp, sizeof (dtrace_difo_t));
9334 }

9336 static void
9337 dtrace_difo_release(dtrace_difo_t *dp, dtrace_vstate_t *vstate)
9338 {
9339        int i;

9341        ASSERT(MUTEX_HELD(&dtrace_lock));
9342        ASSERT(dp->dtdo_refcnt != 0);

9344        for (i = 0; i < dp->dtdo_varlen; i++) {
9345                dtrace_difv_t *v = &dp->dtdo_vartab[i];

9347                if (v->dtdv_id != DIF_VAR_VTIMESTAMP)
9348                        continue;

9350                ASSERT(dtrace_vtime_references > 0);
9351                if (--dtrace_vtime_references == 0)
9352                        dtrace_vtime_disable();
9353        }

9355        if (--dp->dtdo_refcnt == 0)
9356                dtrace_difo_destroy(dp, vstate);
9357 }

9359 /*
9360  * DTrace Format Functions
9361  */
9362 static uint16_t
9363 dtrace_format_add(dtrace_state_t *state, char *str)
9364 {
9365        char *fmt, **new;
9366        uint16_t ndx, len = strlen(str) + 1;
```



```
9368        fmt = kmem_zalloc(len, KM_SLEEP);
9369        bcopy(str, fmt, len);

9371        for (ndx = 0; ndx < state->dts_nformats; ndx++) {
9372                if (state->dts_formats[ndx] == NULL) {
9373                        state->dts_formats[ndx] = fmt;
9374                        return (ndx + 1);
9375                }
9376        }

9378        if (state->dts_nformats == USHRT_MAX) {
9379                /*
9380                 * This is only likely if a denial-of-service attack is being
9381                 * attempted.  As such, it's okay to fail silently here.
9382                 */
9383                kmem_free(fmt, len);
9384                return (0);
9385        }

9387        /*
9388         * For simplicity, we always resize the formats array to be exactly the
9389         * number of formats.
9390         */
9391        ndx = state->dts_nformats++;
9392        new = kmem_alloc((ndx + 1) * sizeof (char *), KM_SLEEP);

9394        if (state->dts_formats != NULL) {
9395                ASSERT(ndx != 0);
9396                bcopy(state->dts_formats, new, ndx * sizeof (char *));
9397                kmem_free(state->dts_formats, ndx * sizeof (char *));
9398        }

9400        state->dts_formats = new;
9401        state->dts_formats[ndx] = fmt;

9403        return (ndx + 1);
9404 }

9406 static void
9407 dtrace_format_remove(dtrace_state_t *state, uint16_t format)
9408 {
9409        char *fmt;

9411        ASSERT(state->dts_formats != NULL);
9412        ASSERT(format <= state->dts_nformats);
9413        ASSERT(state->dts_formats[format - 1] != NULL);

9415        fmt = state->dts_formats[format - 1];
9416        kmem_free(fmt, strlen(fmt) + 1);
9417        state->dts_formats[format - 1] = NULL;
9418 }

9420 static void
9421 dtrace_format_destroy(dtrace_state_t *state)
9422 {
9423        int i;

9425        if (state->dts_nformats == 0) {
9426                ASSERT(state->dts_formats == NULL);
9427                return;
9428        }

9430        ASSERT(state->dts_formats != NULL);

9432        for (i = 0; i < state->dts_nformats; i++) {
9433                char *fmt = state->dts_formats[i];
```

```
9435                 if (fmt == NULL)
9436                         continue;

9438                 kmem_free(fmt, strlen(fmt) + 1);
9439         }

9441         kmem_free(state->dts_formats, state->dts_nformats * sizeof (char *));
9442         state->dts_nformats = 0;
9443         state->dts_formats = NULL;
9444 }

9446 /*
9447  * DTrace Predicate Functions
9448  */
9449 static dtrace_predicate_t *
9450 dtrace_predicate_create(dtrace_difo_t *dp)
9451 {
9452         dtrace_predicate_t *pred;

9454         ASSERT(MUTEX_HELD(&dtrace_lock));
9455         ASSERT(dp->dtdo_refcnt != 0);

9457         pred = kmem_zalloc(sizeof (dtrace_predicate_t), KM_SLEEP);
9458         pred->dtp_difo = dp;
9459         pred->dtp_refcnt = 1;

9461         if (!dtrace_difo_cacheable(dp))
9462                 return (pred);

9464         if (dtrace_predcache_id == DTRACE_CACHEIDNONE) {
9465                 /*
9466                  * This is only theoretically possible -- we have had 2^32
9467                  * cacheable predicates on this machine.  We cannot allow any
9468                  * more predicates to become cacheable:  as unlikely as it is,
9469                  * there may be a thread caching a (now stale) predicate cache
9470                  * ID. (N.B.: the temptation is being successfully resisted to
9471                  * have this cmn_err() "Holy shit -- we executed this code!")
9472                  */
9473                 return (pred);
9474         }

9476         pred->dtp_cacheid = dtrace_predcache_id++;

9478         return (pred);
9479 }

9481 static void
9482 dtrace_predicate_hold(dtrace_predicate_t *pred)
9483 {
9484         ASSERT(MUTEX_HELD(&dtrace_lock));
9485         ASSERT(pred->dtp_difo != NULL && pred->dtp_difo->dtdo_refcnt != 0);
9486         ASSERT(pred->dtp_refcnt > 0);

9488         pred->dtp_refcnt++;
9489 }

9491 static void
9492 dtrace_predicate_release(dtrace_predicate_t *pred, dtrace_vstate_t *vstate)
9493 {
9494         dtrace_difo_t *dp = pred->dtp_difo;

9496         ASSERT(MUTEX_HELD(&dtrace_lock));
9497         ASSERT(dp != NULL && dp->dtdo_refcnt != 0);
9498         ASSERT(pred->dtp_refcnt > 0);
```



```
9500         if (--pred->dtp_refcnt == 0) {
9501                 dtrace_difo_release(pred->dtp_difo, vstate);
9502                 kmem_free(pred, sizeof (dtrace_predicate_t));
9503         }
9504 }

9506 /*
9507  * DTrace Action Description Functions
9508  */
9509 static dtrace_actdesc_t *
9510 dtrace_actdesc_create(dtrace_actkind_t kind, uint32_t ntuple,
9511     uint64_t uarg, uint64_t arg)
9512 {
9513         dtrace_actdesc_t *act;

9515         ASSERT(!DTRACEACT_ISPRINTFLIKE(kind) || (arg != NULL &&
9516             arg >= KERNELBASE) || (arg == NULL && kind == DTRACEACT_PRINTA));

9518         act = kmem_zalloc(sizeof (dtrace_actdesc_t), KM_SLEEP);
9519         act->dtad_kind = kind;
9520         act->dtad_ntuple = ntuple;
9521         act->dtad_uarg = uarg;
9522         act->dtad_arg = arg;
9523         act->dtad_refcnt = 1;

9525         return (act);
9526 }

9528 static void
9529 dtrace_actdesc_hold(dtrace_actdesc_t *act)
9530 {
9531         ASSERT(act->dtad_refcnt >= 1);
9532         act->dtad_refcnt++;
9533 }

9535 static void
9536 dtrace_actdesc_release(dtrace_actdesc_t *act, dtrace_vstate_t *vstate)
9537 {
9538         dtrace_actkind_t kind = act->dtad_kind;
9539         dtrace_difo_t *dp;

9541         ASSERT(act->dtad_refcnt >= 1);

9543         if (--act->dtad_refcnt != 0)
9544                 return;

9546         if ((dp = act->dtad_difo) != NULL)
9547                 dtrace_difo_release(dp, vstate);

9549         if (DTRACEACT_ISPRINTFLIKE(kind)) {
9550                 char *str = (char *)(uintptr_t)act->dtad_arg;

9552                 ASSERT((str != NULL && (uintptr_t)str >= KERNELBASE) ||
9553                     (str == NULL && act->dtad_kind == DTRACEACT_PRINTA));

9555                 if (str != NULL)
9556                         kmem_free(str, strlen(str) + 1);
9557         }

9559         kmem_free(act, sizeof (dtrace_actdesc_t));
9560 }

9562 /*
9563  * DTrace ECB Functions
9564  */
9565 static dtrace_ecb_t *
```

```
9566 dtrace_ecb_add(dtrace_state_t *state, dtrace_probe_t *probe)
9567 {
9568         dtrace_ecb_t *ecb;
9569         dtrace_epid_t epid;

9571         ASSERT(MUTEX_HELD(&dtrace_lock));

9573         ecb = kmem_zalloc(sizeof (dtrace_ecb_t), KM_SLEEP);
9574         ecb->dte_predicate = NULL;
9575         ecb->dte_probe = probe;

9577         /*
9578          * The default size is the size of the default action: recording
9579          * the header.
9580          */
9581         ecb->dte_size = ecb->dte_needed = sizeof (dtrace_rechdr_t);
9582         ecb->dte_alignment = sizeof (dtrace_epid_t);

9584         epid = state->dts_epid++;

9586         if (epid - 1 >= state->dts_necbs) {
9587                 dtrace_ecb_t **oecbs = state->dts_ecbs, **ecbs;
9588                 int necbs = state->dts_necbs << 1;

9590                 ASSERT(epid == state->dts_necbs + 1);

9592                 if (necbs == 0) {
9593                         ASSERT(oecbs == NULL);
9594                         necbs = 1;
9595                 }

9597                 ecbs = kmem_zalloc(necbs * sizeof (*ecbs), KM_SLEEP);

9599                 if (oecbs != NULL)
9600                         bcopy(oecbs, ecbs, state->dts_necbs * sizeof (*ecbs));

9602                 dtrace_membar_producer();
9603                 state->dts_ecbs = ecbs;

9605                 if (oecbs != NULL) {
9606                         /*
9607                          * If this state is active, we must dtrace_sync()
9608                          * before we can free the old dts_ecbs array:  we're
9609                          * coming in hot, and there may be active ring
9610                          * buffer processing (which indexes into the dts_ecbs
9611                          * array) on another CPU.
9612                          */
9613                         if (state->dts_activity != DTRACE_ACTIVITY_INACTIVE)
9614                                 dtrace_sync();

9616                         kmem_free(oecbs, state->dts_necbs * sizeof (*ecbs));
9617                 }

9619                 dtrace_membar_producer();
9620                 state->dts_necbs = necbs;
9621         }

9623         ecb->dte_state = state;

9625         ASSERT(state->dts_ecbs[epid - 1] == NULL);
9626         dtrace_membar_producer();
9627         state->dts_ecbs[(ecb->dte_epid = epid) - 1] = ecb;

9629         return (ecb);
9630 }
```



```
9632 static int
9633 dtrace_ecb_enable(dtrace_ecb_t *ecb)
9634 {
9635         dtrace_probe_t *probe = ecb->dte_probe;

9637         ASSERT(MUTEX_HELD(&cpu_lock));
9638         ASSERT(MUTEX_HELD(&dtrace_lock));
9639         ASSERT(ecb->dte_next == NULL);

9641         if (probe == NULL) {
9642                 /*
9643                  * This is the NULL probe -- there's nothing to do.
9644                  */
9645                 return (0);
9646         }

9648         if (probe->dtpr_ecb == NULL) {
9649                 dtrace_provider_t *prov = probe->dtpr_provider;

9651                 /*
9652                  * We're the first ECB on this probe.
9653                  */
9654                 probe->dtpr_ecb = probe->dtpr_ecb_last = ecb;

9656                 if (ecb->dte_predicate != NULL)
9657                         probe->dtpr_predcache = ecb->dte_predicate->dtp_cacheid;

9659                 return (prov->dtpv_pops.dtps_enable(prov->dtpv_arg,
9660                     probe->dtpr_id, probe->dtpr_arg));
9661         } else {
9662                 /*
9663                  * This probe is already active.  Swing the last pointer to
9664                  * point to the new ECB, and issue a dtrace_sync() to assure
9665                  * that all CPUs have seen the change.
9666                  */
9667                 ASSERT(probe->dtpr_ecb_last != NULL);
9668                 probe->dtpr_ecb_last->dte_next = ecb;
9669                 probe->dtpr_ecb_last = ecb;
9670                 probe->dtpr_predcache = 0;

9672                 dtrace_sync();
9673                 return (0);
9674         }
9675 }

9677 static void
9678 dtrace_ecb_resize(dtrace_ecb_t *ecb)
9679 {
9680         dtrace_action_t *act;
9681         uint32_t curneeded = UINT32_MAX;
9682         uint32_t aggbase = UINT32_MAX;

9684         /*
9685          * If we record anything, we always record the dtrace_rechdr_t.  (And
9686          * we always record it first.)
9687          */
9688         ecb->dte_size = sizeof (dtrace_rechdr_t);
9689         ecb->dte_alignment = sizeof (dtrace_epid_t);

9691         for (act = ecb->dte_action; act != NULL; act = act->dta_next) {
9692                 dtrace_recdesc_t *rec = &act->dta_rec;
9693                 ASSERT(rec->dtrd_size > 0 || rec->dtrd_alignment == 1);

9695                 ecb->dte_alignment = MAX(ecb->dte_alignment,
9696                     rec->dtrd_alignment);
```

```
9698                if (DTRACEACT_ISAGG(act->dta_kind)) {
9699                        dtrace_aggregation_t *agg = (dtrace_aggregation_t *)act;

9701                        ASSERT(rec->dtrd_size != 0);
9702                        ASSERT(agg->dtag_first != NULL);
9703                        ASSERT(act->dta_prev->dta_intuple);
9704                        ASSERT(aggbase != UINT32_MAX);
9705                        ASSERT(curneeded != UINT32_MAX);

9707                        agg->dtag_base = aggbase;

9709                        curneeded = P2ROUNDUP(curneeded, rec->dtrd_alignment);
9710                        rec->dtrd_offset = curneeded;
9711                        curneeded += rec->dtrd_size;
9712                        ecb->dte_needed = MAX(ecb->dte_needed, curneeded);

9714                        aggbase = UINT32_MAX;
9715                        curneeded = UINT32_MAX;
9716                } else if (act->dta_intuple) {
9717                        if (curneeded == UINT32_MAX) {
9718                                /*
9719                                 * This is the first record in a tuple.  Align
9720                                 * curneeded to be at offset 4 in an 8-byte
9721                                 * aligned block.
9722                                 */
9723                                ASSERT(act->dta_prev == NULL ||
9724                                    !act->dta_prev->dta_intuple);
9725                                ASSERT3U(aggbase, ==, UINT32_MAX);
9726                                curneeded = P2PHASEUP(ecb->dte_size,
9727                                    sizeof (uint64_t), sizeof (dtrace_aggid_t));

9729                                aggbase = curneeded - sizeof (dtrace_aggid_t);
9730                                ASSERT(IS_P2ALIGNED(aggbase,
9731                                    sizeof (uint64_t)));
9732                        }
9733                        curneeded = P2ROUNDUP(curneeded, rec->dtrd_alignment);
9734                        rec->dtrd_offset = curneeded;
9735                        curneeded += rec->dtrd_size;
9736                } else {
9737                        /* tuples must be followed by an aggregation */
9738                        ASSERT(act->dta_prev == NULL ||
9739                            !act->dta_prev->dta_intuple);

9741                        ecb->dte_size = P2ROUNDUP(ecb->dte_size,
9742                            rec->dtrd_alignment);
9743                        rec->dtrd_offset = ecb->dte_size;
9744                        ecb->dte_size += rec->dtrd_size;
9745                        ecb->dte_needed = MAX(ecb->dte_needed, ecb->dte_size);
9746                }
9747        }

9749        if ((act = ecb->dte_action) != NULL &&
9750            !(act->dta_kind == DTRACEACT_SPECULATE && act->dta_next == NULL) &&
9751            ecb->dte_size == sizeof (dtrace_rechdr_t)) {
9752                /*
9753                 * If the size is still sizeof (dtrace_rechdr_t), then all
9754                 * actions store no data; set the size to 0.
9755                 */
9756                ecb->dte_size = 0;
9757        }

9759        ecb->dte_size = P2ROUNDUP(ecb->dte_size, sizeof (dtrace_epid_t));
9760        ecb->dte_needed = P2ROUNDUP(ecb->dte_needed, (sizeof (dtrace_epid_t)));
9761        ecb->dte_state->dts_needed = MAX(ecb->dte_state->dts_needed,
9762            ecb->dte_needed);
9763 }
```



```
9765 static dtrace_action_t *
9766 dtrace_ecb_aggregation_create(dtrace_ecb_t *ecb, dtrace_actdesc_t *desc)
9767 {
9768        dtrace_aggregation_t *agg;
9769        size_t size = sizeof (uint64_t);
9770        int ntuple = desc->dtad_ntuple;
9771        dtrace_action_t *act;
9772        dtrace_recdesc_t *frec;
9773        dtrace_aggid_t aggid;
9774        dtrace_state_t *state = ecb->dte_state;

9776        agg = kmem_zalloc(sizeof (dtrace_aggregation_t), KM_SLEEP);
9777        agg->dtag_ecb = ecb;

9779        ASSERT(DTRACEACT_ISAGG(desc->dtad_kind));

9781        switch (desc->dtad_kind) {
9782        case DTRACEAGG_MIN:
9783                agg->dtag_initial = INT64_MAX;
9784                agg->dtag_aggregate = dtrace_aggregate_min;
9785                break;

9787        case DTRACEAGG_MAX:
9788                agg->dtag_initial = INT64_MIN;
9789                agg->dtag_aggregate = dtrace_aggregate_max;
9790                break;

9792        case DTRACEAGG_COUNT:
9793                agg->dtag_aggregate = dtrace_aggregate_count;
9794                break;

9796        case DTRACEAGG_QUANTIZE:
9797                agg->dtag_aggregate = dtrace_aggregate_quantize;
9798                size = (((sizeof (uint64_t) * NBBY) - 1) * 2 + 1) *
9799                    sizeof (uint64_t);
9800                break;

9802        case DTRACEAGG_LQUANTIZE: {
9803                uint16_t step = DTRACE_LQUANTIZE_STEP(desc->dtad_arg);
9804                uint16_t levels = DTRACE_LQUANTIZE_LEVELS(desc->dtad_arg);

9806                agg->dtag_initial = desc->dtad_arg;
9807                agg->dtag_aggregate = dtrace_aggregate_lquantize;

9809                if (step == 0 || levels == 0)
9810                        goto err;

9812                size = levels * sizeof (uint64_t) + 3 * sizeof (uint64_t);
9813                break;
9814        }

9816        case DTRACEAGG_LLQUANTIZE: {
9817                uint16_t factor = DTRACE_LLQUANTIZE_FACTOR(desc->dtad_arg);
9818                uint16_t low = DTRACE_LLQUANTIZE_LOW(desc->dtad_arg);
9819                uint16_t high = DTRACE_LLQUANTIZE_HIGH(desc->dtad_arg);
9820                uint16_t nsteps = DTRACE_LLQUANTIZE_NSTEP(desc->dtad_arg);
9821                int64_t v;

9823                agg->dtag_initial = desc->dtad_arg;
9824                agg->dtag_aggregate = dtrace_aggregate_llquantize;

9826                if (factor < 2 || low >= high || nsteps < factor)
9827                        goto err;

9829                /*
```

```
9830                 * Now check that the number of steps evenly divides a power
9831                 * of the factor.  (This assures both integer bucket size and
9832                 * linearity within each magnitude.)
9833                 */
9834                for (v = factor; v < nsteps; v *= factor)
9835                        continue;
9836
9837                if ((v % nsteps) || (nsteps % factor))
9838                        goto err;
9839
9840                size = (dtrace_aggregate_llquantize_bucket(factor,
9841                    low, high, nsteps, INT64_MAX) + 2) * sizeof (uint64_t);
9842                break;
9843        }
9844
9845        case DTRACEAGG_AVG:
9846                agg->dtag_aggregate = dtrace_aggregate_avg;
9847                size = sizeof (uint64_t) * 2;
9848                break;
9849
9850        case DTRACEAGG_STDDEV:
9851                agg->dtag_aggregate = dtrace_aggregate_stddev;
9852                size = sizeof (uint64_t) * 4;
9853                break;
9854
9855        case DTRACEAGG_SUM:
9856                agg->dtag_aggregate = dtrace_aggregate_sum;
9857                break;
9858
9859        default:
9860                goto err;
9861        }
9862
9863        agg->dtag_action.dta_rec.dtrd_size = size;
9864
9865        if (ntuple == 0)
9866                goto err;
9867
9868        /*
9869         * We must make sure that we have enough actions for the n-tuple.
9870         */
9871        for (act = ecb->dte_action_last; act != NULL; act = act->dta_prev) {
9872                if (DTRACEACT_ISAGG(act->dta_kind))
9873                        break;
9874
9875                if (--ntuple == 0) {
9876                        /*
9877                         * This is the action with which our n-tuple begins.
9878                         */
9879                        agg->dtag_first = act;
9880                        goto success;
9881                }
9882        }
9883
9884        /*
9885         * This n-tuple is short by ntuple elements.  Return failure.
9886         */
9887        ASSERT(ntuple != 0);
9888 err:
9889        kmem_free(agg, sizeof (dtrace_aggregation_t));
9890        return (NULL);
9891
9892 success:
9893        /*
9894         * If the last action in the tuple has a size of zero, it's actually
9895         * an expression argument for the aggregating action.
```



```
9896         */
9897        ASSERT(ecb->dte_action_last != NULL);
9898        act = ecb->dte_action_last;
9899
9900        if (act->dta_kind == DTRACEACT_DIFEXPR) {
9901                ASSERT(act->dta_difo != NULL);
9902
9903                if (act->dta_difo->dtdo_rtype.dtdt_size == 0)
9904                        agg->dtag_hasarg = 1;
9905        }
9906
9907        /*
9908         * We need to allocate an id for this aggregation.
9909         */
9910        aggid = (dtrace_aggid_t)(uintptr_t)vmem_alloc(state->dts_aggid_arena, 1,
9911            VM_BESTFIT | VM_SLEEP);
9912
9913        if (aggid - 1 >= state->dts_naggregations) {
9914                dtrace_aggregation_t **oaggs = state->dts_aggregations;
9915                dtrace_aggregation_t **aggs;
9916                int naggs = state->dts_naggregations << 1;
9917                int onaggs = state->dts_naggregations;
9918
9919                ASSERT(aggid == state->dts_naggregations + 1);
9920
9921                if (naggs == 0) {
9922                        ASSERT(oaggs == NULL);
9923                        naggs = 1;
9924                }
9925
9926                aggs = kmem_zalloc(naggs * sizeof (*aggs), KM_SLEEP);
9927
9928                if (oaggs != NULL) {
9929                        bcopy(oaggs, aggs, onaggs * sizeof (*aggs));
9930                        kmem_free(oaggs, onaggs * sizeof (*aggs));
9931                }
9932
9933                state->dts_aggregations = aggs;
9934                state->dts_naggregations = naggs;
9935        }
9936
9937        ASSERT(state->dts_aggregations[aggid - 1] == NULL);
9938        state->dts_aggregations[(agg->dtag_id = aggid) - 1] = agg;
9939
9940        frec = &agg->dtag_first->dta_rec;
9941        if (frec->dtrd_alignment < sizeof (dtrace_aggid_t))
9942                frec->dtrd_alignment = sizeof (dtrace_aggid_t);
9943
9944        for (act = agg->dtag_first; act != NULL; act = act->dta_next) {
9945                ASSERT(!act->dta_intuple);
9946                act->dta_intuple = 1;
9947        }
9948
9949        return (&agg->dtag_action);
9950 }
9951
9952 static void
9953 dtrace_ecb_aggregation_destroy(dtrace_ecb_t *ecb, dtrace_action_t *act)
9954 {
9955        dtrace_aggregation_t *agg = (dtrace_aggregation_t *)act;
9956        dtrace_state_t *state = ecb->dte_state;
9957        dtrace_aggid_t aggid = agg->dtag_id;
9958
9959        ASSERT(DTRACEACT_ISAGG(act->dta_kind));
9960        vmem_free(state->dts_aggid_arena, (void *)(uintptr_t)aggid, 1);
```

```
9962         ASSERT(state->dts_aggregations[aggid - 1] == agg);
9963         state->dts_aggregations[aggid - 1] = NULL;

9965         kmem_free(agg, sizeof (dtrace_aggregation_t));
9966 }

9968 static int
9969 dtrace_ecb_action_add(dtrace_ecb_t *ecb, dtrace_actdesc_t *desc)
9970 {
9971         dtrace_action_t *action, *last;
9972         dtrace_difo_t *dp = desc->dtad_difo;
9973         uint32_t size = 0, align = sizeof (uint8_t), mask;
9974         uint16_t format = 0;
9975         dtrace_recdesc_t *rec;
9976         dtrace_state_t *state = ecb->dte_state;
9977         dtrace_optval_t *opt = state->dts_options, nframes, strsize;
9978         uint64_t arg = desc->dtad_arg;

9980         ASSERT(MUTEX_HELD(&dtrace_lock));
9981         ASSERT(ecb->dte_action == NULL || ecb->dte_action->dta_refcnt == 1);

9983         if (DTRACEACT_ISAGG(desc->dtad_kind)) {
9984                 /*
9985                  * If this is an aggregating action, there must be neither
9986                  * a speculate nor a commit on the action chain.
9987                  */
9988                 dtrace_action_t *act;

9990                 for (act = ecb->dte_action; act != NULL; act = act->dta_next) {
9991                         if (act->dta_kind == DTRACEACT_COMMIT)
9992                                 return (EINVAL);

9994                         if (act->dta_kind == DTRACEACT_SPECULATE)
9995                                 return (EINVAL);
9996                 }

9998                 action = dtrace_ecb_aggregation_create(ecb, desc);

10000                 if (action == NULL)
10001                         return (EINVAL);
10002         } else {
10003                 if (DTRACEACT_ISDESTRUCTIVE(desc->dtad_kind) ||
10004                     (desc->dtad_kind == DTRACEACT_DIFEXPR &&
10005                     dp != NULL && dp->dtdo_destructive)) {
10006                         state->dts_destructive = 1;
10007                 }

10009                 switch (desc->dtad_kind) {
10010                 case DTRACEACT_PRINTF:
10011                 case DTRACEACT_PRINTA:
10012                 case DTRACEACT_SYSTEM:
10013                 case DTRACEACT_FREOPEN:
10014                 case DTRACEACT_DIFEXPR:
10015                         /*
10016                          * We know that our arg is a string -- turn it into a
10017                          * format.
10018                          */
10019                         if (arg == NULL) {
10020                                 ASSERT(desc->dtad_kind == DTRACEACT_PRINTA ||
10021                                     desc->dtad_kind == DTRACEACT_DIFEXPR);
10022                                 format = 0;
10023                         } else {
10024                                 ASSERT(arg != NULL);
10025                                 ASSERT(arg > KERNELBASE);
10026                                 format = dtrace_format_add(state,
10027                                     (char *)(uintptr_t)arg);
```



```
10028                         }

10030                         /*FALLTHROUGH*/
10031                 case DTRACEACT_LIBACT:
10032                 case DTRACEACT_TRACEMEM:
10033                 case DTRACEACT_TRACEMEM_DYNSIZE:
10034                         if (dp == NULL)
10035                                 return (EINVAL);

10037                         if ((size = dp->dtdo_rtype.dtdt_size) != 0)
10038                                 break;

10040                         if (dp->dtdo_rtype.dtdt_kind == DIF_TYPE_STRING) {
10041                                 if (!(dp->dtdo_rtype.dtdt_flags & DIF_TF_BYREF))
10042                                         return (EINVAL);

10044                                 size = opt[DTRACEOPT_STRSIZE];
10045                         }

10047                         break;

10049                 case DTRACEACT_STACK:
10050                         if ((nframes = arg) == 0) {
10051                                 nframes = opt[DTRACEOPT_STACKFRAMES];
10052                                 ASSERT(nframes > 0);
10053                                 arg = nframes;
10054                         }

10056                         size = nframes * sizeof (pc_t);
10057                         break;

10059                 case DTRACEACT_JSTACK:
10060                         if ((strsize = DTRACE_USTACK_STRSIZE(arg)) == 0)
10061                                 strsize = opt[DTRACEOPT_JSTACKSTRSIZE];

10063                         if ((nframes = DTRACE_USTACK_NFRAMES(arg)) == 0)
10064                                 nframes = opt[DTRACEOPT_JSTACKFRAMES];

10066                         arg = DTRACE_USTACK_ARG(nframes, strsize);

10068                         /*FALLTHROUGH*/
10069                 case DTRACEACT_USTACK:
10070                         if (desc->dtad_kind != DTRACEACT_JSTACK &&
10071                             (nframes = DTRACE_USTACK_NFRAMES(arg)) == 0) {
10072                                 strsize = DTRACE_USTACK_STRSIZE(arg);
10073                                 nframes = opt[DTRACEOPT_USTACKFRAMES];
10074                                 ASSERT(nframes > 0);
10075                                 arg = DTRACE_USTACK_ARG(nframes, strsize);
10076                         }

10078                         /*
10079                          * Save a slot for the pid.
10080                          */
10081                         size = (nframes + 1) * sizeof (uint64_t);
10082                         size += DTRACE_USTACK_STRSIZE(arg);
10083                         size = P2ROUNDUP(size, (uint32_t)(sizeof (uintptr_t)));

10085                         break;

10087                 case DTRACEACT_SYM:
10088                 case DTRACEACT_MOD:
10089                         if (dp == NULL || ((size = dp->dtdo_rtype.dtdt_size) !=
10090                             sizeof (uint64_t)) ||
10091                             (dp->dtdo_rtype.dtdt_flags & DIF_TF_BYREF))
10092                                 return (EINVAL);
10093                         break;
```

```
10095                 case DTRACEACT_USYM:
10096                 case DTRACEACT_UMOD:
10097                 case DTRACEACT_UADDR:
10098                         if (dp == NULL ||
10099                             (dp->dtdo_rtype.dtdt_size != sizeof (uint64_t)) ||
10100                             (dp->dtdo_rtype.dtdt_flags & DIF_TF_BYREF))
10101                                 return (EINVAL);
10102
10103                         /*
10104                          * We have a slot for the pid, plus a slot for the
10105                          * argument.  To keep things simple (aligned with
10106                          * bitness-neutral sizing), we store each as a 64-bit
10107                          * quantity.
10108                          */
10109                         size = 2 * sizeof (uint64_t);
10110                         break;
10111
10112                 case DTRACEACT_STOP:
10113                 case DTRACEACT_BREAKPOINT:
10114                 case DTRACEACT_PANIC:
10115                         break;
10116
10117                 case DTRACEACT_CHILL:
10118                 case DTRACEACT_DISCARD:
10119                 case DTRACEACT_RAISE:
10120                         if (dp == NULL)
10121                                 return (EINVAL);
10122                         break;
10123
10124                 case DTRACEACT_EXIT:
10125                         if (dp == NULL ||
10126                             (size = dp->dtdo_rtype.dtdt_size) != sizeof (int) ||
10127                             (dp->dtdo_rtype.dtdt_flags & DIF_TF_BYREF))
10128                                 return (EINVAL);
10129                         break;
10130
10131                 case DTRACEACT_SPECULATE:
10132                         if (ecb->dte_size > sizeof (dtrace_rechdr_t))
10133                                 return (EINVAL);
10134
10135                         if (dp == NULL)
10136                                 return (EINVAL);
10137
10138                         state->dts_speculates = 1;
10139                         break;
10140
10141                 case DTRACEACT_COMMIT: {
10142                         dtrace_action_t *act = ecb->dte_action;
10143
10144                         for (; act != NULL; act = act->dta_next) {
10145                                 if (act->dta_kind == DTRACEACT_COMMIT)
10146                                         return (EINVAL);
10147                         }
10148
10149                         if (dp == NULL)
10150                                 return (EINVAL);
10151                         break;
10152                 }
10153
10154                 default:
10155                         return (EINVAL);
10156                 }
10157
10158                 if (size != 0 || desc->dtad_kind == DTRACEACT_SPECULATE) {
10159                         /*
```



```
10160                          * If this is a data-storing action or a speculate,
10161                          * we must be sure that there isn't a commit on the
10162                          * action chain.
10163                          */
10164                         dtrace_action_t *act = ecb->dte_action;
10165
10166                         for (; act != NULL; act = act->dta_next) {
10167                                 if (act->dta_kind == DTRACEACT_COMMIT)
10168                                         return (EINVAL);
10169                         }
10170                 }
10171
10172                 action = kmem_zalloc(sizeof (dtrace_action_t), KM_SLEEP);
10173                 action->dta_rec.dtrd_size = size;
10174         }
10175
10176         action->dta_refcnt = 1;
10177         rec = &action->dta_rec;
10178         size = rec->dtrd_size;
10179
10180         for (mask = sizeof (uint64_t) - 1; size != 0 && mask > 0; mask >>= 1) {
10181                 if (!(size & mask)) {
10182                         align = mask + 1;
10183                         break;
10184                 }
10185         }
10186
10187         action->dta_kind = desc->dtad_kind;
10188
10189         if ((action->dta_difo = dp) != NULL)
10190                 dtrace_difo_hold(dp);
10191
10192         rec->dtrd_action = action->dta_kind;
10193         rec->dtrd_arg = arg;
10194         rec->dtrd_uarg = desc->dtad_uarg;
10195         rec->dtrd_alignment = (uint16_t)align;
10196         rec->dtrd_format = format;
10197
10198         if ((last = ecb->dte_action_last) != NULL) {
10199                 ASSERT(ecb->dte_action != NULL);
10200                 action->dta_prev = last;
10201                 last->dta_next = action;
10202         } else {
10203                 ASSERT(ecb->dte_action == NULL);
10204                 ecb->dte_action = action;
10205         }
10206
10207         ecb->dte_action_last = action;
10208
10209         return (0);
10210 }
10211
10212 static void
10213 dtrace_ecb_action_remove(dtrace_ecb_t *ecb)
10214 {
10215         dtrace_action_t *act = ecb->dte_action, *next;
10216         dtrace_vstate_t *vstate = &ecb->dte_state->dts_vstate;
10217         dtrace_difo_t *dp;
10218         uint16_t format;
10219
10220         if (act != NULL && act->dta_refcnt > 1) {
10221                 ASSERT(act->dta_next == NULL || act->dta_next->dta_refcnt == 1);
10222                 act->dta_refcnt--;
10223         } else {
10224                 for (; act != NULL; act = next) {
10225                         next = act->dta_next;
```

```
10226                         ASSERT(next != NULL || act == ecb->dte_action_last);
10227                         ASSERT(act->dta_refcnt == 1);

10229                         if ((format = act->dta_rec.dtrd_format) != 0)
10230                                 dtrace_format_remove(ecb->dte_state, format);

10232                         if ((dp = act->dta_difo) != NULL)
10233                                 dtrace_difo_release(dp, vstate);

10235                         if (DTRACEACT_ISAGG(act->dta_kind)) {
10236                                 dtrace_ecb_aggregation_destroy(ecb, act);
10237                         } else {
10238                                 kmem_free(act, sizeof (dtrace_action_t));
10239                         }
10240                 }
10241         }

10243         ecb->dte_action = NULL;
10244         ecb->dte_action_last = NULL;
10245         ecb->dte_size = 0;
10246 }

10248 static void
10249 dtrace_ecb_disable(dtrace_ecb_t *ecb)
10250 {
10251         /*
10252          * We disable the ECB by removing it from its probe.
10253          */
10254         dtrace_ecb_t *pecb, *prev = NULL;
10255         dtrace_probe_t *probe = ecb->dte_probe;

10257         ASSERT(MUTEX_HELD(&dtrace_lock));

10259         if (probe == NULL) {
10260                 /*
10261                  * This is the NULL probe; there is nothing to disable.
10262                  */
10263                 return;
10264         }

10266         for (pecb = probe->dtpr_ecb; pecb != NULL; pecb = pecb->dte_next) {
10267                 if (pecb == ecb)
10268                         break;
10269                 prev = pecb;
10270         }

10272         ASSERT(pecb != NULL);

10274         if (prev == NULL) {
10275                 probe->dtpr_ecb = ecb->dte_next;
10276         } else {
10277                 prev->dte_next = ecb->dte_next;
10278         }

10280         if (ecb == probe->dtpr_ecb_last) {
10281                 ASSERT(ecb->dte_next == NULL);
10282                 probe->dtpr_ecb_last = prev;
10283         }

10285         /*
10286          * The ECB has been disconnected from the probe; now sync to assure
10287          * that all CPUs have seen the change before returning.
10288          */
10289         dtrace_sync();

10291         if (probe->dtpr_ecb == NULL) {
```



```
10292                 /*
10293                  * That was the last ECB on the probe; clear the predicate
10294                  * cache ID for the probe, disable it and sync one more time
10295                  * to assure that we'll never hit it again.
10296                  */
10297                 dtrace_provider_t *prov = probe->dtpr_provider;

10299                 ASSERT(ecb->dte_next == NULL);
10300                 ASSERT(probe->dtpr_ecb_last == NULL);
10301                 probe->dtpr_predcache = DTRACE_CACHEIDNONE;
10302                 prov->dtpv_pops.dtps_disable(prov->dtpv_arg,
10303                     probe->dtpr_id, probe->dtpr_arg);
10304                 dtrace_sync();
10305         } else {
10306                 /*
10307                  * There is at least one ECB remaining on the probe.  If there
10308                  * is _exactly_ one, set the probe's predicate cache ID to be
10309                  * the predicate cache ID of the remaining ECB.
10310                  */
10311                 ASSERT(probe->dtpr_ecb_last != NULL);
10312                 ASSERT(probe->dtpr_predcache == DTRACE_CACHEIDNONE);

10314                 if (probe->dtpr_ecb == probe->dtpr_ecb_last) {
10315                         dtrace_predicate_t *p = probe->dtpr_ecb->dte_predicate;

10317                         ASSERT(probe->dtpr_ecb->dte_next == NULL);

10319                         if (p != NULL)
10320                                 probe->dtpr_predcache = p->dtp_cacheid;
10321                 }

10323                 ecb->dte_next = NULL;
10324         }
10325 }

10327 static void
10328 dtrace_ecb_destroy(dtrace_ecb_t *ecb)
10329 {
10330         dtrace_state_t *state = ecb->dte_state;
10331         dtrace_vstate_t *vstate = &state->dts_vstate;
10332         dtrace_predicate_t *pred;
10333         dtrace_epid_t epid = ecb->dte_epid;

10335         ASSERT(MUTEX_HELD(&dtrace_lock));
10336         ASSERT(ecb->dte_next == NULL);
10337         ASSERT(ecb->dte_probe == NULL || ecb->dte_probe->dtpr_ecb != ecb);

10339         if ((pred = ecb->dte_predicate) != NULL)
10340                 dtrace_predicate_release(pred, vstate);

10342         dtrace_ecb_action_remove(ecb);

10344         ASSERT(state->dts_ecbs[epid - 1] == ecb);
10345         state->dts_ecbs[epid - 1] = NULL;

10347         kmem_free(ecb, sizeof (dtrace_ecb_t));
10348 }

10350 static dtrace_ecb_t *
10351 dtrace_ecb_create(dtrace_state_t *state, dtrace_probe_t *probe,
10352     dtrace_enabling_t *enab)
10353 {
10354         dtrace_ecb_t *ecb;
10355         dtrace_predicate_t *pred;
10356         dtrace_actdesc_t *act;
10357         dtrace_provider_t *prov;
```

```
10358         dtrace_ecbdesc_t *desc = enab->dten_current;

10360         ASSERT(MUTEX_HELD(&dtrace_lock));
10361         ASSERT(state != NULL);

10363         ecb = dtrace_ecb_add(state, probe);
10364         ecb->dte_uarg = desc->dted_uarg;

10366         if ((pred = desc->dted_pred.dtpdd_predicate) != NULL) {
10367                 dtrace_predicate_hold(pred);
10368                 ecb->dte_predicate = pred;
10369         }

10371         if (probe != NULL) {
10372                 /*
10373                  * If the provider shows more leg than the consumer is old
10374                  * enough to see, we need to enable the appropriate implicit
10375                  * predicate bits to prevent the ecb from activating at
10376                  * revealing times.
10377                  *
10378                  * Providers specifying DTRACE_PRIV_USER at register time
10379                  * are stating that they need the /proc-style privilege
10380                  * model to be enforced, and this is what DTRACE_COND_OWNER
10381                  * and DTRACE_COND_ZONEOWNER will then do at probe time.
10382                  */
10383                 prov = probe->dtpr_provider;
10384                 if (!(state->dts_cred.dcr_visible & DTRACE_CRV_ALLPROC) &&
10385                     (prov->dtpv_priv.dtpp_flags & DTRACE_PRIV_USER))
10386                         ecb->dte_cond |= DTRACE_COND_OWNER;

10388                 if (!(state->dts_cred.dcr_visible & DTRACE_CRV_ALLZONE) &&
10389                     (prov->dtpv_priv.dtpp_flags & DTRACE_PRIV_USER))
10390                         ecb->dte_cond |= DTRACE_COND_ZONEOWNER;

10392                 /*
10393                  * If the provider shows us kernel innards and the user
10394                  * is lacking sufficient privilege, enable the
10395                  * DTRACE_COND_USERMODE implicit predicate.
10396                  */
10397                 if (!(state->dts_cred.dcr_visible & DTRACE_CRV_KERNEL) &&
10398                     (prov->dtpv_priv.dtpp_flags & DTRACE_PRIV_KERNEL))
10399                         ecb->dte_cond |= DTRACE_COND_USERMODE;
10400         }

10402         if (dtrace_ecb_create_cache != NULL) {
10403                 /*
10404                  * If we have a cached ecb, we'll use its action list instead
10405                  * of creating our own (saving both time and space).
10406                  */
10407                 dtrace_ecb_t *cached = dtrace_ecb_create_cache;
10408                 dtrace_action_t *act = cached->dte_action;

10410                 if (act != NULL) {
10411                         ASSERT(act->dta_refcnt > 0);
10412                         act->dta_refcnt++;
10413                         ecb->dte_action = act;
10414                         ecb->dte_action_last = cached->dte_action_last;
10415                         ecb->dte_needed = cached->dte_needed;
10416                         ecb->dte_size = cached->dte_size;
10417                         ecb->dte_alignment = cached->dte_alignment;
10418                 }

10420                 return (ecb);
10421         }

10423         for (act = desc->dted_action; act != NULL; act = act->dtad_next) {
```



```
10424                 if ((enab->dten_error = dtrace_ecb_action_add(ecb, act)) != 0) {
10425                         dtrace_ecb_destroy(ecb);
10426                         return (NULL);
10427                 }
10428         }

10430         dtrace_ecb_resize(ecb);

10432         return (dtrace_ecb_create_cache = ecb);
10433 }

10435 static int
10436 dtrace_ecb_create_enable(dtrace_probe_t *probe, void *arg)
10437 {
10438         dtrace_ecb_t *ecb;
10439         dtrace_enabling_t *enab = arg;
10440         dtrace_state_t *state = enab->dten_vstate->dtvs_state;

10442         ASSERT(state != NULL);

10444         if (probe != NULL && probe->dtpr_gen < enab->dten_probegen) {
10445                 /*
10446                  * This probe was created in a generation for which this
10447                  * enabling has previously created ECBs; we don't want to
10448                  * enable it again, so just kick out.
10449                  */
10450                 return (DTRACE_MATCH_NEXT);
10451         }

10453         if ((ecb = dtrace_ecb_create(state, probe, enab)) == NULL)
10454                 return (DTRACE_MATCH_DONE);

10456         if (dtrace_ecb_enable(ecb) < 0)
10457                 return (DTRACE_MATCH_FAIL);

10459         return (DTRACE_MATCH_NEXT);
10460 }

10462 static dtrace_ecb_t *
10463 dtrace_epid2ecb(dtrace_state_t *state, dtrace_epid_t id)
10464 {
10465         dtrace_ecb_t *ecb;

10467         ASSERT(MUTEX_HELD(&dtrace_lock));

10469         if (id == 0 || id > state->dts_necbs)
10470                 return (NULL);

10472         ASSERT(state->dts_necbs > 0 && state->dts_ecbs != NULL);
10473         ASSERT((ecb = state->dts_ecbs[id - 1]) == NULL || ecb->dte_epid == id);

10475         return (state->dts_ecbs[id - 1]);
10476 }

10478 static dtrace_aggregation_t *
10479 dtrace_aggid2agg(dtrace_state_t *state, dtrace_aggid_t id)
10480 {
10481         dtrace_aggregation_t *agg;

10483         ASSERT(MUTEX_HELD(&dtrace_lock));

10485         if (id == 0 || id > state->dts_naggregations)
10486                 return (NULL);

10488         ASSERT(state->dts_naggregations > 0 && state->dts_aggregations != NULL);
10489         ASSERT((agg = state->dts_aggregations[id - 1]) == NULL ||
```

```
10490             agg->dtag_id == id);

10492         return (state->dts_aggregations[id - 1]);
10493 }

10495 /*
10496  * DTrace Buffer Functions
10497  *
10498  * The following functions manipulate DTrace buffers.  Most of these functions
10499  * are called in the context of establishing or processing consumer state;
10500  * exceptions are explicitly noted.
10501  */

10503 /*
10504  * Note:  called from cross call context.  This function switches the two
10505  * buffers on a given CPU.  The atomicity of this operation is assured by
10506  * disabling interrupts while the actual switch takes place; the disabling of
10507  * interrupts serializes the execution with any execution of dtrace_probe() on
10508  * the same CPU.
10509  */
10510 static void
10511 dtrace_buffer_switch(dtrace_buffer_t *buf)
10512 {
10513         caddr_t tomax = buf->dtb_tomax;
10514         caddr_t xamot = buf->dtb_xamot;
10515         dtrace_icookie_t cookie;
10516         hrtime_t now;

10518         ASSERT(!(buf->dtb_flags & DTRACEBUF_NOSWITCH));
10519         ASSERT(!(buf->dtb_flags & DTRACEBUF_RING));

10521         cookie = dtrace_interrupt_disable();
10522         now = dtrace_gethrtime();
10523         buf->dtb_tomax = xamot;
10524         buf->dtb_xamot = tomax;
10525         buf->dtb_xamot_drops = buf->dtb_drops;
10526         buf->dtb_xamot_offset = buf->dtb_offset;
10527         buf->dtb_xamot_errors = buf->dtb_errors;
10528         buf->dtb_xamot_flags = buf->dtb_flags;
10529         buf->dtb_offset = 0;
10530         buf->dtb_drops = 0;
10531         buf->dtb_errors = 0;
10532         buf->dtb_flags &= ~(DTRACEBUF_ERROR | DTRACEBUF_DROPPED);
10533         buf->dtb_interval = now - buf->dtb_switched;
10534         buf->dtb_switched = now;
10535         dtrace_interrupt_enable(cookie);
10536 }

10538 /*
10539  * Note:  called from cross call context.  This function activates a buffer
10540  * on a CPU.  As with dtrace_buffer_switch(), the atomicity of the operation
10541  * is guaranteed by the disabling of interrupts.
10542  */
10543 static void
10544 dtrace_buffer_activate(dtrace_state_t *state)
10545 {
10546         dtrace_buffer_t *buf;
10547         dtrace_icookie_t cookie = dtrace_interrupt_disable();

10549         buf = &state->dts_buffer[CPU->cpu_id];

10551         if (buf->dtb_tomax != NULL) {
10552                 /*
10553                  * We might like to assert that the buffer is marked inactive,
10554                  * but this isn't necessarily true:  the buffer for the CPU
10555                  * that processes the BEGIN probe has its buffer activated
```



```
10556                  * manually.  In this case, we take the (harmless) action
10557                  * re-clearing the bit INACTIVE bit.
10558                  */
10559                 buf->dtb_flags &= ~DTRACEBUF_INACTIVE;
10560         }

10562         dtrace_interrupt_enable(cookie);
10563 }

10565 static int
10566 dtrace_buffer_alloc(dtrace_buffer_t *bufs, size_t size, int flags,
10567     processorid_t cpu, int *factor)
10568 {
10569         cpu_t *cp;
10570         dtrace_buffer_t *buf;
10571         int allocated = 0, desired = 0;

10573         ASSERT(MUTEX_HELD(&cpu_lock));
10574         ASSERT(MUTEX_HELD(&dtrace_lock));

10576         *factor = 1;

10578         if (size > dtrace_nonroot_maxsize &&
10579             !PRIV_POLICY_CHOICE(CRED(), PRIV_ALL, B_FALSE))
10580                 return (EFBIG);

10582         cp = cpu_list;

10584         do {
10585                 if (cpu != DTRACE_CPUALL && cpu != cp->cpu_id)
10586                         continue;

10588                 buf = &bufs[cp->cpu_id];

10590                 /*
10591                  * If there is already a buffer allocated for this CPU, it
10592                  * is only possible that this is a DR event.  In this case,
10593                  * the buffer size must match our specified size.
10594                  */
10595                 if (buf->dtb_tomax != NULL) {
10596                         ASSERT(buf->dtb_size == size);
10597                         continue;
10598                 }

10600                 ASSERT(buf->dtb_xamot == NULL);

10602                 if ((buf->dtb_tomax = kmem_zalloc(size,
10603                     KM_NOSLEEP | KM_NORMALPRI)) == NULL)
10604                         goto err;

10606                 buf->dtb_size = size;
10607                 buf->dtb_flags = flags;
10608                 buf->dtb_offset = 0;
10609                 buf->dtb_drops = 0;

10611                 if (flags & DTRACEBUF_NOSWITCH)
10612                         continue;

10614                 if ((buf->dtb_xamot = kmem_zalloc(size,
10615                     KM_NOSLEEP | KM_NORMALPRI)) == NULL)
10616                         goto err;
10617         } while ((cp = cp->cpu_next) != cpu_list);

10619         return (0);

10621 err:
```

```
10622         cp = cpu_list;

10624         do {
10625                 if (cpu != DTRACE_CPUALL && cpu != cp->cpu_id)
10626                         continue;

10628                 buf = &bufs[cp->cpu_id];
10629                 desired += 2;

10631                 if (buf->dtb_xamot != NULL) {
10632                         ASSERT(buf->dtb_tomax != NULL);
10633                         ASSERT(buf->dtb_size == size);
10634                         kmem_free(buf->dtb_xamot, size);
10635                         allocated++;
10636                 }

10638                 if (buf->dtb_tomax != NULL) {
10639                         ASSERT(buf->dtb_size == size);
10640                         kmem_free(buf->dtb_tomax, size);
10641                         allocated++;
10642                 }

10644                 buf->dtb_tomax = NULL;
10645                 buf->dtb_xamot = NULL;
10646                 buf->dtb_size = 0;
10647         } while ((cp = cp->cpu_next) != cpu_list);

10649         *factor = desired / (allocated > 0 ? allocated : 1);

10651         return (ENOMEM);
10652 }

10654 /*
10655  * Note:  called from probe context.  This function just increments the drop
10656  * count on a buffer.  It has been made a function to allow for the
10657  * possibility of understanding the source of mysterious drop counts.  (A
10658  * problem for which one may be particularly disappointed that DTrace cannot
10659  * be used to understand DTrace.)
10660  */
10661 static void
10662 dtrace_buffer_drop(dtrace_buffer_t *buf)
10663 {
10664         buf->dtb_drops++;
10665 }

10667 /*
10668  * Note:  called from probe context.  This function is called to reserve space
10669  * in a buffer.  If mstate is non-NULL, sets the scratch base and size in the
10670  * mstate.  Returns the new offset in the buffer, or a negative value if an
10671  * error has occurred.
10672  */
10673 static intptr_t
10674 dtrace_buffer_reserve(dtrace_buffer_t *buf, size_t needed, size_t align,
10675     dtrace_state_t *state, dtrace_mstate_t *mstate)
10676 {
10677         intptr_t offs = buf->dtb_offset, soffs;
10678         intptr_t woffs;
10679         caddr_t tomax;
10680         size_t total;

10682         if (buf->dtb_flags & DTRACEBUF_INACTIVE)
10683                 return (-1);

10685         if ((tomax = buf->dtb_tomax) == NULL) {
10686                 dtrace_buffer_drop(buf);
10687                 return (-1);
```



```
10688         }

10690         if (!(buf->dtb_flags & (DTRACEBUF_RING | DTRACEBUF_FILL))) {
10691                 while (offs & (align - 1)) {
10692                         /*
10693                          * Assert that our alignment is off by a number which
10694                          * is itself sizeof (uint32_t) aligned.
10695                          */
10696                         ASSERT(!((align - (offs & (align - 1))) &
10697                             (sizeof (uint32_t) - 1)));
10698                         DTRACE_STORE(uint32_t, tomax, offs, DTRACE_EPIDNONE);
10699                         offs += sizeof (uint32_t);
10700                 }

10702                 if ((soffs = offs + needed) > buf->dtb_size) {
10703                         dtrace_buffer_drop(buf);
10704                         return (-1);
10705                 }

10707                 if (mstate == NULL)
10708                         return (offs);

10710                 mstate->dtms_scratch_base = (uintptr_t)tomax + soffs;
10711                 mstate->dtms_scratch_size = buf->dtb_size - soffs;
10712                 mstate->dtms_scratch_ptr = mstate->dtms_scratch_base;

10714                 return (offs);
10715         }

10717         if (buf->dtb_flags & DTRACEBUF_FILL) {
10718                 if (state->dts_activity != DTRACE_ACTIVITY_COOLDOWN &&
10719                     (buf->dtb_flags & DTRACEBUF_FULL))
10720                         return (-1);
10721                 goto out;
10722         }

10724         total = needed + (offs & (align - 1));

10726         /*
10727          * For a ring buffer, life is quite a bit more complicated.  Before
10728          * we can store any padding, we need to adjust our wrapping offset.
10729          * (If we've never before wrapped or we're not about to, no adjustment
10730          * is required.)
10731          */
10732         if ((buf->dtb_flags & DTRACEBUF_WRAPPED) ||
10733             offs + total > buf->dtb_size) {
10734                 woffs = buf->dtb_xamot_offset;

10736                 if (offs + total > buf->dtb_size) {
10737                         /*
10738                          * We can't fit in the end of the buffer.  First, a
10739                          * sanity check that we can fit in the buffer at all.
10740                          */
10741                         if (total > buf->dtb_size) {
10742                                 dtrace_buffer_drop(buf);
10743                                 return (-1);
10744                         }

10746                         /*
10747                          * We're going to be storing at the top of the buffer,
10748                          * so now we need to deal with the wrapped offset.  We
10749                          * only reset our wrapped offset to 0 if it is
10750                          * currently greater than the current offset.  If it
10751                          * is less than the current offset, it is because a
10752                          * previous allocation induced a wrap -- but the
10753                          * allocation didn't subsequently take the space due
```

```
10754                          * to an error or false predicate evaluation.  In this
10755                          * case, we'll just leave the wrapped offset alone: if
10756                          * the wrapped offset hasn't been advanced far enough
10757                          * for this allocation, it will be adjusted in the
10758                          * lower loop.
10759                          */
10760                         if (buf->dtb_flags & DTRACEBUF_WRAPPED) {
10761                                 if (woffs >= offs)
10762                                         woffs = 0;
10763                         } else {
10764                                 woffs = 0;
10765                         }
10766
10767                         /*
10768                          * Now we know that we're going to be storing to the
10769                          * top of the buffer and that there is room for us
10770                          * there.  We need to clear the buffer from the current
10771                          * offset to the end (there may be old gunk there).
10772                          */
10773                         while (offs < buf->dtb_size)
10774                                 tomax[offs++] = 0;
10775
10776                         /*
10777                          * We need to set our offset to zero.  And because we
10778                          * are wrapping, we need to set the bit indicating as
10779                          * much.  We can also adjust our needed space back
10780                          * down to the space required by the ECB -- we know
10781                          * that the top of the buffer is aligned.
10782                          */
10783                         offs = 0;
10784                         total = needed;
10785                         buf->dtb_flags |= DTRACEBUF_WRAPPED;
10786                 } else {
10787                         /*
10788                          * There is room for us in the buffer, so we simply
10789                          * need to check the wrapped offset.
10790                          */
10791                         if (woffs < offs) {
10792                                 /*
10793                                  * The wrapped offset is less than the offset.
10794                                  * This can happen if we allocated buffer space
10795                                  * that induced a wrap, but then we didn't
10796                                  * subsequently take the space due to an error
10797                                  * or false predicate evaluation.  This is
10798                                  * okay; we know that _this_ allocation isn't
10799                                  * going to induce a wrap.  We still can't
10800                                  * reset the wrapped offset to be zero,
10801                                  * however: the space may have been trashed in
10802                                  * the previous failed probe attempt.  But at
10803                                  * least the wrapped offset doesn't need to
10804                                  * be adjusted at all...
10805                                  */
10806                                 goto out;
10807                         }
10808                 }
10809
10810                 while (offs + total > woffs) {
10811                         dtrace_epid_t epid = *(uint32_t *)(tomax + woffs);
10812                         size_t size;
10813
10814                         if (epid == DTRACE_EPIDNONE) {
10815                                 size = sizeof (uint32_t);
10816                         } else {
10817                                 ASSERT3U(epid, <=, state->dts_necbs);
10818                                 ASSERT(state->dts_ecbs[epid - 1] != NULL);
```



```
10820                                 size = state->dts_ecbs[epid - 1]->dte_size;
10821                         }
10822
10823                         ASSERT(woffs + size <= buf->dtb_size);
10824                         ASSERT(size != 0);
10825
10826                         if (woffs + size == buf->dtb_size) {
10827                                 /*
10828                                  * We've reached the end of the buffer; we want
10829                                  * to set the wrapped offset to 0 and break
10830                                  * out.  However, if the offs is 0, then we're
10831                                  * in a strange edge-condition:  the amount of
10832                                  * space that we want to reserve plus the size
10833                                  * of the record that we're overwriting is
10834                                  * greater than the size of the buffer.  This
10835                                  * is problematic because if we reserve the
10836                                  * space but subsequently don't consume it (due
10837                                  * to a failed predicate or error) the wrapped
10838                                  * offset will be 0 -- yet the EPID at offset 0
10839                                  * will not be committed.  This situation is
10840                                  * relatively easy to deal with:  if we're in
10841                                  * this case, the buffer is indistinguishable
10842                                  * from one that hasn't wrapped; we need only
10843                                  * finish the job by clearing the wrapped bit,
10844                                  * explicitly setting the offset to be 0, and
10845                                  * zero'ing out the old data in the buffer.
10846                                  */
10847                                 if (offs == 0) {
10848                                         buf->dtb_flags &= ~DTRACEBUF_WRAPPED;
10849                                         buf->dtb_offset = 0;
10850                                         woffs = total;
10851
10852                                         while (woffs < buf->dtb_size)
10853                                                 tomax[woffs++] = 0;
10854                                 }
10855
10856                                 woffs = 0;
10857                                 break;
10858                         }
10859
10860                         woffs += size;
10861                 }
10862
10863                 /*
10864                  * We have a wrapped offset.  It may be that the wrapped offset
10865                  * has become zero -- that's okay.
10866                  */
10867                 buf->dtb_xamot_offset = woffs;
10868         }
10869
10870 out:
10871         /*
10872          * Now we can plow the buffer with any necessary padding.
10873          */
10874         while (offs & (align - 1)) {
10875                 /*
10876                  * Assert that our alignment is off by a number which
10877                  * is itself sizeof (uint32_t) aligned.
10878                  */
10879                 ASSERT(!((align - (offs & (align - 1))) &
10880                     (sizeof (uint32_t) - 1)));
10881                 DTRACE_STORE(uint32_t, tomax, offs, DTRACE_EPIDNONE);
10882                 offs += sizeof (uint32_t);
10883         }
10884
10885         if (buf->dtb_flags & DTRACEBUF_FILL) {
```

```
10886                 if (offs + needed > buf->dtb_size - state->dts_reserve) {
10887                         buf->dtb_flags |= DTRACEBUF_FULL;
10888                         return (-1);
10889                 }
10890         }

10892         if (mstate == NULL)
10893                 return (offs);

10895         /*
10896          * For ring buffers and fill buffers, the scratch space is always
10897          * the inactive buffer.
10898          */
10899         mstate->dtms_scratch_base = (uintptr_t)buf->dtb_xamot;
10900         mstate->dtms_scratch_size = buf->dtb_size;
10901         mstate->dtms_scratch_ptr = mstate->dtms_scratch_base;

10903         return (offs);
10904 }

10906 static void
10907 dtrace_buffer_polish(dtrace_buffer_t *buf)
10908 {
10909         ASSERT(buf->dtb_flags & DTRACEBUF_RING);
10910         ASSERT(MUTEX_HELD(&dtrace_lock));

10912         if (!(buf->dtb_flags & DTRACEBUF_WRAPPED))
10913                 return;

10915         /*
10916          * We need to polish the ring buffer.  There are three cases:
10917          *
10918          * - The first (and presumably most common) is that there is no gap
10919          *   between the buffer offset and the wrapped offset.  In this case,
10920          *   there is nothing in the buffer that isn't valid data; we can
10921          *   mark the buffer as polished and return.
10922          *
10923          * - The second (less common than the first but still more common
10924          *   than the third) is that there is a gap between the buffer offset
10925          *   and the wrapped offset, and the wrapped offset is larger than the
10926          *   buffer offset.  This can happen because of an alignment issue, or
10927          *   can happen because of a call to dtrace_buffer_reserve() that
10928          *   didn't subsequently consume the buffer space.  In this case,
10929          *   we need to zero the data from the buffer offset to the wrapped
10930          *   offset.
10931          *
10932          * - The third (and least common) is that there is a gap between the
10933          *   buffer offset and the wrapped offset, but the wrapped offset is
10934          *   _less_ than the buffer offset.  This can only happen because a
10935          *   call to dtrace_buffer_reserve() induced a wrap, but the space
10936          *   was not subsequently consumed.  In this case, we need to zero the
10937          *   space from the offset to the end of the buffer _and_ from the
10938          *   top of the buffer to the wrapped offset.
10939          */
10940         if (buf->dtb_offset < buf->dtb_xamot_offset) {
10941                 bzero(buf->dtb_tomax + buf->dtb_offset,
10942                     buf->dtb_xamot_offset - buf->dtb_offset);
10943         }

10945         if (buf->dtb_offset > buf->dtb_xamot_offset) {
10946                 bzero(buf->dtb_tomax + buf->dtb_offset,
10947                     buf->dtb_size - buf->dtb_offset);
10948                 bzero(buf->dtb_tomax, buf->dtb_xamot_offset);
10949         }
10950 }
```



```
10952 /*
10953  * This routine determines if data generated at the specified time has likely
10954  * been entirely consumed at user-level.  This routine is called to determine
10955  * if an ECB on a defunct probe (but for an active enabling) can be safely
10956  * disabled and destroyed.
10957  */
10958 static int
10959 dtrace_buffer_consumed(dtrace_buffer_t *bufs, hrtime_t when)
10960 {
10961         int i;

10963         for (i = 0; i < NCPU; i++) {
10964                 dtrace_buffer_t *buf = &bufs[i];

10966                 if (buf->dtb_size == 0)
10967                         continue;

10969                 if (buf->dtb_flags & DTRACEBUF_RING)
10970                         return (0);

10972                 if (!buf->dtb_switched && buf->dtb_offset != 0)
10973                         return (0);

10975                 if (buf->dtb_switched - buf->dtb_interval < when)
10976                         return (0);
10977         }

10979         return (1);
10980 }

10982 static void
10983 dtrace_buffer_free(dtrace_buffer_t *bufs)
10984 {
10985         int i;

10987         for (i = 0; i < NCPU; i++) {
10988                 dtrace_buffer_t *buf = &bufs[i];

10990                 if (buf->dtb_tomax == NULL) {
10991                         ASSERT(buf->dtb_xamot == NULL);
10992                         ASSERT(buf->dtb_size == 0);
10993                         continue;
10994                 }

10996                 if (buf->dtb_xamot != NULL) {
10997                         ASSERT(!(buf->dtb_flags & DTRACEBUF_NOSWITCH));
10998                         kmem_free(buf->dtb_xamot, buf->dtb_size);
10999                 }

11001                 kmem_free(buf->dtb_tomax, buf->dtb_size);
11002                 buf->dtb_size = 0;
11003                 buf->dtb_tomax = NULL;
11004                 buf->dtb_xamot = NULL;
11005         }
11006 }

11008 /*
11009  * DTrace Enabling Functions
11010  */
11011 static dtrace_enabling_t *
11012 dtrace_enabling_create(dtrace_vstate_t *vstate)
11013 {
11014         dtrace_enabling_t *enab;

11016         enab = kmem_zalloc(sizeof (dtrace_enabling_t), KM_SLEEP);
11017         enab->dten_vstate = vstate;
```

```
11019         return (enab);
11020 }

11022 static void
11023 dtrace_enabling_add(dtrace_enabling_t *enab, dtrace_ecbdesc_t *ecb)
11024 {
11025         dtrace_ecbdesc_t **ndesc;
11026         size_t osize, nsize;

11028         /*
11029          * We can't add to enablings after we've enabled them, or after we've
11030          * retained them.
11031          */
11032         ASSERT(enab->dten_probegen == 0);
11033         ASSERT(enab->dten_next == NULL && enab->dten_prev == NULL);

11035         if (enab->dten_ndesc < enab->dten_maxdesc) {
11036                 enab->dten_desc[enab->dten_ndesc++] = ecb;
11037                 return;
11038         }

11040         osize = enab->dten_maxdesc * sizeof (dtrace_enabling_t *);

11042         if (enab->dten_maxdesc == 0) {
11043                 enab->dten_maxdesc = 1;
11044         } else {
11045                 enab->dten_maxdesc <<= 1;
11046         }

11048         ASSERT(enab->dten_ndesc < enab->dten_maxdesc);

11050         nsize = enab->dten_maxdesc * sizeof (dtrace_enabling_t *);
11051         ndesc = kmem_zalloc(nsize, KM_SLEEP);
11052         bcopy(enab->dten_desc, ndesc, osize);
11053         kmem_free(enab->dten_desc, osize);

11055         enab->dten_desc = ndesc;
11056         enab->dten_desc[enab->dten_ndesc++] = ecb;
11057 }

11059 static void
11060 dtrace_enabling_addlike(dtrace_enabling_t *enab, dtrace_ecbdesc_t *ecb,
11061     dtrace_probedesc_t *pd)
11062 {
11063         dtrace_ecbdesc_t *new;
11064         dtrace_predicate_t *pred;
11065         dtrace_actdesc_t *act;

11067         /*
11068          * We're going to create a new ECB description that matches the
11069          * specified ECB in every way, but has the specified probe description.
11070          */
11071         new = kmem_zalloc(sizeof (dtrace_ecbdesc_t), KM_SLEEP);

11073         if ((pred = ecb->dted_pred.dtpdd_predicate) != NULL)
11074                 dtrace_predicate_hold(pred);

11076         for (act = ecb->dted_action; act != NULL; act = act->dtad_next)
11077                 dtrace_actdesc_hold(act);

11079         new->dted_action = ecb->dted_action;
11080         new->dted_pred = ecb->dted_pred;
11081         new->dted_probe = *pd;
11082         new->dted_uarg = ecb->dted_uarg;
```



```
11084         dtrace_enabling_add(enab, new);
11085 }

11087 static void
11088 dtrace_enabling_dump(dtrace_enabling_t *enab)
11089 {
11090         int i;

11092         for (i = 0; i < enab->dten_ndesc; i++) {
11093                 dtrace_probedesc_t *desc = &enab->dten_desc[i]->dted_probe;

11095                 cmn_err(CE_NOTE, "enabling probe %d (%s:%s:%s:%s)", i,
11096                     desc->dtpd_provider, desc->dtpd_mod,
11097                     desc->dtpd_func, desc->dtpd_name);
11098         }
11099 }

11101 static void
11102 dtrace_enabling_destroy(dtrace_enabling_t *enab)
11103 {
11104         int i;
11105         dtrace_ecbdesc_t *ep;
11106         dtrace_vstate_t *vstate = enab->dten_vstate;

11108         ASSERT(MUTEX_HELD(&dtrace_lock));

11110         for (i = 0; i < enab->dten_ndesc; i++) {
11111                 dtrace_actdesc_t *act, *next;
11112                 dtrace_predicate_t *pred;

11114                 ep = enab->dten_desc[i];

11116                 if ((pred = ep->dted_pred.dtpdd_predicate) != NULL)
11117                         dtrace_predicate_release(pred, vstate);

11119                 for (act = ep->dted_action; act != NULL; act = next) {
11120                         next = act->dtad_next;
11121                         dtrace_actdesc_release(act, vstate);
11122                 }

11124                 kmem_free(ep, sizeof (dtrace_ecbdesc_t));
11125         }

11127         kmem_free(enab->dten_desc,
11128             enab->dten_maxdesc * sizeof (dtrace_enabling_t *));

11130         /*
11131          * If this was a retained enabling, decrement the dts_nretained count
11132          * and take it off of the dtrace_retained list.
11133          */
11134         if (enab->dten_prev != NULL || enab->dten_next != NULL ||
11135             dtrace_retained == enab) {
11136                 ASSERT(enab->dten_vstate->dtvs_state != NULL);
11137                 ASSERT(enab->dten_vstate->dtvs_state->dts_nretained > 0);
11138                 enab->dten_vstate->dtvs_state->dts_nretained--;
11139                 dtrace_retained_gen++;
11140         }

11142         if (enab->dten_prev == NULL) {
11143                 if (dtrace_retained == enab) {
11144                         dtrace_retained = enab->dten_next;

11146                         if (dtrace_retained != NULL)
11147                                 dtrace_retained->dten_prev = NULL;
11148                 }
11149         } else {
```

```
11150                   ASSERT(enab != dtrace_retained);
11151                   ASSERT(dtrace_retained != NULL);
11152                   enab->dten_prev->dten_next = enab->dten_next;
11153           }

11155           if (enab->dten_next != NULL) {
11156                   ASSERT(dtrace_retained != NULL);
11157                   enab->dten_next->dten_prev = enab->dten_prev;
11158           }

11160           kmem_free(enab, sizeof (dtrace_enabling_t));
11161   }

11163   static int
11164   dtrace_enabling_retain(dtrace_enabling_t *enab)
11165   {
11166           dtrace_state_t *state;

11168           ASSERT(MUTEX_HELD(&dtrace_lock));
11169           ASSERT(enab->dten_next == NULL && enab->dten_prev == NULL);
11170           ASSERT(enab->dten_vstate != NULL);

11172           state = enab->dten_vstate->dtvs_state;
11173           ASSERT(state != NULL);

11175           /*
11176            * We only allow each state to retain dtrace_retain_max enablings.
11177            */
11178           if (state->dts_nretained >= dtrace_retain_max)
11179                   return (ENOSPC);

11181           state->dts_nretained++;
11182           dtrace_retained_gen++;

11184           if (dtrace_retained == NULL) {
11185                   dtrace_retained = enab;
11186                   return (0);
11187           }

11189           enab->dten_next = dtrace_retained;
11190           dtrace_retained->dten_prev = enab;
11191           dtrace_retained = enab;

11193           return (0);
11194   }

11196   static int
11197   dtrace_enabling_replicate(dtrace_state_t *state, dtrace_probedesc_t *match,
11198       dtrace_probedesc_t *create)
11199   {
11200           dtrace_enabling_t *new, *enab;
11201           int found = 0, err = ENOENT;

11203           ASSERT(MUTEX_HELD(&dtrace_lock));
11204           ASSERT(strlen(match->dtpd_provider) < DTRACE_PROVNAMELEN);
11205           ASSERT(strlen(match->dtpd_mod) < DTRACE_MODNAMELEN);
11206           ASSERT(strlen(match->dtpd_func) < DTRACE_FUNCNAMELEN);
11207           ASSERT(strlen(match->dtpd_name) < DTRACE_NAMELEN);

11209           new = dtrace_enabling_create(&state->dts_vstate);

11211           /*
11212            * Iterate over all retained enablings, looking for enablings that
11213            * match the specified state.
11214            */
11215           for (enab = dtrace_retained; enab != NULL; enab = enab->dten_next) {
```



```
11216                   int i;

11218                   /*
11219                    * dtvs_state can only be NULL for helper enablings -- and
11220                    * helper enablings can't be retained.
11221                    */
11222                   ASSERT(enab->dten_vstate->dtvs_state != NULL);

11224                   if (enab->dten_vstate->dtvs_state != state)
11225                           continue;

11227                   /*
11228                    * Now iterate over each probe description; we're looking for
11229                    * an exact match to the specified probe description.
11230                    */
11231                   for (i = 0; i < enab->dten_ndesc; i++) {
11232                           dtrace_ecbdesc_t *ep = enab->dten_desc[i];
11233                           dtrace_probedesc_t *pd = &ep->dted_probe;

11235                           if (strcmp(pd->dtpd_provider, match->dtpd_provider))
11236                                   continue;

11238                           if (strcmp(pd->dtpd_mod, match->dtpd_mod))
11239                                   continue;

11241                           if (strcmp(pd->dtpd_func, match->dtpd_func))
11242                                   continue;

11244                           if (strcmp(pd->dtpd_name, match->dtpd_name))
11245                                   continue;

11247                           /*
11248                            * We have a winning probe!  Add it to our growing
11249                            * enabling.
11250                            */
11251                           found = 1;
11252                           dtrace_enabling_addlike(new, ep, create);
11253                   }
11254           }

11256           if (!found || (err = dtrace_enabling_retain(new)) != 0) {
11257                   dtrace_enabling_destroy(new);
11258                   return (err);
11259           }

11261           return (0);
11262   }

11264   static void
11265   dtrace_enabling_retract(dtrace_state_t *state)
11266   {
11267           dtrace_enabling_t *enab, *next;

11269           ASSERT(MUTEX_HELD(&dtrace_lock));

11271           /*
11272            * Iterate over all retained enablings, destroy the enablings retained
11273            * for the specified state.
11274            */
11275           for (enab = dtrace_retained; enab != NULL; enab = next) {
11276                   next = enab->dten_next;

11278                   /*
11279                    * dtvs_state can only be NULL for helper enablings -- and
11280                    * helper enablings can't be retained.
11281                    */
```

```
11282                 ASSERT(enab->dten_vstate->dtvs_state != NULL);

11284                 if (enab->dten_vstate->dtvs_state == state) {
11285                         ASSERT(state->dts_nretained > 0);
11286                         dtrace_enabling_destroy(enab);
11287                 }
11288         }

11290         ASSERT(state->dts_nretained == 0);
11291 }

11293 static int
11294 dtrace_enabling_match(dtrace_enabling_t *enab, int *nmatched)
11295 {
11296         int i = 0;
11297         int total_matched = 0, matched = 0;

11299         ASSERT(MUTEX_HELD(&cpu_lock));
11300         ASSERT(MUTEX_HELD(&dtrace_lock));

11302         for (i = 0; i < enab->dten_ndesc; i++) {
11303                 dtrace_ecbdesc_t *ep = enab->dten_desc[i];

11305                 enab->dten_current = ep;
11306                 enab->dten_error = 0;

11308                 /*
11309                  * If a provider failed to enable a probe then get out and
11310                  * let the consumer know we failed.
11311                  */
11312                 if ((matched = dtrace_probe_enable(&ep->dted_probe, enab)) < 0)
11313                         return (EBUSY);

11315                 total_matched += matched;

11317                 if (enab->dten_error != 0) {
11318                         /*
11319                          * If we get an error half-way through enabling the
11320                          * probes, we kick out -- perhaps with some number of
11321                          * them enabled.  Leaving enabled probes enabled may
11322                          * be slightly confusing for user-level, but we expect
11323                          * that no one will attempt to actually drive on in
11324                          * the face of such errors.  If this is an anonymous
11325                          * enabling (indicated with a NULL nmatched pointer),
11326                          * we cmn_err() a message.  We aren't expecting to
11327                          * get such an error -- such as it can exist at all,
11328                          * it would be a result of corrupted DOF in the driver
11329                          * properties.
11330                          */
11331                         if (nmatched == NULL) {
11332                                 cmn_err(CE_WARN, "dtrace_enabling_match() "
11333                                     "error on %p: %d", (void *)ep,
11334                                     enab->dten_error);
11335                         }

11337                         return (enab->dten_error);
11338                 }
11339         }

11341         enab->dten_probegen = dtrace_probegen;
11342         if (nmatched != NULL)
11343                 *nmatched = total_matched;

11345         return (0);
11346 }
```



```
11348 static void
11349 dtrace_enabling_matchall(void)
11350 {
11351         dtrace_enabling_t *enab;

11353         mutex_enter(&cpu_lock);
11354         mutex_enter(&dtrace_lock);

11356         /*
11357          * Iterate over all retained enablings to see if any probes match
11358          * against them.  We only perform this operation on enablings for which
11359          * we have sufficient permissions by virtue of being in the global zone
11360          * or in the same zone as the DTrace client.  Because we can be called
11361          * after dtrace_detach() has been called, we cannot assert that there
11362          * are retained enablings.  We can safely load from dtrace_retained,
11363          * however:  the taskq_destroy() at the end of dtrace_detach() will
11364          * block pending our completion.
11365          */
11366         for (enab = dtrace_retained; enab != NULL; enab = enab->dten_next) {
11367                 dtrace_cred_t *dcr = &enab->dten_vstate->dtvs_state->dts_cred;
11368                 cred_t *cr = dcr->dcr_cred;
11369                 zoneid_t zone = cr != NULL ? crgetzoneid(cr) : 0;

11371                 if ((dcr->dcr_visible & DTRACE_CRV_ALLZONE) || (cr != NULL &&
11372                     (zone == GLOBAL_ZONEID || getzoneid() == zone)))
11373                         (void) dtrace_enabling_match(enab, NULL);
11374         }

11376         mutex_exit(&dtrace_lock);
11377         mutex_exit(&cpu_lock);
11378 }

11380 /*
11381  * If an enabling is to be enabled without having matched probes (that is, if
11382  * dtrace_state_go() is to be called on the underlying dtrace_state_t), the
11383  * enabling must be _primed_ by creating an ECB for every ECB description.
11384  * This must be done to assure that we know the number of speculations, the
11385  * number of aggregations, the minimum buffer size needed, etc. before we
11386  * transition out of DTRACE_ACTIVITY_INACTIVE.  To do this without actually
11387  * enabling any probes, we create ECBs for every ECB decription, but with a
11388  * NULL probe -- which is exactly what this function does.
11389  */
11390 static void
11391 dtrace_enabling_prime(dtrace_state_t *state)
11392 {
11393         dtrace_enabling_t *enab;
11394         int i;

11396         for (enab = dtrace_retained; enab != NULL; enab = enab->dten_next) {
11397                 ASSERT(enab->dten_vstate->dtvs_state != NULL);

11399                 if (enab->dten_vstate->dtvs_state != state)
11400                         continue;

11402                 /*
11403                  * We don't want to prime an enabling more than once, lest
11404                  * we allow a malicious user to induce resource exhaustion.
11405                  * (The ECBs that result from priming an enabling aren't
11406                  * leaked -- but they also aren't deallocated until the
11407                  * consumer state is destroyed.)
11408                  */
11409                 if (enab->dten_primed)
11410                         continue;

11412                 for (i = 0; i < enab->dten_ndesc; i++) {
11413                         enab->dten_current = enab->dten_desc[i];
```

```
11414                           (void) dtrace_probe_enable(NULL, enab);
11415                   }
11417                   enab->dten_primed = 1;
11418           }
11419   }
11421   /*
11422    * Called to indicate that probes should be provided due to retained
11423    * enablings.  This is implemented in terms of dtrace_probe_provide(), but it
11424    * must take an initial lap through the enabling calling the dtps_provide()
11425    * entry point explicitly to allow for autocreated probes.
11426    */
11427   static void
11428   dtrace_enabling_provide(dtrace_provider_t *prv)
11429   {
11430           int i, all = 0;
11431           dtrace_probedesc_t desc;
11432           dtrace_genid_t gen;
11434           ASSERT(MUTEX_HELD(&dtrace_lock));
11435           ASSERT(MUTEX_HELD(&dtrace_provider_lock));
11437           if (prv == NULL) {
11438                   all = 1;
11439                   prv = dtrace_provider;
11440           }
11442           do {
11443                   dtrace_enabling_t *enab;
11444                   void *parg = prv->dtpv_arg;
11446   retry:
11447                   gen = dtrace_retained_gen;
11448                   for (enab = dtrace_retained; enab != NULL;
11449                       enab = enab->dten_next) {
11450                           for (i = 0; i < enab->dten_ndesc; i++) {
11451                                   desc = enab->dten_desc[i]->dted_probe;
11452                                   mutex_exit(&dtrace_lock);
11453                                   prv->dtpv_pops.dtps_provide(parg, &desc);
11454                                   mutex_enter(&dtrace_lock);
11455                                   /*
11456                                    * Process the retained enablings again if
11457                                    * they have changed while we weren't holding
11458                                    * dtrace_lock.
11459                                    */
11460                                   if (gen != dtrace_retained_gen)
11461                                           goto retry;
11462                           }
11463                   }
11464           } while (all && (prv = prv->dtpv_next) != NULL);
11466           mutex_exit(&dtrace_lock);
11467           dtrace_probe_provide(NULL, all ? NULL : prv);
11468           mutex_enter(&dtrace_lock);
11469   }
11471   /*
11472    * Called to reap ECBs that are attached to probes from defunct providers.
11473    */
11474   static void
11475   dtrace_enabling_reap(void)
11476   {
11477           dtrace_provider_t *prov;
11478           dtrace_probe_t *probe;
11479           dtrace_ecb_t *ecb;
```



```
11480           hrtime_t when;
11481           int i;
11483           mutex_enter(&cpu_lock);
11484           mutex_enter(&dtrace_lock);
11486           for (i = 0; i < dtrace_nprobes; i++) {
11487                   if ((probe = dtrace_probes[i]) == NULL)
11488                           continue;
11490                   if (probe->dtpr_ecb == NULL)
11491                           continue;
11493                   prov = probe->dtpr_provider;
11495                   if ((when = prov->dtpv_defunct) == 0)
11496                           continue;
11498                   /*
11499                    * We have ECBs on a defunct provider:  we want to reap these
11500                    * ECBs to allow the provider to unregister.  The destruction
11501                    * of these ECBs must be done carefully:  if we destroy the ECB
11502                    * and the consumer later wishes to consume an EPID that
11503                    * corresponds to the destroyed ECB (and if the EPID metadata
11504                    * has not been previously consumed), the consumer will abort
11505                    * processing on the unknown EPID.  To reduce (but not, sadly,
11506                    * eliminate) the possibility of this, we will only destroy an
11507                    * ECB for a defunct provider if, for the state that
11508                    * corresponds to the ECB:
11509                    *
11510                    *  (a) There is no speculative tracing (which can effectively
11511                    *      cache an EPID for an arbitrary amount of time).
11512                    *
11513                    *  (b) The principal buffers have been switched twice since the
11514                    *      provider became defunct.
11515                    *
11516                    *  (c) The aggregation buffers are of zero size or have been
11517                    *      switched twice since the provider became defunct.
11518                    *
11519                    * We use dts_speculates to determine (a) and call a function
11520                    * (dtrace_buffer_consumed()) to determine (b) and (c).  Note
11521                    * that as soon as we've been unable to destroy one of the ECBs
11522                    * associated with the probe, we quit trying -- reaping is only
11523                    * fruitful in as much as we can destroy all ECBs associated
11524                    * with the defunct provider's probes.
11525                    */
11526                   while ((ecb = probe->dtpr_ecb) != NULL) {
11527                           dtrace_state_t *state = ecb->dte_state;
11528                           dtrace_buffer_t *buf = state->dts_buffer;
11529                           dtrace_buffer_t *aggbuf = state->dts_aggbuffer;
11531                           if (state->dts_speculates)
11532                                   break;
11534                           if (!dtrace_buffer_consumed(buf, when))
11535                                   break;
11537                           if (!dtrace_buffer_consumed(aggbuf, when))
11538                                   break;
11540                           dtrace_ecb_disable(ecb);
11541                           ASSERT(probe->dtpr_ecb != ecb);
11542                           dtrace_ecb_destroy(ecb);
11543                   }
11544           }
```

```
11546         mutex_exit(&dtrace_lock);
11547         mutex_exit(&cpu_lock);
11548 }

11550 /*
11551  * DTrace DOF Functions
11552  */
11553 /*ARGSUSED*/
11554 static void
11555 dtrace_dof_error(dof_hdr_t *dof, const char *str)
11556 {
11557         if (dtrace_err_verbose)
11558                 cmn_err(CE_WARN, "failed to process DOF: %s", str);

11560 #ifdef DTRACE_ERRDEBUG
11561         dtrace_errdebug(str);
11562 #endif
11563 }

11565 /*
11566  * Create DOF out of a currently enabled state.  Right now, we only create
11567  * DOF containing the run-time options -- but this could be expanded to create
11568  * complete DOF representing the enabled state.
11569  */
11570 static dof_hdr_t *
11571 dtrace_dof_create(dtrace_state_t *state)
11572 {
11573         dof_hdr_t *dof;
11574         dof_sec_t *sec;
11575         dof_optdesc_t *opt;
11576         int i, len = sizeof (dof_hdr_t) +
11577             roundup(sizeof (dof_sec_t), sizeof (uint64_t)) +
11578             sizeof (dof_optdesc_t) * DTRACEOPT_MAX;

11580         ASSERT(MUTEX_HELD(&dtrace_lock));

11582         dof = kmem_zalloc(len, KM_SLEEP);
11583         dof->dofh_ident[DOF_ID_MAG0] = DOF_MAG_MAG0;
11584         dof->dofh_ident[DOF_ID_MAG1] = DOF_MAG_MAG1;
11585         dof->dofh_ident[DOF_ID_MAG2] = DOF_MAG_MAG2;
11586         dof->dofh_ident[DOF_ID_MAG3] = DOF_MAG_MAG3;

11588         dof->dofh_ident[DOF_ID_MODEL] = DOF_MODEL_NATIVE;
11589         dof->dofh_ident[DOF_ID_ENCODING] = DOF_ENCODE_NATIVE;
11590         dof->dofh_ident[DOF_ID_VERSION] = DOF_VERSION;
11591         dof->dofh_ident[DOF_ID_DIFVERS] = DIF_VERSION;
11592         dof->dofh_ident[DOF_ID_DIFIREG] = DIF_DIR_NREGS;
11593         dof->dofh_ident[DOF_ID_DIFTREG] = DIF_DTR_NREGS;

11595         dof->dofh_flags = 0;
11596         dof->dofh_hdrsize = sizeof (dof_hdr_t);
11597         dof->dofh_secsize = sizeof (dof_sec_t);
11598         dof->dofh_secnum = 1;   /* only DOF_SECT_OPTDESC */
11599         dof->dofh_secoff = sizeof (dof_hdr_t);
11600         dof->dofh_loadsz = len;
11601         dof->dofh_filesz = len;
11602         dof->dofh_pad = 0;

11604         /*
11605          * Fill in the option section header...
11606          */
11607         sec = (dof_sec_t *)((uintptr_t)dof + sizeof (dof_hdr_t));
11608         sec->dofs_type = DOF_SECT_OPTDESC;
11609         sec->dofs_align = sizeof (uint64_t);
11610         sec->dofs_flags = DOF_SECF_LOAD;
11611         sec->dofs_entsize = sizeof (dof_optdesc_t);
```



```
11613         opt = (dof_optdesc_t *)((uintptr_t)sec +
11614             roundup(sizeof (dof_sec_t), sizeof (uint64_t)));

11616         sec->dofs_offset = (uintptr_t)opt - (uintptr_t)dof;
11617         sec->dofs_size = sizeof (dof_optdesc_t) * DTRACEOPT_MAX;

11619         for (i = 0; i < DTRACEOPT_MAX; i++) {
11620                 opt[i].dofo_option = i;
11621                 opt[i].dofo_strtab = DOF_SECIDX_NONE;
11622                 opt[i].dofo_value = state->dts_options[i];
11623         }

11625         return (dof);
11626 }

11628 static dof_hdr_t *
11629 dtrace_dof_copyin(uintptr_t uarg, int *errp)
11630 {
11631         dof_hdr_t hdr, *dof;

11633         ASSERT(!MUTEX_HELD(&dtrace_lock));

11635         /*
11636          * First, we're going to copyin() the sizeof (dof_hdr_t).
11637          */
11638         if (copyin((void *)uarg, &hdr, sizeof (hdr)) != 0) {
11639                 dtrace_dof_error(NULL, "failed to copyin DOF header");
11640                 *errp = EFAULT;
11641                 return (NULL);
11642         }

11644         /*
11645          * Now we'll allocate the entire DOF and copy it in -- provided
11646          * that the length isn't outrageous.
11647          */
11648         if (hdr.dofh_loadsz >= dtrace_dof_maxsize) {
11649                 dtrace_dof_error(&hdr, "load size exceeds maximum");
11650                 *errp = E2BIG;
11651                 return (NULL);
11652         }

11654         if (hdr.dofh_loadsz < sizeof (hdr)) {
11655                 dtrace_dof_error(&hdr, "invalid load size");
11656                 *errp = EINVAL;
11657                 return (NULL);
11658         }

11660         dof = kmem_alloc(hdr.dofh_loadsz, KM_SLEEP);

11662         if (copyin((void *)uarg, dof, hdr.dofh_loadsz) != 0 ||
11663             dof->dofh_loadsz != hdr.dofh_loadsz) {
11664                 kmem_free(dof, hdr.dofh_loadsz);
11665                 *errp = EFAULT;
11666                 return (NULL);
11667         }

11669         return (dof);
11670 }

11672 static dof_hdr_t *
11673 dtrace_dof_property(const char *name)
11674 {
11675         uchar_t *buf;
11676         uint64_t loadsz;
11677         unsigned int len, i;
```

```
11678         dof_hdr_t *dof;

11680         /*
11681          * Unfortunately, array of values in .conf files are always (and
11682          * only) interpreted to be integer arrays.  We must read our DOF
11683          * as an integer array, and then squeeze it into a byte array.
11684          */
11685         if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, dtrace_devi, 0,
11686             (char *)name, (int **)&buf, &len) != DDI_PROP_SUCCESS)
11687                 return (NULL);

11689         for (i = 0; i < len; i++)
11690                 buf[i] = (uchar_t)(((int *)buf)[i]);

11692         if (len < sizeof (dof_hdr_t)) {
11693                 ddi_prop_free(buf);
11694                 dtrace_dof_error(NULL, "truncated header");
11695                 return (NULL);
11696         }

11698         if (len < (loadsz = ((dof_hdr_t *)buf)->dofh_loadsz)) {
11699                 ddi_prop_free(buf);
11700                 dtrace_dof_error(NULL, "truncated DOF");
11701                 return (NULL);
11702         }

11704         if (loadsz >= dtrace_dof_maxsize) {
11705                 ddi_prop_free(buf);
11706                 dtrace_dof_error(NULL, "oversized DOF");
11707                 return (NULL);
11708         }

11710         dof = kmem_alloc(loadsz, KM_SLEEP);
11711         bcopy(buf, dof, loadsz);
11712         ddi_prop_free(buf);

11714         return (dof);
11715 }

11717 static void
11718 dtrace_dof_destroy(dof_hdr_t *dof)
11719 {
11720         kmem_free(dof, dof->dofh_loadsz);
11721 }

11723 /*
11724  * Return the dof_sec_t pointer corresponding to a given section index.  If the
11725  * index is not valid, dtrace_dof_error() is called and NULL is returned.  If
11726  * a type other than DOF_SECT_NONE is specified, the header is checked against
11727  * this type and NULL is returned if the types do not match.
11728  */
11729 static dof_sec_t *
11730 dtrace_dof_sect(dof_hdr_t *dof, uint32_t type, dof_secidx_t i)
11731 {
11732         dof_sec_t *sec = (dof_sec_t *)(uintptr_t)
11733             ((uintptr_t)dof + dof->dofh_secoff + i * dof->dofh_secsize);

11735         if (i >= dof->dofh_secnum) {
11736                 dtrace_dof_error(dof, "referenced section index is invalid");
11737                 return (NULL);
11738         }

11740         if (!(sec->dofs_flags & DOF_SECF_LOAD)) {
11741                 dtrace_dof_error(dof, "referenced section is not loadable");
11742                 return (NULL);
11743         }
```



```
11745         if (type != DOF_SECT_NONE && type != sec->dofs_type) {
11746                 dtrace_dof_error(dof, "referenced section is the wrong type");
11747                 return (NULL);
11748         }

11750         return (sec);
11751 }

11753 static dtrace_probedesc_t *
11754 dtrace_dof_probedesc(dof_hdr_t *dof, dof_sec_t *sec, dtrace_probedesc_t *desc)
11755 {
11756         dof_probedesc_t *probe;
11757         dof_sec_t *strtab;
11758         uintptr_t daddr = (uintptr_t)dof;
11759         uintptr_t str;
11760         size_t size;

11762         if (sec->dofs_type != DOF_SECT_PROBEDESC) {
11763                 dtrace_dof_error(dof, "invalid probe section");
11764                 return (NULL);
11765         }

11767         if (sec->dofs_align != sizeof (dof_secidx_t)) {
11768                 dtrace_dof_error(dof, "bad alignment in probe description");
11769                 return (NULL);
11770         }

11772         if (sec->dofs_offset + sizeof (dof_probedesc_t) > dof->dofh_loadsz) {
11773                 dtrace_dof_error(dof, "truncated probe description");
11774                 return (NULL);
11775         }

11777         probe = (dof_probedesc_t *)(uintptr_t)(daddr + sec->dofs_offset);
11778         strtab = dtrace_dof_sect(dof, DOF_SECT_STRTAB, probe->dofp_strtab);

11780         if (strtab == NULL)
11781                 return (NULL);

11783         str = daddr + strtab->dofs_offset;
11784         size = strtab->dofs_size;

11786         if (probe->dofp_provider >= strtab->dofs_size) {
11787                 dtrace_dof_error(dof, "corrupt probe provider");
11788                 return (NULL);
11789         }

11791         (void) strncpy(desc->dtpd_provider,
11792             (char *)(str + probe->dofp_provider),
11793             MIN(DTRACE_PROVNAMELEN - 1, size - probe->dofp_provider));

11795         if (probe->dofp_mod >= strtab->dofs_size) {
11796                 dtrace_dof_error(dof, "corrupt probe module");
11797                 return (NULL);
11798         }

11800         (void) strncpy(desc->dtpd_mod, (char *)(str + probe->dofp_mod),
11801             MIN(DTRACE_MODNAMELEN - 1, size - probe->dofp_mod));

11803         if (probe->dofp_func >= strtab->dofs_size) {
11804                 dtrace_dof_error(dof, "corrupt probe function");
11805                 return (NULL);
11806         }

11808         (void) strncpy(desc->dtpd_func, (char *)(str + probe->dofp_func),
11809             MIN(DTRACE_FUNCNAMELEN - 1, size - probe->dofp_func));
```

```
11811         if (probe->dofp_name >= strtab->dofs_size) {
11812                 dtrace_dof_error(dof, "corrupt probe name");
11813                 return (NULL);
11814         }

11816         (void) strncpy(desc->dtpd_name, (char *)(str + probe->dofp_name),
11817             MIN(DTRACE_NAMELEN - 1, size - probe->dofp_name));

11819         return (desc);
11820 }

11822 static dtrace_difo_t *
11823 dtrace_dof_difo(dof_hdr_t *dof, dof_sec_t *sec, dtrace_vstate_t *vstate,
11824     cred_t *cr)
11825 {
11826         dtrace_difo_t *dp;
11827         size_t ttl = 0;
11828         dof_difohdr_t *dofd;
11829         uintptr_t daddr = (uintptr_t)dof;
11830         size_t max = dtrace_difo_maxsize;
11831         int i, l, n;

11833         static const struct {
11834                 int section;
11835                 int bufoffs;
11836                 int lenoffs;
11837                 int entsize;
11838                 int align;
11839                 const char *msg;
11840         } difo[] = {
11841                 { DOF_SECT_DIF, offsetof(dtrace_difo_t, dtdo_buf),
11842                 offsetof(dtrace_difo_t, dtdo_len), sizeof (dif_instr_t),
11843                 sizeof (dif_instr_t), "multiple DIF sections" },

11845                 { DOF_SECT_INTTAB, offsetof(dtrace_difo_t, dtdo_inttab),
11846                 offsetof(dtrace_difo_t, dtdo_intlen), sizeof (uint64_t),
11847                 sizeof (uint64_t), "multiple integer tables" },

11849                 { DOF_SECT_STRTAB, offsetof(dtrace_difo_t, dtdo_strtab),
11850                 offsetof(dtrace_difo_t, dtdo_strlen), 0,
11851                 sizeof (char), "multiple string tables" },

11853                 { DOF_SECT_VARTAB, offsetof(dtrace_difo_t, dtdo_vartab),
11854                 offsetof(dtrace_difo_t, dtdo_varlen), sizeof (dtrace_difv_t),
11855                 sizeof (uint_t), "multiple variable tables" },

11857                 { DOF_SECT_NONE, 0, 0, 0, NULL }
11858         };

11860         if (sec->dofs_type != DOF_SECT_DIFOHDR) {
11861                 dtrace_dof_error(dof, "invalid DIFO header section");
11862                 return (NULL);
11863         }

11865         if (sec->dofs_align != sizeof (dof_secidx_t)) {
11866                 dtrace_dof_error(dof, "bad alignment in DIFO header");
11867                 return (NULL);
11868         }

11870         if (sec->dofs_size < sizeof (dof_difohdr_t) ||
11871             sec->dofs_size % sizeof (dof_secidx_t)) {
11872                 dtrace_dof_error(dof, "bad size in DIFO header");
11873                 return (NULL);
11874         }
```



```
11876         dofd = (dof_difohdr_t *)(uintptr_t)(daddr + sec->dofs_offset);
11877         n = (sec->dofs_size - sizeof (*dofd)) / sizeof (dof_secidx_t) + 1;

11879         dp = kmem_zalloc(sizeof (dtrace_difo_t), KM_SLEEP);
11880         dp->dtdo_rtype = dofd->dofd_rtype;

11882         for (l = 0; l < n; l++) {
11883                 dof_sec_t *subsec;
11884                 void **bufp;
11885                 uint32_t *lenp;

11887                 if ((subsec = dtrace_dof_sect(dof, DOF_SECT_NONE,
11888                     dofd->dofd_links[l])) == NULL)
11889                         goto err; /* invalid section link */

11891                 if (ttl + subsec->dofs_size > max) {
11892                         dtrace_dof_error(dof, "exceeds maximum size");
11893                         goto err;
11894                 }

11896                 ttl += subsec->dofs_size;

11898                 for (i = 0; difo[i].section != DOF_SECT_NONE; i++) {
11899                         if (subsec->dofs_type != difo[i].section)
11900                                 continue;

11902                         if (!(subsec->dofs_flags & DOF_SECF_LOAD)) {
11903                                 dtrace_dof_error(dof, "section not loaded");
11904                                 goto err;
11905                         }

11907                         if (subsec->dofs_align != difo[i].align) {
11908                                 dtrace_dof_error(dof, "bad alignment");
11909                                 goto err;
11910                         }

11912                         bufp = (void **)((uintptr_t)dp + difo[i].bufoffs);
11913                         lenp = (uint32_t *)((uintptr_t)dp + difo[i].lenoffs);

11915                         if (*bufp != NULL) {
11916                                 dtrace_dof_error(dof, difo[i].msg);
11917                                 goto err;
11918                         }

11920                         if (difo[i].entsize != subsec->dofs_entsize) {
11921                                 dtrace_dof_error(dof, "entry size mismatch");
11922                                 goto err;
11923                         }

11925                         if (subsec->dofs_entsize != 0 &&
11926                             (subsec->dofs_size % subsec->dofs_entsize) != 0) {
11927                                 dtrace_dof_error(dof, "corrupt entry size");
11928                                 goto err;
11929                         }

11931                         *lenp = subsec->dofs_size;
11932                         *bufp = kmem_alloc(subsec->dofs_size, KM_SLEEP);
11933                         bcopy((char *)(uintptr_t)(daddr + subsec->dofs_offset),
11934                             *bufp, subsec->dofs_size);

11936                         if (subsec->dofs_entsize != 0)
11937                                 *lenp /= subsec->dofs_entsize;

11939                         break;
11940                 }
```

```
11942                 /*
11943                  * If we encounter a loadable DIFO sub-section that is not
11944                  * known to us, assume this is a broken program and fail.
11945                  */
11946                 if (difo[i].section == DOF_SECT_NONE &&
11947                     (subsec->dofs_flags & DOF_SECF_LOAD)) {
11948                         dtrace_dof_error(dof, "unrecognized DIFO subsection");
11949                         goto err;
11950                 }
11951         }

11953         if (dp->dtdo_buf == NULL) {
11954                 /*
11955                  * We can't have a DIF object without DIF text.
11956                  */
11957                 dtrace_dof_error(dof, "missing DIF text");
11958                 goto err;
11959         }

11961         /*
11962          * Before we validate the DIF object, run through the variable table
11963          * looking for the strings -- if any of their size are under, we'll set
11964          * their size to be the system-wide default string size.  Note that
11965          * this should _not_ happen if the "strsize" option has been set --
11966          * in this case, the compiler should have set the size to reflect the
11967          * setting of the option.
11968          */
11969         for (i = 0; i < dp->dtdo_varlen; i++) {
11970                 dtrace_difv_t *v = &dp->dtdo_vartab[i];
11971                 dtrace_diftype_t *t = &v->dtdv_type;

11973                 if (v->dtdv_id < DIF_VAR_OTHER_UBASE)
11974                         continue;

11976                 if (t->dtdt_kind == DIF_TYPE_STRING && t->dtdt_size == 0)
11977                         t->dtdt_size = dtrace_strsize_default;
11978         }

11980         if (dtrace_difo_validate(dp, vstate, DIF_DIR_NREGS, cr) != 0)
11981                 goto err;

11983         dtrace_difo_init(dp, vstate);
11984         return (dp);

11986 err:
11987         kmem_free(dp->dtdo_buf, dp->dtdo_len * sizeof (dif_instr_t));
11988         kmem_free(dp->dtdo_inttab, dp->dtdo_intlen * sizeof (uint64_t));
11989         kmem_free(dp->dtdo_strtab, dp->dtdo_strlen);
11990         kmem_free(dp->dtdo_vartab, dp->dtdo_varlen * sizeof (dtrace_difv_t));

11992         kmem_free(dp, sizeof (dtrace_difo_t));
11993         return (NULL);
11994 }

11996 static dtrace_predicate_t *
11997 dtrace_dof_predicate(dof_hdr_t *dof, dof_sec_t *sec, dtrace_vstate_t *vstate,
11998     cred_t *cr)
11999 {
12000         dtrace_difo_t *dp;

12002         if ((dp = dtrace_dof_difo(dof, sec, vstate, cr)) == NULL)
12003                 return (NULL);

12005         return (dtrace_predicate_create(dp));
12006 }
```



```
12008 static dtrace_actdesc_t *
12009 dtrace_dof_actdesc(dof_hdr_t *dof, dof_sec_t *sec, dtrace_vstate_t *vstate,
12010     cred_t *cr)
12011 {
12012         dtrace_actdesc_t *act, *first = NULL, *last = NULL, *next;
12013         dof_actdesc_t *desc;
12014         dof_sec_t *difosec;
12015         size_t offs;
12016         uintptr_t daddr = (uintptr_t)dof;
12017         uint64_t arg;
12018         dtrace_actkind_t kind;

12020         if (sec->dofs_type != DOF_SECT_ACTDESC) {
12021                 dtrace_dof_error(dof, "invalid action section");
12022                 return (NULL);
12023         }

12025         if (sec->dofs_offset + sizeof (dof_actdesc_t) > dof->dofh_loadsz) {
12026                 dtrace_dof_error(dof, "truncated action description");
12027                 return (NULL);
12028         }

12030         if (sec->dofs_align != sizeof (uint64_t)) {
12031                 dtrace_dof_error(dof, "bad alignment in action description");
12032                 return (NULL);
12033         }

12035         if (sec->dofs_size < sec->dofs_entsize) {
12036                 dtrace_dof_error(dof, "section entry size exceeds total size");
12037                 return (NULL);
12038         }

12040         if (sec->dofs_entsize != sizeof (dof_actdesc_t)) {
12041                 dtrace_dof_error(dof, "bad entry size in action description");
12042                 return (NULL);
12043         }

12045         if (sec->dofs_size / sec->dofs_entsize > dtrace_actions_max) {
12046                 dtrace_dof_error(dof, "actions exceed dtrace_actions_max");
12047                 return (NULL);
12048         }

12050         for (offs = 0; offs < sec->dofs_size; offs += sec->dofs_entsize) {
12051                 desc = (dof_actdesc_t *)(daddr +
12052                     (uintptr_t)sec->dofs_offset + offs);
12053                 kind = (dtrace_actkind_t)desc->dofa_kind;

12055                 if ((DTRACEACT_ISPRINTFLIKE(kind) &&
12056                     (kind != DTRACEACT_PRINTA ||
12057                     desc->dofa_strtab != DOF_SECIDX_NONE)) ||
12058                     (kind == DTRACEACT_DIFEXPR &&
12059                     desc->dofa_strtab != DOF_SECIDX_NONE)) {
12060                         dof_sec_t *strtab;
12061                         char *str, *fmt;
12062                         uint64_t i;

12064                         /*
12065                          * The argument to these actions is an index into the
12066                          * DOF string table.  For printf()-like actions, this
12067                          * is the format string.  For print(), this is the
12068                          * CTF type of the expression result.
12069                          */
12070                         if ((strtab = dtrace_dof_sect(dof,
12071                             DOF_SECT_STRTAB, desc->dofa_strtab)) == NULL)
12072                                 goto err;
```

```
12074                         str = (char *)((uintptr_t)dof +
12075                             (uintptr_t)strtab->dofs_offset);
12076
12077                         for (i = desc->dofa_arg; i < strtab->dofs_size; i++) {
12078                                 if (str[i] == '\0')
12079                                         break;
12080                         }
12081
12082                         if (i >= strtab->dofs_size) {
12083                                 dtrace_dof_error(dof, "bogus format string");
12084                                 goto err;
12085                         }
12086
12087                         if (i == desc->dofa_arg) {
12088                                 dtrace_dof_error(dof, "empty format string");
12089                                 goto err;
12090                         }
12091
12092                         i -= desc->dofa_arg;
12093                         fmt = kmem_alloc(i + 1, KM_SLEEP);
12094                         bcopy(&str[desc->dofa_arg], fmt, i + 1);
12095                         arg = (uint64_t)(uintptr_t)fmt;
12096                 } else {
12097                         if (kind == DTRACEACT_PRINTA) {
12098                                 ASSERT(desc->dofa_strtab == DOF_SECIDX_NONE);
12099                                 arg = 0;
12100                         } else {
12101                                 arg = desc->dofa_arg;
12102                         }
12103                 }
12104
12105                 act = dtrace_actdesc_create(kind, desc->dofa_ntuple,
12106                     desc->dofa_uarg, arg);
12107
12108                 if (last != NULL) {
12109                         last->dtad_next = act;
12110                 } else {
12111                         first = act;
12112                 }
12113
12114                 last = act;
12115
12116                 if (desc->dofa_difo == DOF_SECIDX_NONE)
12117                         continue;
12118
12119                 if ((difosec = dtrace_dof_sect(dof,
12120                     DOF_SECT_DIFOHDR, desc->dofa_difo)) == NULL)
12121                         goto err;
12122
12123                 act->dtad_difo = dtrace_dof_difo(dof, difosec, vstate, cr);
12124
12125                 if (act->dtad_difo == NULL)
12126                         goto err;
12127         }
12128
12129         ASSERT(first != NULL);
12130         return (first);
12131
12132 err:
12133         for (act = first; act != NULL; act = next) {
12134                 next = act->dtad_next;
12135                 dtrace_actdesc_release(act, vstate);
12136         }
12137
12138         return (NULL);
12139 }
```



```
12141 static dtrace_ecbdesc_t *
12142 dtrace_dof_ecbdesc(dof_hdr_t *dof, dof_sec_t *sec, dtrace_vstate_t *vstate,
12143     cred_t *cr)
12144 {
12145         dtrace_ecbdesc_t *ep;
12146         dof_ecbdesc_t *ecb;
12147         dtrace_probedesc_t *desc;
12148         dtrace_predicate_t *pred = NULL;
12149
12150         if (sec->dofs_size < sizeof (dof_ecbdesc_t)) {
12151                 dtrace_dof_error(dof, "truncated ECB description");
12152                 return (NULL);
12153         }
12154
12155         if (sec->dofs_align != sizeof (uint64_t)) {
12156                 dtrace_dof_error(dof, "bad alignment in ECB description");
12157                 return (NULL);
12158         }
12159
12160         ecb = (dof_ecbdesc_t *)((uintptr_t)dof + (uintptr_t)sec->dofs_offset);
12161         sec = dtrace_dof_sect(dof, DOF_SECT_PROBEDESC, ecb->dofe_probes);
12162
12163         if (sec == NULL)
12164                 return (NULL);
12165
12166         ep = kmem_zalloc(sizeof (dtrace_ecbdesc_t), KM_SLEEP);
12167         ep->dted_uarg = ecb->dofe_uarg;
12168         desc = &ep->dted_probe;
12169
12170         if (dtrace_dof_probedesc(dof, sec, desc) == NULL)
12171                 goto err;
12172
12173         if (ecb->dofe_pred != DOF_SECIDX_NONE) {
12174                 if ((sec = dtrace_dof_sect(dof,
12175                     DOF_SECT_DIFOHDR, ecb->dofe_pred)) == NULL)
12176                         goto err;
12177
12178                 if ((pred = dtrace_dof_predicate(dof, sec, vstate, cr)) == NULL)
12179                         goto err;
12180
12181                 ep->dted_pred.dtpdd_predicate = pred;
12182         }
12183
12184         if (ecb->dofe_actions != DOF_SECIDX_NONE) {
12185                 if ((sec = dtrace_dof_sect(dof,
12186                     DOF_SECT_ACTDESC, ecb->dofe_actions)) == NULL)
12187                         goto err;
12188
12189                 ep->dted_action = dtrace_dof_actdesc(dof, sec, vstate, cr);
12190
12191                 if (ep->dted_action == NULL)
12192                         goto err;
12193         }
12194
12195         return (ep);
12196
12197 err:
12198         if (pred != NULL)
12199                 dtrace_predicate_release(pred, vstate);
12200         kmem_free(ep, sizeof (dtrace_ecbdesc_t));
12201         return (NULL);
12202 }
12203
12204 /*
12205  * Apply the relocations from the specified 'sec' (a DOF_SECT_URELHDR) to the
```

```
12206  * specified DOF.  At present, this amounts to simply adding 'ubase' to the
12207  * site of any user SETX relocations to account for load object base address.
12208  * In the future, if we need other relocations, this function can be extended.
12209  */
12210 static int
12211 dtrace_dof_relocate(dof_hdr_t *dof, dof_sec_t *sec, uint64_t ubase)
12212 {
12213         uintptr_t daddr = (uintptr_t)dof;
12214         dof_relohdr_t *dofr =
12215             (dof_relohdr_t *)(uintptr_t)(daddr + sec->dofs_offset);
12216         dof_sec_t *ss, *rs, *ts;
12217         dof_relodesc_t *r;
12218         uint_t i, n;

12220         if (sec->dofs_size < sizeof (dof_relohdr_t) ||
12221             sec->dofs_align != sizeof (dof_secidx_t)) {
12222                 dtrace_dof_error(dof, "invalid relocation header");
12223                 return (-1);
12224         }

12226         ss = dtrace_dof_sect(dof, DOF_SECT_STRTAB, dofr->dofr_strtab);
12227         rs = dtrace_dof_sect(dof, DOF_SECT_RELTAB, dofr->dofr_relsec);
12228         ts = dtrace_dof_sect(dof, DOF_SECT_NONE, dofr->dofr_tgtsec);

12230         if (ss == NULL || rs == NULL || ts == NULL)
12231                 return (-1); /* dtrace_dof_error() has been called already */

12233         if (rs->dofs_entsize < sizeof (dof_relodesc_t) ||
12234             rs->dofs_align != sizeof (uint64_t)) {
12235                 dtrace_dof_error(dof, "invalid relocation section");
12236                 return (-1);
12237         }

12239         r = (dof_relodesc_t *)(uintptr_t)(daddr + rs->dofs_offset);
12240         n = rs->dofs_size / rs->dofs_entsize;

12242         for (i = 0; i < n; i++) {
12243                 uintptr_t taddr = daddr + ts->dofs_offset + r->dofr_offset;

12245                 switch (r->dofr_type) {
12246                 case DOF_RELO_NONE:
12247                         break;
12248                 case DOF_RELO_SETX:
12249                         if (r->dofr_offset >= ts->dofs_size || r->dofr_offset +
12250                             sizeof (uint64_t) > ts->dofs_size) {
12251                                 dtrace_dof_error(dof, "bad relocation offset");
12252                                 return (-1);
12253                         }

12255                         if (!IS_P2ALIGNED(taddr, sizeof (uint64_t))) {
12256                                 dtrace_dof_error(dof, "misaligned setx relo");
12257                                 return (-1);
12258                         }

12260                         *(uint64_t *)taddr += ubase;
12261                         break;
12262                 default:
12263                         dtrace_dof_error(dof, "invalid relocation type");
12264                         return (-1);
12265                 }

12267                 r = (dof_relodesc_t *)((uintptr_t)r + rs->dofs_entsize);
12268         }

12270         return (0);
12271 }
```



```
12273 /*
12274  * The dof_hdr_t passed to dtrace_dof_slurp() should be a partially validated
12275  * header:  it should be at the front of a memory region that is at least
12276  * sizeof (dof_hdr_t) in size -- and then at least dof_hdr.dofh_loadsz in
12277  * size.  It need not be validated in any other way.
12278  */
12279 static int
12280 dtrace_dof_slurp(dof_hdr_t *dof, dtrace_vstate_t *vstate, cred_t *cr,
12281     dtrace_enabling_t **enabp, uint64_t ubase, int noprobes)
12282 {
12283         uint64_t len = dof->dofh_loadsz, seclen;
12284         uintptr_t daddr = (uintptr_t)dof;
12285         dtrace_ecbdesc_t *ep;
12286         dtrace_enabling_t *enab;
12287         uint_t i;

12289         ASSERT(MUTEX_HELD(&dtrace_lock));
12290         ASSERT(dof->dofh_loadsz >= sizeof (dof_hdr_t));

12292         /*
12293          * Check the DOF header identification bytes.  In addition to checking
12294          * valid settings, we also verify that unused bits/bytes are zeroed so
12295          * we can use them later without fear of regressing existing binaries.
12296          */
12297         if (bcmp(&dof->dofh_ident[DOF_ID_MAG0],
12298             DOF_MAG_STRING, DOF_MAG_STRLEN) != 0) {
12299                 dtrace_dof_error(dof, "DOF magic string mismatch");
12300                 return (-1);
12301         }

12303         if (dof->dofh_ident[DOF_ID_MODEL] != DOF_MODEL_ILP32 &&
12304             dof->dofh_ident[DOF_ID_MODEL] != DOF_MODEL_LP64) {
12305                 dtrace_dof_error(dof, "DOF has invalid data model");
12306                 return (-1);
12307         }

12309         if (dof->dofh_ident[DOF_ID_ENCODING] != DOF_ENCODE_NATIVE) {
12310                 dtrace_dof_error(dof, "DOF encoding mismatch");
12311                 return (-1);
12312         }

12314         if (dof->dofh_ident[DOF_ID_VERSION] != DOF_VERSION_1 &&
12315             dof->dofh_ident[DOF_ID_VERSION] != DOF_VERSION_2) {
12316                 dtrace_dof_error(dof, "DOF version mismatch");
12317                 return (-1);
12318         }

12320         if (dof->dofh_ident[DOF_ID_DIFVERS] != DIF_VERSION_2) {
12321                 dtrace_dof_error(dof, "DOF uses unsupported instruction set");
12322                 return (-1);
12323         }

12325         if (dof->dofh_ident[DOF_ID_DIFIREG] > DIF_DIR_NREGS) {
12326                 dtrace_dof_error(dof, "DOF uses too many integer registers");
12327                 return (-1);
12328         }

12330         if (dof->dofh_ident[DOF_ID_DIFTREG] > DIF_DTR_NREGS) {
12331                 dtrace_dof_error(dof, "DOF uses too many tuple registers");
12332                 return (-1);
12333         }

12335         for (i = DOF_ID_PAD; i < DOF_ID_SIZE; i++) {
12336                 if (dof->dofh_ident[i] != 0) {
12337                         dtrace_dof_error(dof, "DOF has invalid ident byte set");
```

```
12338                           return (-1);
12339                   }
12340           }

12342           if (dof->dofh_flags & ~DOF_FL_VALID) {
12343                   dtrace_dof_error(dof, "DOF has invalid flag bits set");
12344                   return (-1);
12345           }

12347           if (dof->dofh_secsize == 0) {
12348                   dtrace_dof_error(dof, "zero section header size");
12349                   return (-1);
12350           }

12352           /*
12353            * Check that the section headers don't exceed the amount of DOF
12354            * data.  Note that we cast the section size and number of sections
12355            * to uint64_t's to prevent possible overflow in the multiplication.
12356            */
12357           seclen = (uint64_t)dof->dofh_secnum * (uint64_t)dof->dofh_secsize;

12359           if (dof->dofh_secoff > len || seclen > len ||
12360               dof->dofh_secoff + seclen > len) {
12361                   dtrace_dof_error(dof, "truncated section headers");
12362                   return (-1);
12363           }

12365           if (!IS_P2ALIGNED(dof->dofh_secoff, sizeof (uint64_t))) {
12366                   dtrace_dof_error(dof, "misaligned section headers");
12367                   return (-1);
12368           }

12370           if (!IS_P2ALIGNED(dof->dofh_secsize, sizeof (uint64_t))) {
12371                   dtrace_dof_error(dof, "misaligned section size");
12372                   return (-1);
12373           }

12375           /*
12376            * Take an initial pass through the section headers to be sure that
12377            * the headers don't have stray offsets.  If the 'noprobes' flag is
12378            * set, do not permit sections relating to providers, probes, or args.
12379            */
12380           for (i = 0; i < dof->dofh_secnum; i++) {
12381                   dof_sec_t *sec = (dof_sec_t *)(daddr +
12382                       (uintptr_t)dof->dofh_secoff + i * dof->dofh_secsize);

12384                   if (noprobes) {
12385                           switch (sec->dofs_type) {
12386                           case DOF_SECT_PROVIDER:
12387                           case DOF_SECT_PROBES:
12388                           case DOF_SECT_PRARGS:
12389                           case DOF_SECT_PROFFS:
12390                                   dtrace_dof_error(dof, "illegal sections "
12391                                       "for enabling");
12392                                   return (-1);
12393                           }
12394                   }

12396                   if (DOF_SEC_ISLOADABLE(sec->dofs_type) &&
12397                       !(sec->dofs_flags & DOF_SECF_LOAD)) {
12398                           dtrace_dof_error(dof, "loadable section with load "
12399                               "flag unset");
12400                           return (-1);
12401                   }

12403                   if (!(sec->dofs_flags & DOF_SECF_LOAD))
```



```
12404                           continue; /* just ignore non-loadable sections */

12406                   if (sec->dofs_align & (sec->dofs_align - 1)) {
12407                           dtrace_dof_error(dof, "bad section alignment");
12408                           return (-1);
12409                   }

12411                   if (sec->dofs_offset & (sec->dofs_align - 1)) {
12412                           dtrace_dof_error(dof, "misaligned section");
12413                           return (-1);
12414                   }

12416                   if (sec->dofs_offset > len || sec->dofs_size > len ||
12417                       sec->dofs_offset + sec->dofs_size > len) {
12418                           dtrace_dof_error(dof, "corrupt section header");
12419                           return (-1);
12420                   }

12422                   if (sec->dofs_type == DOF_SECT_STRTAB && *((char *)daddr +
12423                       sec->dofs_offset + sec->dofs_size - 1) != '\0') {
12424                           dtrace_dof_error(dof, "non-terminating string table");
12425                           return (-1);
12426                   }
12427           }

12429           /*
12430            * Take a second pass through the sections and locate and perform any
12431            * relocations that are present.  We do this after the first pass to
12432            * be sure that all sections have had their headers validated.
12433            */
12434           for (i = 0; i < dof->dofh_secnum; i++) {
12435                   dof_sec_t *sec = (dof_sec_t *)(daddr +
12436                       (uintptr_t)dof->dofh_secoff + i * dof->dofh_secsize);

12438                   if (!(sec->dofs_flags & DOF_SECF_LOAD))
12439                           continue; /* skip sections that are not loadable */

12441                   switch (sec->dofs_type) {
12442                   case DOF_SECT_URELHDR:
12443                           if (dtrace_dof_relocate(dof, sec, ubase) != 0)
12444                                   return (-1);
12445                           break;
12446                   }
12447           }

12449           if ((enab = *enabp) == NULL)
12450                   enab = *enabp = dtrace_enabling_create(vstate);

12452           for (i = 0; i < dof->dofh_secnum; i++) {
12453                   dof_sec_t *sec = (dof_sec_t *)(daddr +
12454                       (uintptr_t)dof->dofh_secoff + i * dof->dofh_secsize);

12456                   if (sec->dofs_type != DOF_SECT_ECBDESC)
12457                           continue;

12459                   if ((ep = dtrace_dof_ecbdesc(dof, sec, vstate, cr)) == NULL) {
12460                           dtrace_enabling_destroy(enab);
12461                           *enabp = NULL;
12462                           return (-1);
12463                   }

12465                   dtrace_enabling_add(enab, ep);
12466           }

12468           return (0);
12469   }
```

```
12471 /*
12472  * Process DOF for any options.  This routine assumes that the DOF has been
12473  * at least processed by dtrace_dof_slurp().
12474  */
12475 static int
12476 dtrace_dof_options(dof_hdr_t *dof, dtrace_state_t *state)
12477 {
12478         int i, rval;
12479         uint32_t entsize;
12480         size_t offs;
12481         dof_optdesc_t *desc;

12483         for (i = 0; i < dof->dofh_secnum; i++) {
12484                 dof_sec_t *sec = (dof_sec_t *)((uintptr_t)dof +
12485                     (uintptr_t)dof->dofh_secoff + i * dof->dofh_secsize);

12487                 if (sec->dofs_type != DOF_SECT_OPTDESC)
12488                         continue;

12490                 if (sec->dofs_align != sizeof (uint64_t)) {
12491                         dtrace_dof_error(dof, "bad alignment in "
12492                             "option description");
12493                         return (EINVAL);
12494                 }

12496                 if ((entsize = sec->dofs_entsize) == 0) {
12497                         dtrace_dof_error(dof, "zeroed option entry size");
12498                         return (EINVAL);
12499                 }

12501                 if (entsize < sizeof (dof_optdesc_t)) {
12502                         dtrace_dof_error(dof, "bad option entry size");
12503                         return (EINVAL);
12504                 }

12506                 for (offs = 0; offs < sec->dofs_size; offs += entsize) {
12507                         desc = (dof_optdesc_t *)((uintptr_t)dof +
12508                             (uintptr_t)sec->dofs_offset + offs);

12510                         if (desc->dofo_strtab != DOF_SECIDX_NONE) {
12511                                 dtrace_dof_error(dof, "non-zero option string");
12512                                 return (EINVAL);
12513                         }

12515                         if (desc->dofo_value == DTRACEOPT_UNSET) {
12516                                 dtrace_dof_error(dof, "unset option");
12517                                 return (EINVAL);
12518                         }

12520                         if ((rval = dtrace_state_option(state,
12521                             desc->dofo_option, desc->dofo_value)) != 0) {
12522                                 dtrace_dof_error(dof, "rejected option");
12523                                 return (rval);
12524                         }
12525                 }
12526         }

12528         return (0);
12529 }

12531 /*
12532  * DTrace Consumer State Functions
12533  */
12534 int
12535 dtrace_dstate_init(dtrace_dstate_t *dstate, size_t size)
```



```
12536 {
12537         size_t hashsize, maxper, min, chunksize = dstate->dtds_chunksize;
12538         void *base;
12539         uintptr_t limit;
12540         dtrace_dynvar_t *dvar, *next, *start;
12541         int i;

12543         ASSERT(MUTEX_HELD(&dtrace_lock));
12544         ASSERT(dstate->dtds_base == NULL && dstate->dtds_percpu == NULL);

12546         bzero(dstate, sizeof (dtrace_dstate_t));

12548         if ((dstate->dtds_chunksize = chunksize) == 0)
12549                 dstate->dtds_chunksize = DTRACE_DYNVAR_CHUNKSIZE;

12551         if (size < (min = dstate->dtds_chunksize + sizeof (dtrace_dynhash_t)))
12552                 size = min;

12554         if ((base = kmem_zalloc(size, KM_NOSLEEP | KM_NORMALPRI)) == NULL)
12555                 return (ENOMEM);

12557         dstate->dtds_size = size;
12558         dstate->dtds_base = base;
12559         dstate->dtds_percpu = kmem_cache_alloc(dtrace_state_cache, KM_SLEEP);
12560         bzero(dstate->dtds_percpu, NCPU * sizeof (dtrace_dstate_percpu_t));

12562         hashsize = size / (dstate->dtds_chunksize + sizeof (dtrace_dynhash_t));

12564         if (hashsize != 1 && (hashsize & 1))
12565                 hashsize--;

12567         dstate->dtds_hashsize = hashsize;
12568         dstate->dtds_hash = dstate->dtds_base;

12570         /*
12571          * Set all of our hash buckets to point to the single sink, and (if
12572          * it hasn't already been set), set the sink's hash value to be the
12573          * sink sentinel value.  The sink is needed for dynamic variable
12574          * lookups to know that they have iterated over an entire, valid hash
12575          * chain.
12576          */
12577         for (i = 0; i < hashsize; i++)
12578                 dstate->dtds_hash[i].dtdh_chain = &dtrace_dynhash_sink;

12580         if (dtrace_dynhash_sink.dtdv_hashval != DTRACE_DYNHASH_SINK)
12581                 dtrace_dynhash_sink.dtdv_hashval = DTRACE_DYNHASH_SINK;

12583         /*
12584          * Determine number of active CPUs.  Divide free list evenly among
12585          * active CPUs.
12586          */
12587         start = (dtrace_dynvar_t *)
12588             ((uintptr_t)base + hashsize * sizeof (dtrace_dynhash_t));
12589         limit = (uintptr_t)base + size;

12591         maxper = (limit - (uintptr_t)start) / NCPU;
12592         maxper = (maxper / dstate->dtds_chunksize) * dstate->dtds_chunksize;

12594         for (i = 0; i < NCPU; i++) {
12595                 dstate->dtds_percpu[i].dtdsc_free = dvar = start;

12597                 /*
12598                  * If we don't even have enough chunks to make it once through
12599                  * NCPUs, we're just going to allocate everything to the first
12600                  * CPU.  And if we're on the last CPU, we're going to allocate
12601                  * whatever is left over.  In either case, we set the limit to
```

```
12602                  * be the limit of the dynamic variable space.
12603                  */
12604                 if (maxper == 0 || i == NCPU - 1) {
12605                         limit = (uintptr_t)base + size;
12606                         start = NULL;
12607                 } else {
12608                         limit = (uintptr_t)start + maxper;
12609                         start = (dtrace_dynvar_t *)limit;
12610                 }

12612                 ASSERT(limit <= (uintptr_t)base + size);

12614                 for (;;) {
12615                         next = (dtrace_dynvar_t *)((uintptr_t)dvar +
12616                             dstate->dtds_chunksize);

12618                         if ((uintptr_t)next + dstate->dtds_chunksize >= limit)
12619                                 break;

12621                         dvar->dtdv_next = next;
12622                         dvar = next;
12623                 }

12625                 if (maxper == 0)
12626                         break;
12627         }

12629         return (0);
12630 }

12632 void
12633 dtrace_dstate_fini(dtrace_dstate_t *dstate)
12634 {
12635         ASSERT(MUTEX_HELD(&cpu_lock));

12637         if (dstate->dtds_base == NULL)
12638                 return;

12640         kmem_free(dstate->dtds_base, dstate->dtds_size);
12641         kmem_cache_free(dtrace_state_cache, dstate->dtds_percpu);
12642 }

12644 static void
12645 dtrace_vstate_fini(dtrace_vstate_t *vstate)
12646 {
12647         /*
12648          * Logical XOR, where are you?
12649          */
12650         ASSERT((vstate->dtvs_nglobals == 0) ^ (vstate->dtvs_globals != NULL));

12652         if (vstate->dtvs_nglobals > 0) {
12653                 kmem_free(vstate->dtvs_globals, vstate->dtvs_nglobals *
12654                     sizeof (dtrace_statvar_t *));
12655         }

12657         if (vstate->dtvs_ntlocals > 0) {
12658                 kmem_free(vstate->dtvs_tlocals, vstate->dtvs_ntlocals *
12659                     sizeof (dtrace_difv_t));
12660         }

12662         ASSERT((vstate->dtvs_nlocals == 0) ^ (vstate->dtvs_locals != NULL));

12664         if (vstate->dtvs_nlocals > 0) {
12665                 kmem_free(vstate->dtvs_locals, vstate->dtvs_nlocals *
12666                     sizeof (dtrace_statvar_t *));
12667         }
```



```
12668 }

12670 static void
12671 dtrace_state_clean(dtrace_state_t *state)
12672 {
12673         if (state->dts_activity == DTRACE_ACTIVITY_INACTIVE)
12674                 return;

12676         dtrace_dynvar_clean(&state->dts_vstate.dtvs_dynvars);
12677         dtrace_speculation_clean(state);
12678 }

12680 static void
12681 dtrace_state_deadman(dtrace_state_t *state)
12682 {
12683         hrtime_t now;

12685         dtrace_sync();

12687         now = dtrace_gethrtime();

12689         if (state != dtrace_anon.dta_state &&
12690             now - state->dts_laststatus >= dtrace_deadman_user)
12691                 return;

12693         /*
12694          * We must be sure that dts_alive never appears to be less than the
12695          * value upon entry to dtrace_state_deadman(), and because we lack a
12696          * dtrace_cas64(), we cannot store to it atomically.  We thus instead
12697          * store INT64_MAX to it, followed by a memory barrier, followed by
12698          * the new value.  This assures that dts_alive never appears to be
12699          * less than its true value, regardless of the order in which the
12700          * stores to the underlying storage are issued.
12701          */
12702         state->dts_alive = INT64_MAX;
12703         dtrace_membar_producer();
12704         state->dts_alive = now;
12705 }

12707 dtrace_state_t *
12708 dtrace_state_create(dev_t *devp, cred_t *cr)
12709 {
12710         minor_t minor;
12711         major_t major;
12712         char c[30];
12713         dtrace_state_t *state;
12714         dtrace_optval_t *opt;
12715         int bufsize = NCPU * sizeof (dtrace_buffer_t), i;

12717         ASSERT(MUTEX_HELD(&dtrace_lock));
12718         ASSERT(MUTEX_HELD(&cpu_lock));

12720         minor = (minor_t)(uintptr_t)vmem_alloc(dtrace_minor, 1,
12721             VM_BESTFIT | VM_SLEEP);

12723         if (ddi_soft_state_zalloc(dtrace_softstate, minor) != DDI_SUCCESS) {
12724                 vmem_free(dtrace_minor, (void *)(uintptr_t)minor, 1);
12725                 return (NULL);
12726         }

12728         state = ddi_get_soft_state(dtrace_softstate, minor);
12729         state->dts_epid = DTRACE_EPIDNONE + 1;

12731         (void) snprintf(c, sizeof (c), "dtrace_aggid_%d", minor);
12732         state->dts_aggid_arena = vmem_create(c, (void *)1, UINT32_MAX, 1,
12733             NULL, NULL, NULL, 0, VM_SLEEP | VMC_IDENTIFIER);
```

```
12735         if (devp != NULL) {
12736                 major = getemajor(*devp);
12737         } else {
12738                 major = ddi_driver_major(dtrace_devi);
12739         }

12741         state->dts_dev = makedevice(major, minor);

12743         if (devp != NULL)
12744                 *devp = state->dts_dev;

12746         /*
12747          * We allocate NCPU buffers.  On the one hand, this can be quite
12748          * a bit of memory per instance (nearly 36K on a Starcat).  On the
12749          * other hand, it saves an additional memory reference in the probe
12750          * path.
12751          */
12752         state->dts_buffer = kmem_zalloc(bufsize, KM_SLEEP);
12753         state->dts_aggbuffer = kmem_zalloc(bufsize, KM_SLEEP);
12754         state->dts_cleaner = CYCLIC_NONE;
12755         state->dts_deadman = CYCLIC_NONE;
12756         state->dts_vstate.dtvs_state = state;

12758         for (i = 0; i < DTRACEOPT_MAX; i++)
12759                 state->dts_options[i] = DTRACEOPT_UNSET;

12761         /*
12762          * Set the default options.
12763          */
12764         opt = state->dts_options;
12765         opt[DTRACEOPT_BUFPOLICY] = DTRACEOPT_BUFPOLICY_SWITCH;
12766         opt[DTRACEOPT_BUFRESIZE] = DTRACEOPT_BUFRESIZE_AUTO;
12767         opt[DTRACEOPT_NSPEC] = dtrace_nspec_default;
12768         opt[DTRACEOPT_SPECSIZE] = dtrace_specsize_default;
12769         opt[DTRACEOPT_CPU] = (dtrace_optval_t)DTRACE_CPUALL;
12770         opt[DTRACEOPT_STRSIZE] = dtrace_strsize_default;
12771         opt[DTRACEOPT_STACKFRAMES] = dtrace_stackframes_default;
12772         opt[DTRACEOPT_USTACKFRAMES] = dtrace_ustackframes_default;
12773         opt[DTRACEOPT_CLEANRATE] = dtrace_cleanrate_default;
12774         opt[DTRACEOPT_AGGRATE] = dtrace_aggrate_default;
12775         opt[DTRACEOPT_SWITCHRATE] = dtrace_switchrate_default;
12776         opt[DTRACEOPT_STATUSRATE] = dtrace_statusrate_default;
12777         opt[DTRACEOPT_JSTACKFRAMES] = dtrace_jstackframes_default;
12778         opt[DTRACEOPT_JSTACKSTRSIZE] = dtrace_jstackstrsize_default;

12780         state->dts_activity = DTRACE_ACTIVITY_INACTIVE;

12782         /*
12783          * Depending on the user credentials, we set flag bits which alter probe
12784          * visibility or the amount of destructiveness allowed.  In the case of
12785          * actual anonymous tracing, or the possession of all privileges, all of
12786          * the normal checks are bypassed.
12787          */
12788         if (cr == NULL || PRIV_POLICY_ONLY(cr, PRIV_ALL, B_FALSE)) {
12789                 state->dts_cred.dcr_visible = DTRACE_CRV_ALL;
12790                 state->dts_cred.dcr_action = DTRACE_CRA_ALL;
12791         } else {
12792                 /*
12793                  * Set up the credentials for this instantiation.  We take a
12794                  * hold on the credential to prevent it from disappearing on
12795                  * us; this in turn prevents the zone_t referenced by this
12796                  * credential from disappearing.  This means that we can
12797                  * examine the credential and the zone from probe context.
12798                  */
12799                 crhold(cr);
```



```
12800                 state->dts_cred.dcr_cred = cr;

12802                 /*
12803                  * CRA_PROC means "we have *some* privilege for dtrace" and
12804                  * unlocks the use of variables like pid, zonename, etc.
12805                  */
12806                 if (PRIV_POLICY_ONLY(cr, PRIV_DTRACE_USER, B_FALSE) ||
12807                     PRIV_POLICY_ONLY(cr, PRIV_DTRACE_PROC, B_FALSE)) {
12808                         state->dts_cred.dcr_action |= DTRACE_CRA_PROC;
12809                 }

12811                 /*
12812                  * dtrace_user allows use of syscall and profile providers.
12813                  * If the user also has proc_owner and/or proc_zone, we
12814                  * extend the scope to include additional visibility and
12815                  * destructive power.
12816                  */
12817                 if (PRIV_POLICY_ONLY(cr, PRIV_DTRACE_USER, B_FALSE)) {
12818                         if (PRIV_POLICY_ONLY(cr, PRIV_PROC_OWNER, B_FALSE)) {
12819                                 state->dts_cred.dcr_visible |=
12820                                     DTRACE_CRV_ALLPROC;

12822                                 state->dts_cred.dcr_action |=
12823                                     DTRACE_CRA_PROC_DESTRUCTIVE_ALLUSER;
12824                         }

12826                         if (PRIV_POLICY_ONLY(cr, PRIV_PROC_ZONE, B_FALSE)) {
12827                                 state->dts_cred.dcr_visible |=
12828                                     DTRACE_CRV_ALLZONE;

12830                                 state->dts_cred.dcr_action |=
12831                                     DTRACE_CRA_PROC_DESTRUCTIVE_ALLZONE;
12832                         }

12834                         /*
12835                          * If we have all privs in whatever zone this is,
12836                          * we can do destructive things to processes which
12837                          * have altered credentials.
12838                          */
12839                         if (priv_isequalset(priv_getset(cr, PRIV_EFFECTIVE),
12840                             cr->cr_zone->zone_privset)) {
12841                                 state->dts_cred.dcr_action |=
12842                                     DTRACE_CRA_PROC_DESTRUCTIVE_CREDCHG;
12843                         }
12844                 }

12846                 /*
12847                  * Holding the dtrace_kernel privilege also implies that
12848                  * the user has the dtrace_user privilege from a visibility
12849                  * perspective.  But without further privileges, some
12850                  * destructive actions are not available.
12851                  */
12852                 if (PRIV_POLICY_ONLY(cr, PRIV_DTRACE_KERNEL, B_FALSE)) {
12853                         /*
12854                          * Make all probes in all zones visible.  However,
12855                          * this doesn't mean that all actions become available
12856                          * to all zones.
12857                          */
12858                         state->dts_cred.dcr_visible |= DTRACE_CRV_KERNEL |
12859                             DTRACE_CRV_ALLPROC | DTRACE_CRV_ALLZONE;

12861                         state->dts_cred.dcr_action |= DTRACE_CRA_KERNEL |
12862                             DTRACE_CRA_PROC;
12863                         /*
12864                          * Holding proc_owner means that destructive actions
12865                          * for *this* zone are allowed.
```

```
12866                          */
12867                         if (PRIV_POLICY_ONLY(cr, PRIV_PROC_OWNER, B_FALSE))
12868                                 state->dts_cred.dcr_action |=
12869                                     DTRACE_CRA_PROC_DESTRUCTIVE_ALLUSER;
12870
12871                         /*
12872                          * Holding proc_zone means that destructive actions
12873                          * for this user/group ID in all zones is allowed.
12874                          */
12875                         if (PRIV_POLICY_ONLY(cr, PRIV_PROC_ZONE, B_FALSE))
12876                                 state->dts_cred.dcr_action |=
12877                                     DTRACE_CRA_PROC_DESTRUCTIVE_ALLZONE;
12878
12879                         /*
12880                          * If we have all privs in whatever zone this is,
12881                          * we can do destructive things to processes which
12882                          * have altered credentials.
12883                          */
12884                         if (priv_isequalset(priv_getset(cr, PRIV_EFFECTIVE),
12885                             cr->cr_zone->zone_privset)) {
12886                                 state->dts_cred.dcr_action |=
12887                                     DTRACE_CRA_PROC_DESTRUCTIVE_CREDCHG;
12888                         }
12889                 }
12890
12891                 /*
12892                  * Holding the dtrace_proc privilege gives control over fasttrap
12893                  * and pid providers.  We need to grant wider destructive
12894                  * privileges in the event that the user has proc_owner and/or
12895                  * proc_zone.
12896                  */
12897                 if (PRIV_POLICY_ONLY(cr, PRIV_DTRACE_PROC, B_FALSE)) {
12898                         if (PRIV_POLICY_ONLY(cr, PRIV_PROC_OWNER, B_FALSE))
12899                                 state->dts_cred.dcr_action |=
12900                                     DTRACE_CRA_PROC_DESTRUCTIVE_ALLUSER;
12901
12902                         if (PRIV_POLICY_ONLY(cr, PRIV_PROC_ZONE, B_FALSE))
12903                                 state->dts_cred.dcr_action |=
12904                                     DTRACE_CRA_PROC_DESTRUCTIVE_ALLZONE;
12905                 }
12906         }
12907
12908         return (state);
12909 }
12910
12911 static int
12912 dtrace_state_buffer(dtrace_state_t *state, dtrace_buffer_t *buf, int which)
12913 {
12914         dtrace_optval_t *opt = state->dts_options, size;
12915         processorid_t cpu;
12916         int flags = 0, rval, factor, divisor = 1;
12917
12918         ASSERT(MUTEX_HELD(&dtrace_lock));
12919         ASSERT(MUTEX_HELD(&cpu_lock));
12920         ASSERT(which < DTRACEOPT_MAX);
12921         ASSERT(state->dts_activity == DTRACE_ACTIVITY_INACTIVE ||
12922             (state == dtrace_anon.dta_state &&
12923             state->dts_activity == DTRACE_ACTIVITY_ACTIVE));
12924
12925         if (opt[which] == DTRACEOPT_UNSET || opt[which] == 0)
12926                 return (0);
12927
12928         if (opt[DTRACEOPT_CPU] != DTRACEOPT_UNSET)
12929                 cpu = opt[DTRACEOPT_CPU];
12930
12931         if (which == DTRACEOPT_SPECSIZE)
```



```
12932                 flags |= DTRACEBUF_NOSWITCH;
12933
12934         if (which == DTRACEOPT_BUFSIZE) {
12935                 if (opt[DTRACEOPT_BUFPOLICY] == DTRACEOPT_BUFPOLICY_RING)
12936                         flags |= DTRACEBUF_RING;
12937
12938                 if (opt[DTRACEOPT_BUFPOLICY] == DTRACEOPT_BUFPOLICY_FILL)
12939                         flags |= DTRACEBUF_FILL;
12940
12941                 if (state != dtrace_anon.dta_state ||
12942                     state->dts_activity != DTRACE_ACTIVITY_ACTIVE)
12943                         flags |= DTRACEBUF_INACTIVE;
12944         }
12945
12946         for (size = opt[which]; size >= sizeof (uint64_t); size /= divisor) {
12947                 /*
12948                  * The size must be 8-byte aligned.  If the size is not 8-byte
12949                  * aligned, drop it down by the difference.
12950                  */
12951                 if (size & (sizeof (uint64_t) - 1))
12952                         size -= size & (sizeof (uint64_t) - 1);
12953
12954                 if (size < state->dts_reserve) {
12955                         /*
12956                          * Buffers always must be large enough to accommodate
12957                          * their prereserved space.  We return E2BIG instead
12958                          * of ENOMEM in this case to allow for user-level
12959                          * software to differentiate the cases.
12960                          */
12961                         return (E2BIG);
12962                 }
12963
12964                 rval = dtrace_buffer_alloc(buf, size, flags, cpu, &factor);
12965
12966                 if (rval != ENOMEM) {
12967                         opt[which] = size;
12968                         return (rval);
12969                 }
12970
12971                 if (opt[DTRACEOPT_BUFRESIZE] == DTRACEOPT_BUFRESIZE_MANUAL)
12972                         return (rval);
12973
12974                 for (divisor = 2; divisor < factor; divisor <<= 1)
12975                         continue;
12976         }
12977
12978         return (ENOMEM);
12979 }
12980
12981 static int
12982 dtrace_state_buffers(dtrace_state_t *state)
12983 {
12984         dtrace_speculation_t *spec = state->dts_speculations;
12985         int rval, i;
12986
12987         if ((rval = dtrace_state_buffer(state, state->dts_buffer,
12988             DTRACEOPT_BUFSIZE)) != 0)
12989                 return (rval);
12990
12991         if ((rval = dtrace_state_buffer(state, state->dts_aggbuffer,
12992             DTRACEOPT_AGGSIZE)) != 0)
12993                 return (rval);
12994
12995         for (i = 0; i < state->dts_nspeculations; i++) {
12996                 if ((rval = dtrace_state_buffer(state,
12997                     spec[i].dtsp_buffer, DTRACEOPT_SPECSIZE)) != 0)
```

```
12998                           return (rval);
12999           }
13000
13001           return (0);
13002   }
13003
13004   static void
13005   dtrace_state_prereserve(dtrace_state_t *state)
13006   {
13007           dtrace_ecb_t *ecb;
13008           dtrace_probe_t *probe;
13009
13010           state->dts_reserve = 0;
13011
13012           if (state->dts_options[DTRACEOPT_BUFPOLICY] != DTRACEOPT_BUFPOLICY_FILL)
13013                   return;
13014
13015           /*
13016            * If our buffer policy is a "fill" buffer policy, we need to set the
13017            * prereserved space to be the space required by the END probes.
13018            */
13019           probe = dtrace_probes[dtrace_probeid_end - 1];
13020           ASSERT(probe != NULL);
13021
13022           for (ecb = probe->dtpr_ecb; ecb != NULL; ecb = ecb->dte_next) {
13023                   if (ecb->dte_state != state)
13024                           continue;
13025
13026                   state->dts_reserve += ecb->dte_needed + ecb->dte_alignment;
13027           }
13028   }
13029
13030   static int
13031   dtrace_state_go(dtrace_state_t *state, processorid_t *cpu)
13032   {
13033           dtrace_optval_t *opt = state->dts_options, sz, nspec;
13034           dtrace_speculation_t *spec;
13035           dtrace_buffer_t *buf;
13036           cyc_handler_t hdlr;
13037           cyc_time_t when;
13038           int rval = 0, i, bufsize = NCPU * sizeof (dtrace_buffer_t);
13039           dtrace_icookie_t cookie;
13040
13041           mutex_enter(&cpu_lock);
13042           mutex_enter(&dtrace_lock);
13043
13044           if (state->dts_activity != DTRACE_ACTIVITY_INACTIVE) {
13045                   rval = EBUSY;
13046                   goto out;
13047           }
13048
13049           /*
13050            * Before we can perform any checks, we must prime all of the
13051            * retained enablings that correspond to this state.
13052            */
13053           dtrace_enabling_prime(state);
13054
13055           if (state->dts_destructive && !state->dts_cred.dcr_destructive) {
13056                   rval = EACCES;
13057                   goto out;
13058           }
13059
13060           dtrace_state_prereserve(state);
13061
13062           /*
13063            * Now we want to do is try to allocate our speculations.
```



```
13064            * We do not automatically resize the number of speculations; if
13065            * this fails, we will fail the operation.
13066            */
13067           nspec = opt[DTRACEOPT_NSPEC];
13068           ASSERT(nspec != DTRACEOPT_UNSET);
13069
13070           if (nspec > INT_MAX) {
13071                   rval = ENOMEM;
13072                   goto out;
13073           }
13074
13075           spec = kmem_zalloc(nspec * sizeof (dtrace_speculation_t),
13076               KM_NOSLEEP | KM_NORMALPRI);
13077
13078           if (spec == NULL) {
13079                   rval = ENOMEM;
13080                   goto out;
13081           }
13082
13083           state->dts_speculations = spec;
13084           state->dts_nspeculations = (int)nspec;
13085
13086           for (i = 0; i < nspec; i++) {
13087                   if ((buf = kmem_zalloc(bufsize,
13088                       KM_NOSLEEP | KM_NORMALPRI)) == NULL) {
13089                           rval = ENOMEM;
13090                           goto err;
13091                   }
13092
13093                   spec[i].dtsp_buffer = buf;
13094           }
13095
13096           if (opt[DTRACEOPT_GRABANON] != DTRACEOPT_UNSET) {
13097                   if (dtrace_anon.dta_state == NULL) {
13098                           rval = ENOENT;
13099                           goto out;
13100                   }
13101
13102                   if (state->dts_necbs != 0) {
13103                           rval = EALREADY;
13104                           goto out;
13105                   }
13106
13107                   state->dts_anon = dtrace_anon_grab();
13108                   ASSERT(state->dts_anon != NULL);
13109                   state = state->dts_anon;
13110
13111                   /*
13112                    * We want "grabanon" to be set in the grabbed state, so we'll
13113                    * copy that option value from the grabbing state into the
13114                    * grabbed state.
13115                    */
13116                   state->dts_options[DTRACEOPT_GRABANON] =
13117                       opt[DTRACEOPT_GRABANON];
13118
13119                   *cpu = dtrace_anon.dta_beganon;
13120
13121                   /*
13122                    * If the anonymous state is active (as it almost certainly
13123                    * is if the anonymous enabling ultimately matched anything),
13124                    * we don't allow any further option processing -- but we
13125                    * don't return failure.
13126                    */
13127                   if (state->dts_activity != DTRACE_ACTIVITY_INACTIVE)
13128                           goto out;
13129           }
```

```
13131         if (opt[DTRACEOPT_AGGSIZE] != DTRACEOPT_UNSET &&
13132             opt[DTRACEOPT_AGGSIZE] != 0) {
13133                 if (state->dts_aggregations == NULL) {
13134                         /*
13135                          * We're not going to create an aggregation buffer
13136                          * because we don't have any ECBs that contain
13137                          * aggregations -- set this option to 0.
13138                          */
13139                         opt[DTRACEOPT_AGGSIZE] = 0;
13140                 } else {
13141                         /*
13142                          * If we have an aggregation buffer, we must also have
13143                          * a buffer to use as scratch.
13144                          */
13145                         if (opt[DTRACEOPT_BUFSIZE] == DTRACEOPT_UNSET ||
13146                             opt[DTRACEOPT_BUFSIZE] < state->dts_needed) {
13147                                 opt[DTRACEOPT_BUFSIZE] = state->dts_needed;
13148                         }
13149                 }
13150         }

13152         if (opt[DTRACEOPT_SPECSIZE] != DTRACEOPT_UNSET &&
13153             opt[DTRACEOPT_SPECSIZE] != 0) {
13154                 if (!state->dts_speculates) {
13155                         /*
13156                          * We're not going to create speculation buffers
13157                          * because we don't have any ECBs that actually
13158                          * speculate -- set the speculation size to 0.
13159                          */
13160                         opt[DTRACEOPT_SPECSIZE] = 0;
13161                 }
13162         }

13164         /*
13165          * The bare minimum size for any buffer that we're actually going to
13166          * do anything to is sizeof (uint64_t).
13167          */
13168         sz = sizeof (uint64_t);

13170         if ((state->dts_needed != 0 && opt[DTRACEOPT_BUFSIZE] < sz) ||
13171             (state->dts_speculates && opt[DTRACEOPT_SPECSIZE] < sz) ||
13172             (state->dts_aggregations != NULL && opt[DTRACEOPT_AGGSIZE] < sz)) {
13173                 /*
13174                  * A buffer size has been explicitly set to 0 (or to a size
13175                  * that will be adjusted to 0) and we need the space -- we
13176                  * need to return failure.  We return ENOSPC to differentiate
13177                  * it from failing to allocate a buffer due to failure to meet
13178                  * the reserve (for which we return E2BIG).
13179                  */
13180                 rval = ENOSPC;
13181                 goto out;
13182         }

13184         if ((rval = dtrace_state_buffers(state)) != 0)
13185                 goto err;

13187         if ((sz = opt[DTRACEOPT_DYNVARSIZE]) == DTRACEOPT_UNSET)
13188                 sz = dtrace_dstate_defsize;

13190         do {
13191                 rval = dtrace_dstate_init(&state->dts_vstate.dtvs_dynvars, sz);

13193                 if (rval == 0)
13194                         break;
```



```
13196                 if (opt[DTRACEOPT_BUFRESIZE] == DTRACEOPT_BUFRESIZE_MANUAL)
13197                         goto err;
13198         } while (sz >>= 1);

13200         opt[DTRACEOPT_DYNVARSIZE] = sz;

13202         if (rval != 0)
13203                 goto err;

13205         if (opt[DTRACEOPT_STATUSRATE] > dtrace_statusrate_max)
13206                 opt[DTRACEOPT_STATUSRATE] = dtrace_statusrate_max;

13208         if (opt[DTRACEOPT_CLEANRATE] == 0)
13209                 opt[DTRACEOPT_CLEANRATE] = dtrace_cleanrate_max;

13211         if (opt[DTRACEOPT_CLEANRATE] < dtrace_cleanrate_min)
13212                 opt[DTRACEOPT_CLEANRATE] = dtrace_cleanrate_min;

13214         if (opt[DTRACEOPT_CLEANRATE] > dtrace_cleanrate_max)
13215                 opt[DTRACEOPT_CLEANRATE] = dtrace_cleanrate_max;

13217         hdlr.cyh_func = (cyc_func_t)dtrace_state_clean;
13218         hdlr.cyh_arg = state;
13219         hdlr.cyh_level = CY_LOW_LEVEL;

13221         when.cyt_when = 0;
13222         when.cyt_interval = opt[DTRACEOPT_CLEANRATE];

13224         state->dts_cleaner = cyclic_add(&hdlr, &when);

13226         hdlr.cyh_func = (cyc_func_t)dtrace_state_deadman;
13227         hdlr.cyh_arg = state;
13228         hdlr.cyh_level = CY_LOW_LEVEL;

13230         when.cyt_when = 0;
13231         when.cyt_interval = dtrace_deadman_interval;

13233         state->dts_alive = state->dts_laststatus = dtrace_gethrtime();
13234         state->dts_deadman = cyclic_add(&hdlr, &when);

13236         state->dts_activity = DTRACE_ACTIVITY_WARMUP;

13238         if (state->dts_getf != 0 &&
13239             !(state->dts_cred.dcr_visible & DTRACE_CRV_KERNEL)) {
13240                 /*
13241                  * We don't have kernel privs but we have at least one call
13242                  * to getf(); we need to bump our zone's count, and (if
13243                  * this is the first enabling to have an unprivileged call
13244                  * to getf()) we need to hook into closef().
13245                  */
13246                 state->dts_cred.dcr_cred->cr_zone->zone_dtrace_getf++;

13248                 if (dtrace_getf++ == 0) {
13249                         ASSERT(dtrace_closef == NULL);
13250                         dtrace_closef = dtrace_getf_barrier;
13251                 }
13252         }

13254 #endif /* ! codereview */
13255         /*
13256          * Now it's time to actually fire the BEGIN probe.  We need to disable
13257          * interrupts here both to record the CPU on which we fired the BEGIN
13258          * probe (the data from this CPU will be processed first at user
13259          * level) and to manually activate the buffer for this CPU.
13260          */
13261         cookie = dtrace_interrupt_disable();
```

```
13262         *cpu = CPU->cpu_id;
13263         ASSERT(state->dts_buffer[*cpu].dtb_flags & DTRACEBUF_INACTIVE);
13264         state->dts_buffer[*cpu].dtb_flags &= ~DTRACEBUF_INACTIVE;

13266         dtrace_probe(dtrace_probeid_begin,
13267             (uint64_t)(uintptr_t)state, 0, 0, 0, 0);
13268         dtrace_interrupt_enable(cookie);
13269         /*
13270          * We may have had an exit action from a BEGIN probe; only change our
13271          * state to ACTIVE if we're still in WARMUP.
13272          */
13273         ASSERT(state->dts_activity == DTRACE_ACTIVITY_WARMUP ||
13274             state->dts_activity == DTRACE_ACTIVITY_DRAINING);

13276         if (state->dts_activity == DTRACE_ACTIVITY_WARMUP)
13277                 state->dts_activity = DTRACE_ACTIVITY_ACTIVE;

13279         /*
13280          * Regardless of whether or not now we're in ACTIVE or DRAINING, we
13281          * want each CPU to transition its principal buffer out of the
13282          * INACTIVE state.  Doing this assures that no CPU will suddenly begin
13283          * processing an ECB halfway down a probe's ECB chain; all CPUs will
13284          * atomically transition from processing none of a state's ECBs to
13285          * processing all of them.
13286          */
13287         dtrace_xcall(DTRACE_CPUALL,
13288             (dtrace_xcall_t)dtrace_buffer_activate, state);
13289         goto out;

13291 err:
13292         dtrace_buffer_free(state->dts_buffer);
13293         dtrace_buffer_free(state->dts_aggbuffer);

13295         if ((nspec = state->dts_nspeculations) == 0) {
13296                 ASSERT(state->dts_speculations == NULL);
13297                 goto out;
13298         }

13300         spec = state->dts_speculations;
13301         ASSERT(spec != NULL);

13303         for (i = 0; i < state->dts_nspeculations; i++) {
13304                 if ((buf = spec[i].dtsp_buffer) == NULL)
13305                         break;

13307                 dtrace_buffer_free(buf);
13308                 kmem_free(buf, bufsize);
13309         }

13311         kmem_free(spec, nspec * sizeof (dtrace_speculation_t));
13312         state->dts_nspeculations = 0;
13313         state->dts_speculations = NULL;

13315 out:
13316         mutex_exit(&dtrace_lock);
13317         mutex_exit(&cpu_lock);

13319         return (rval);
13320 }

13322 static int
13323 dtrace_state_stop(dtrace_state_t *state, processorid_t *cpu)
13324 {
13325         dtrace_icookie_t cookie;

13327         ASSERT(MUTEX_HELD(&dtrace_lock));
```



```
13329         if (state->dts_activity != DTRACE_ACTIVITY_ACTIVE &&
13330             state->dts_activity != DTRACE_ACTIVITY_DRAINING)
13331                 return (EINVAL);

13333         /*
13334          * We'll set the activity to DTRACE_ACTIVITY_DRAINING, and issue a sync
13335          * to be sure that every CPU has seen it.  See below for the details
13336          * on why this is done.
13337          */
13338         state->dts_activity = DTRACE_ACTIVITY_DRAINING;
13339         dtrace_sync();

13341         /*
13342          * By this point, it is impossible for any CPU to be still processing
13343          * with DTRACE_ACTIVITY_ACTIVE.  We can thus set our activity to
13344          * DTRACE_ACTIVITY_COOLDOWN and know that we're not racing with any
13345          * other CPU in dtrace_buffer_reserve().  This allows dtrace_probe()
13346          * and callees to know that the activity is DTRACE_ACTIVITY_COOLDOWN
13347          * iff we're in the END probe.
13348          */
13349         state->dts_activity = DTRACE_ACTIVITY_COOLDOWN;
13350         dtrace_sync();
13351         ASSERT(state->dts_activity == DTRACE_ACTIVITY_COOLDOWN);

13353         /*
13354          * Finally, we can release the reserve and call the END probe.  We
13355          * disable interrupts across calling the END probe to allow us to
13356          * return the CPU on which we actually called the END probe.  This
13357          * allows user-land to be sure that this CPU's principal buffer is
13358          * processed last.
13359          */
13360         state->dts_reserve = 0;

13362         cookie = dtrace_interrupt_disable();
13363         *cpu = CPU->cpu_id;
13364         dtrace_probe(dtrace_probeid_end,
13365             (uint64_t)(uintptr_t)state, 0, 0, 0, 0);
13366         dtrace_interrupt_enable(cookie);

13368         state->dts_activity = DTRACE_ACTIVITY_STOPPED;
13369         dtrace_sync();

13371         if (state->dts_getf != 0 &&
13372             !(state->dts_cred.dcr_visible & DTRACE_CRV_KERNEL)) {
13373                 /*
13374                  * We don't have kernel privs but we have at least one call
13375                  * to getf(); we need to lower our zone's count, and (if
13376                  * this is the last enabling to have an unprivileged call
13377                  * to getf()) we need to clear the closef() hook.
13378                  */
13379                 ASSERT(state->dts_cred.dcr_cred->cr_zone->zone_dtrace_getf > 0);
13380                 ASSERT(dtrace_closef == dtrace_getf_barrier);
13381                 ASSERT(dtrace_getf > 0);

13383                 state->dts_cred.dcr_cred->cr_zone->zone_dtrace_getf--;

13385                 if (--dtrace_getf == 0)
13386                         dtrace_closef = NULL;
13387         }

13389 #endif /* ! codereview */
13390         return (0);
13391 }

13393 static int
```

```
13394 dtrace_state_option(dtrace_state_t *state, dtrace_optid_t option,
13395     dtrace_optval_t val)
13396 {
13397         ASSERT(MUTEX_HELD(&dtrace_lock));

13399         if (state->dts_activity != DTRACE_ACTIVITY_INACTIVE)
13400                 return (EBUSY);

13402         if (option >= DTRACEOPT_MAX)
13403                 return (EINVAL);

13405         if (option != DTRACEOPT_CPU && val < 0)
13406                 return (EINVAL);

13408         switch (option) {
13409         case DTRACEOPT_DESTRUCTIVE:
13410                 if (dtrace_destructive_disallow)
13411                         return (EACCES);

13413                 state->dts_cred.dcr_destructive = 1;
13414                 break;

13416         case DTRACEOPT_BUFSIZE:
13417         case DTRACEOPT_DYNVARSIZE:
13418         case DTRACEOPT_AGGSIZE:
13419         case DTRACEOPT_SPECSIZE:
13420         case DTRACEOPT_STRSIZE:
13421                 if (val < 0)
13422                         return (EINVAL);

13424                 if (val >= LONG_MAX) {
13425                         /*
13426                          * If this is an otherwise negative value, set it to
13427                          * the highest multiple of 128m less than LONG_MAX.
13428                          * Technically, we're adjusting the size without
13429                          * regard to the buffer resizing policy, but in fact,
13430                          * this has no effect -- if we set the buffer size to
13431                          * ~LONG_MAX and the buffer policy is ultimately set to
13432                          * be "manual", the buffer allocation is guaranteed to
13433                          * fail, if only because the allocation requires two
13434                          * buffers.  (We set the the size to the highest
13435                          * multiple of 128m because it ensures that the size
13436                          * will remain a multiple of a megabyte when
13437                          * repeatedly halved -- all the way down to 15m.)
13438                          */
13439                         val = LONG_MAX - (1 << 27) + 1;
13440                 }
13441         }

13443         state->dts_options[option] = val;

13445         return (0);
13446 }

13448 static void
13449 dtrace_state_destroy(dtrace_state_t *state)
13450 {
13451         dtrace_ecb_t *ecb;
13452         dtrace_vstate_t *vstate = &state->dts_vstate;
13453         minor_t minor = getminor(state->dts_dev);
13454         int i, bufsize = NCPU * sizeof (dtrace_buffer_t);
13455         dtrace_speculation_t *spec = state->dts_speculations;
13456         int nspec = state->dts_nspeculations;
13457         uint32_t match;

13459         ASSERT(MUTEX_HELD(&dtrace_lock));
```



```
13460         ASSERT(MUTEX_HELD(&cpu_lock));

13462         /*
13463          * First, retract any retained enablings for this state.
13464          */
13465         dtrace_enabling_retract(state);
13466         ASSERT(state->dts_nretained == 0);

13468         if (state->dts_activity == DTRACE_ACTIVITY_ACTIVE ||
13469             state->dts_activity == DTRACE_ACTIVITY_DRAINING) {
13470                 /*
13471                  * We have managed to come into dtrace_state_destroy() on a
13472                  * hot enabling -- almost certainly because of a disorderly
13473                  * shutdown of a consumer.  (That is, a consumer that is
13474                  * exiting without having called dtrace_stop().) In this case,
13475                  * we're going to set our activity to be KILLED, and then
13476                  * issue a sync to be sure that everyone is out of probe
13477                  * context before we start blowing away ECBs.
13478                  */
13479                 state->dts_activity = DTRACE_ACTIVITY_KILLED;
13480                 dtrace_sync();
13481         }

13483         /*
13484          * Release the credential hold we took in dtrace_state_create().
13485          */
13486         if (state->dts_cred.dcr_cred != NULL)
13487                 crfree(state->dts_cred.dcr_cred);

13489         /*
13490          * Now we can safely disable and destroy any enabled probes.  Because
13491          * any DTRACE_PRIV_KERNEL probes may actually be slowing our progress
13492          * (especially if they're all enabled), we take two passes through the
13493          * ECBs:  in the first, we disable just DTRACE_PRIV_KERNEL probes, and
13494          * in the second we disable whatever is left over.
13495          */
13496         for (match = DTRACE_PRIV_KERNEL; ; match = 0) {
13497                 for (i = 0; i < state->dts_necbs; i++) {
13498                         if ((ecb = state->dts_ecbs[i]) == NULL)
13499                                 continue;

13501                         if (match && ecb->dte_probe != NULL) {
13502                                 dtrace_probe_t *probe = ecb->dte_probe;
13503                                 dtrace_provider_t *prov = probe->dtpr_provider;

13505                                 if (!(prov->dtpv_priv.dtpp_flags & match))
13506                                         continue;
13507                         }

13509                         dtrace_ecb_disable(ecb);
13510                         dtrace_ecb_destroy(ecb);
13511                 }

13513                 if (!match)
13514                         break;
13515         }

13517         /*
13518          * Before we free the buffers, perform one more sync to assure that
13519          * every CPU is out of probe context.
13520          */
13521         dtrace_sync();

13523         dtrace_buffer_free(state->dts_buffer);
13524         dtrace_buffer_free(state->dts_aggbuffer);
```

```
13526         for (i = 0; i < nspec; i++)
13527                 dtrace_buffer_free(spec[i].dtsp_buffer);

13529         if (state->dts_cleaner != CYCLIC_NONE)
13530                 cyclic_remove(state->dts_cleaner);

13532         if (state->dts_deadman != CYCLIC_NONE)
13533                 cyclic_remove(state->dts_deadman);

13535         dtrace_dstate_fini(&vstate->dtvs_dynvars);
13536         dtrace_vstate_fini(vstate);
13537         kmem_free(state->dts_ecbs, state->dts_necbs * sizeof (dtrace_ecb_t *));

13539         if (state->dts_aggregations != NULL) {
13540 #ifdef DEBUG
13541                 for (i = 0; i < state->dts_naggregations; i++)
13542                         ASSERT(state->dts_aggregations[i] == NULL);
13543 #endif
13544                 ASSERT(state->dts_naggregations > 0);
13545                 kmem_free(state->dts_aggregations,
13546                     state->dts_naggregations * sizeof (dtrace_aggregation_t *));
13547         }

13549         kmem_free(state->dts_buffer, bufsize);
13550         kmem_free(state->dts_aggbuffer, bufsize);

13552         for (i = 0; i < nspec; i++)
13553                 kmem_free(spec[i].dtsp_buffer, bufsize);

13555         kmem_free(spec, nspec * sizeof (dtrace_speculation_t));

13557         dtrace_format_destroy(state);

13559         vmem_destroy(state->dts_aggid_arena);
13560         ddi_soft_state_free(dtrace_softstate, minor);
13561         vmem_free(dtrace_minor, (void *)(uintptr_t)minor, 1);
13562 }

13564 /*
13565  * DTrace Anonymous Enabling Functions
13566  */
13567 static dtrace_state_t *
13568 dtrace_anon_grab(void)
13569 {
13570         dtrace_state_t *state;

13572         ASSERT(MUTEX_HELD(&dtrace_lock));

13574         if ((state = dtrace_anon.dta_state) == NULL) {
13575                 ASSERT(dtrace_anon.dta_enabling == NULL);
13576                 return (NULL);
13577         }

13579         ASSERT(dtrace_anon.dta_enabling != NULL);
13580         ASSERT(dtrace_retained != NULL);

13582         dtrace_enabling_destroy(dtrace_anon.dta_enabling);
13583         dtrace_anon.dta_enabling = NULL;
13584         dtrace_anon.dta_state = NULL;

13586         return (state);
13587 }

13589 static void
13590 dtrace_anon_property(void)
13591 {
```



```
13592         int i, rv;
13593         dtrace_state_t *state;
13594         dof_hdr_t *dof;
13595         char c[32];             /* enough for "dof-data-" + digits */

13597         ASSERT(MUTEX_HELD(&dtrace_lock));
13598         ASSERT(MUTEX_HELD(&cpu_lock));

13600         for (i = 0; ; i++) {
13601                 (void) snprintf(c, sizeof (c), "dof-data-%d", i);

13603                 dtrace_err_verbose = 1;

13605                 if ((dof = dtrace_dof_property(c)) == NULL) {
13606                         dtrace_err_verbose = 0;
13607                         break;
13608                 }

13610                 /*
13611                  * We want to create anonymous state, so we need to transition
13612                  * the kernel debugger to indicate that DTrace is active.  If
13613                  * this fails (e.g. because the debugger has modified text in
13614                  * some way), we won't continue with the processing.
13615                  */
13616                 if (kdi_dtrace_set(KDI_DTSET_DTRACE_ACTIVATE) != 0) {
13617                         cmn_err(CE_NOTE, "kernel debugger active; anonymous "
13618                             "enabling ignored.");
13619                         dtrace_dof_destroy(dof);
13620                         break;
13621                 }

13623                 /*
13624                  * If we haven't allocated an anonymous state, we'll do so now.
13625                  */
13626                 if ((state = dtrace_anon.dta_state) == NULL) {
13627                         state = dtrace_state_create(NULL, NULL);
13628                         dtrace_anon.dta_state = state;

13630                         if (state == NULL) {
13631                                 /*
13632                                  * This basically shouldn't happen:  the only
13633                                  * failure mode from dtrace_state_create() is a
13634                                  * failure of ddi_soft_state_zalloc() that
13635                                  * itself should never happen.  Still, the
13636                                  * interface allows for a failure mode, and
13637                                  * we want to fail as gracefully as possible:
13638                                  * we'll emit an error message and cease
13639                                  * processing anonymous state in this case.
13640                                  */
13641                                 cmn_err(CE_WARN, "failed to create "
13642                                     "anonymous state");
13643                                 dtrace_dof_destroy(dof);
13644                                 break;
13645                         }
13646                 }

13648                 rv = dtrace_dof_slurp(dof, &state->dts_vstate, CRED(),
13649                     &dtrace_anon.dta_enabling, 0, B_TRUE);

13651                 if (rv == 0)
13652                         rv = dtrace_dof_options(dof, state);

13654                 dtrace_err_verbose = 0;
13655                 dtrace_dof_destroy(dof);

13657                 if (rv != 0) {
```

```
13658                         /*
13659                          * This is malformed DOF; chuck any anonymous state
13660                          * that we created.
13661                          */
13662                         ASSERT(dtrace_anon.dta_enabling == NULL);
13663                         dtrace_state_destroy(state);
13664                         dtrace_anon.dta_state = NULL;
13665                         break;
13666                 }

13668                 ASSERT(dtrace_anon.dta_enabling != NULL);
13669         }

13671         if (dtrace_anon.dta_enabling != NULL) {
13672                 int rval;

13674                 /*
13675                  * dtrace_enabling_retain() can only fail because we are
13676                  * trying to retain more enablings than are allowed -- but
13677                  * we only have one anonymous enabling, and we are guaranteed
13678                  * to be allowed at least one retained enabling; we assert
13679                  * that dtrace_enabling_retain() returns success.
13680                  */
13681                 rval = dtrace_enabling_retain(dtrace_anon.dta_enabling);
13682                 ASSERT(rval == 0);

13684                 dtrace_enabling_dump(dtrace_anon.dta_enabling);
13685         }
13686 }

13688 /*
13689  * DTrace Helper Functions
13690  */
13691 static void
13692 dtrace_helper_trace(dtrace_helper_action_t *helper,
13693     dtrace_mstate_t *mstate, dtrace_vstate_t *vstate, int where)
13694 {
13695         uint32_t size, next, nnext, i;
13696         dtrace_helptrace_t *ent;
13697         uint16_t flags = cpu_core[CPU->cpu_id].cpuc_dtrace_flags;

13699         if (!dtrace_helptrace_enabled)
13700                 return;

13702         ASSERT(vstate->dtvs_nlocals <= dtrace_helptrace_nlocals);

13704         /*
13705          * What would a tracing framework be without its own tracing
13706          * framework?  (Well, a hell of a lot simpler, for starters...)
13707          */
13708         size = sizeof (dtrace_helptrace_t) + dtrace_helptrace_nlocals *
13709             sizeof (uint64_t) - sizeof (uint64_t);

13711         /*
13712          * Iterate until we can allocate a slot in the trace buffer.
13713          */
13714         do {
13715                 next = dtrace_helptrace_next;

13717                 if (next + size < dtrace_helptrace_bufsize) {
13718                         nnext = next + size;
13719                 } else {
13720                         nnext = size;
13721                 }
13722         } while (dtrace_cas32(&dtrace_helptrace_next, next, nnext) != next);
```



```
13724         /*
13725          * We have our slot; fill it in.
13726          */
13727         if (nnext == size)
13728                 next = 0;

13730         ent = (dtrace_helptrace_t *)&dtrace_helptrace_buffer[next];
13731         ent->dtht_helper = helper;
13732         ent->dtht_where = where;
13733         ent->dtht_nlocals = vstate->dtvs_nlocals;

13735         ent->dtht_fltoffs = (mstate->dtms_present & DTRACE_MSTATE_FLTOFFS) ?
13736             mstate->dtms_fltoffs : -1;
13737         ent->dtht_fault = DTRACE_FLAGS2FLT(flags);
13738         ent->dtht_illval = cpu_core[CPU->cpu_id].cpuc_dtrace_illval;

13740         for (i = 0; i < vstate->dtvs_nlocals; i++) {
13741                 dtrace_statvar_t *svar;

13743                 if ((svar = vstate->dtvs_locals[i]) == NULL)
13744                         continue;

13746                 ASSERT(svar->dtsv_size >= NCPU * sizeof (uint64_t));
13747                 ent->dtht_locals[i] =
13748                     ((uint64_t *)(uintptr_t)svar->dtsv_data)[CPU->cpu_id];
13749         }
13750 }

13752 static uint64_t
13753 dtrace_helper(int which, dtrace_mstate_t *mstate,
13754     dtrace_state_t *state, uint64_t arg0, uint64_t arg1)
13755 {
13756         uint16_t *flags = &cpu_core[CPU->cpu_id].cpuc_dtrace_flags;
13757         uint64_t sarg0 = mstate->dtms_arg[0];
13758         uint64_t sarg1 = mstate->dtms_arg[1];
13759         uint64_t rval;
13760         dtrace_helpers_t *helpers = curproc->p_dtrace_helpers;
13761         dtrace_helper_action_t *helper;
13762         dtrace_vstate_t *vstate;
13763         dtrace_difo_t *pred;
13764         int i, trace = dtrace_helptrace_enabled;

13766         ASSERT(which >= 0 && which < DTRACE_NHELPER_ACTIONS);

13768         if (helpers == NULL)
13769                 return (0);

13771         if ((helper = helpers->dthps_actions[which]) == NULL)
13772                 return (0);

13774         vstate = &helpers->dthps_vstate;
13775         mstate->dtms_arg[0] = arg0;
13776         mstate->dtms_arg[1] = arg1;

13778         /*
13779          * Now iterate over each helper.  If its predicate evaluates to 'true',
13780          * we'll call the corresponding actions.  Note that the below calls
13781          * to dtrace_dif_emulate() may set faults in machine state.  This is
13782          * okay:  our caller (the outer dtrace_dif_emulate()) will simply plow
13783          * the stored DIF offset with its own (which is the desired behavior).
13784          * Also, note the calls to dtrace_dif_emulate() may allocate scratch
13785          * from machine state; this is okay, too.
13786          */
13787         for (; helper != NULL; helper = helper->dtha_next) {
13788                 if ((pred = helper->dtha_predicate) != NULL) {
13789                         if (trace)
```

```
13790                                   dtrace_helper_trace(helper, mstate, vstate, 0);

13792                           if (!dtrace_dif_emulate(pred, mstate, vstate, state))
13793                                   goto next;

13795                           if (*flags & CPU_DTRACE_FAULT)
13796                                   goto err;
13797                   }

13799                   for (i = 0; i < helper->dtha_nactions; i++) {
13800                           if (trace)
13801                                   dtrace_helper_trace(helper,
13802                                       mstate, vstate, i + 1);

13804                           rval = dtrace_dif_emulate(helper->dtha_actions[i],
13805                               mstate, vstate, state);

13807                           if (*flags & CPU_DTRACE_FAULT)
13808                                   goto err;
13809                   }

13811 next:
13812                   if (trace)
13813                           dtrace_helper_trace(helper, mstate, vstate,
13814                               DTRACE_HELPTRACE_NEXT);
13815           }

13817           if (trace)
13818                   dtrace_helper_trace(helper, mstate, vstate,
13819                       DTRACE_HELPTRACE_DONE);

13821           /*
13822            * Restore the arg0 that we saved upon entry.
13823            */
13824           mstate->dtms_arg[0] = sarg0;
13825           mstate->dtms_arg[1] = sarg1;

13827           return (rval);

13829 err:
13830           if (trace)
13831                   dtrace_helper_trace(helper, mstate, vstate,
13832                       DTRACE_HELPTRACE_ERR);

13834           /*
13835            * Restore the arg0 that we saved upon entry.
13836            */
13837           mstate->dtms_arg[0] = sarg0;
13838           mstate->dtms_arg[1] = sarg1;

13840           return (NULL);
13841 }

13843 static void
13844 dtrace_helper_action_destroy(dtrace_helper_action_t *helper,
13845     dtrace_vstate_t *vstate)
13846 {
13847           int i;

13849           if (helper->dtha_predicate != NULL)
13850                   dtrace_difo_release(helper->dtha_predicate, vstate);

13852           for (i = 0; i < helper->dtha_nactions; i++) {
13853                   ASSERT(helper->dtha_actions[i] != NULL);
13854                   dtrace_difo_release(helper->dtha_actions[i], vstate);
13855           }
```



```
13857           kmem_free(helper->dtha_actions,
13858               helper->dtha_nactions * sizeof (dtrace_difo_t *));
13859           kmem_free(helper, sizeof (dtrace_helper_action_t));
13860 }

13862 static int
13863 dtrace_helper_destroygen(int gen)
13864 {
13865           proc_t *p = curproc;
13866           dtrace_helpers_t *help = p->p_dtrace_helpers;
13867           dtrace_vstate_t *vstate;
13868           int i;

13870           ASSERT(MUTEX_HELD(&dtrace_lock));

13872           if (help == NULL || gen > help->dthps_generation)
13873                   return (EINVAL);

13875           vstate = &help->dthps_vstate;

13877           for (i = 0; i < DTRACE_NHELPER_ACTIONS; i++) {
13878                   dtrace_helper_action_t *last = NULL, *h, *next;

13880                   for (h = help->dthps_actions[i]; h != NULL; h = next) {
13881                           next = h->dtha_next;

13883                           if (h->dtha_generation == gen) {
13884                                   if (last != NULL) {
13885                                           last->dtha_next = next;
13886                                   } else {
13887                                           help->dthps_actions[i] = next;
13888                                   }

13890                                   dtrace_helper_action_destroy(h, vstate);
13891                           } else {
13892                                   last = h;
13893                           }
13894                   }
13895           }

13897           /*
13898            * Interate until we've cleared out all helper providers with the
13899            * given generation number.
13900            */
13901           for (;;) {
13902                   dtrace_helper_provider_t *prov;

13904                   /*
13905                    * Look for a helper provider with the right generation. We
13906                    * have to start back at the beginning of the list each time
13907                    * because we drop dtrace_lock. It's unlikely that we'll make
13908                    * more than two passes.
13909                    */
13910                   for (i = 0; i < help->dthps_nprovs; i++) {
13911                           prov = help->dthps_provs[i];

13913                           if (prov->dthp_generation == gen)
13914                                   break;
13915                   }

13917                   /*
13918                    * If there were no matches, we're done.
13919                    */
13920                   if (i == help->dthps_nprovs)
13921                           break;
```

```
13923                 /*
13924                  * Move the last helper provider into this slot.
13925                  */
13926                 help->dthps_nprovs--;
13927                 help->dthps_provs[i] = help->dthps_provs[help->dthps_nprovs];
13928                 help->dthps_provs[help->dthps_nprovs] = NULL;

13930                 mutex_exit(&dtrace_lock);

13932                 /*
13933                  * If we have a meta provider, remove this helper provider.
13934                  */
13935                 mutex_enter(&dtrace_meta_lock);
13936                 if (dtrace_meta_pid != NULL) {
13937                         ASSERT(dtrace_deferred_pid == NULL);
13938                         dtrace_helper_provider_remove(&prov->dthp_prov,
13939                             p->p_pid);
13940                 }
13941                 mutex_exit(&dtrace_meta_lock);

13943                 dtrace_helper_provider_destroy(prov);

13945                 mutex_enter(&dtrace_lock);
13946         }

13948         return (0);
13949 }

13951 static int
13952 dtrace_helper_validate(dtrace_helper_action_t *helper)
13953 {
13954         int err = 0, i;
13955         dtrace_difo_t *dp;

13957         if ((dp = helper->dtha_predicate) != NULL)
13958                 err += dtrace_difo_validate_helper(dp);

13960         for (i = 0; i < helper->dtha_nactions; i++)
13961                 err += dtrace_difo_validate_helper(helper->dtha_actions[i]);

13963         return (err == 0);
13964 }

13966 static int
13967 dtrace_helper_action_add(int which, dtrace_ecbdesc_t *ep)
13968 {
13969         dtrace_helpers_t *help;
13970         dtrace_helper_action_t *helper, *last;
13971         dtrace_actdesc_t *act;
13972         dtrace_vstate_t *vstate;
13973         dtrace_predicate_t *pred;
13974         int count = 0, nactions = 0, i;

13976         if (which < 0 || which >= DTRACE_NHELPER_ACTIONS)
13977                 return (EINVAL);

13979         help = curproc->p_dtrace_helpers;
13980         last = help->dthps_actions[which];
13981         vstate = &help->dthps_vstate;

13983         for (count = 0; last != NULL; last = last->dtha_next) {
13984                 count++;
13985                 if (last->dtha_next == NULL)
13986                         break;
13987         }
```



```
13989         /*
13990          * If we already have dtrace_helper_actions_max helper actions for this
13991          * helper action type, we'll refuse to add a new one.
13992          */
13993         if (count >= dtrace_helper_actions_max)
13994                 return (ENOSPC);

13996         helper = kmem_zalloc(sizeof (dtrace_helper_action_t), KM_SLEEP);
13997         helper->dtha_generation = help->dthps_generation;

13999         if ((pred = ep->dted_pred.dtpdd_predicate) != NULL) {
14000                 ASSERT(pred->dtp_difo != NULL);
14001                 dtrace_difo_hold(pred->dtp_difo);
14002                 helper->dtha_predicate = pred->dtp_difo;
14003         }

14005         for (act = ep->dted_action; act != NULL; act = act->dtad_next) {
14006                 if (act->dtad_kind != DTRACEACT_DIFEXPR)
14007                         goto err;

14009                 if (act->dtad_difo == NULL)
14010                         goto err;

14012                 nactions++;
14013         }

14015         helper->dtha_actions = kmem_zalloc(sizeof (dtrace_difo_t *) *
14016             (helper->dtha_nactions = nactions), KM_SLEEP);

14018         for (act = ep->dted_action, i = 0; act != NULL; act = act->dtad_next) {
14019                 dtrace_difo_hold(act->dtad_difo);
14020                 helper->dtha_actions[i++] = act->dtad_difo;
14021         }

14023         if (!dtrace_helper_validate(helper))
14024                 goto err;

14026         if (last == NULL) {
14027                 help->dthps_actions[which] = helper;
14028         } else {
14029                 last->dtha_next = helper;
14030         }

14032         if (vstate->dtvs_nlocals > dtrace_helptrace_nlocals) {
14033                 dtrace_helptrace_nlocals = vstate->dtvs_nlocals;
14034                 dtrace_helptrace_next = 0;
14035         }

14037         return (0);
14038 err:
14039         dtrace_helper_action_destroy(helper, vstate);
14040         return (EINVAL);
14041 }

14043 static void
14044 dtrace_helper_provider_register(proc_t *p, dtrace_helpers_t *help,
14045     dof_helper_t *dofhp)
14046 {
14047         ASSERT(MUTEX_NOT_HELD(&dtrace_lock));

14049         mutex_enter(&dtrace_meta_lock);
14050         mutex_enter(&dtrace_lock);

14052         if (!dtrace_attached() || dtrace_meta_pid == NULL) {
14053                 /*
```

```
14054                  * If the dtrace module is loaded but not attached, or if
14055                  * there aren't isn't a meta provider registered to deal with
14056                  * these provider descriptions, we need to postpone creating
14057                  * the actual providers until later.
14058                  */
14060                 if (help->dthps_next == NULL && help->dthps_prev == NULL &&
14061                     dtrace_deferred_pid != help) {
14062                         help->dthps_deferred = 1;
14063                         help->dthps_pid = p->p_pid;
14064                         help->dthps_next = dtrace_deferred_pid;
14065                         help->dthps_prev = NULL;
14066                         if (dtrace_deferred_pid != NULL)
14067                                 dtrace_deferred_pid->dthps_prev = help;
14068                         dtrace_deferred_pid = help;
14069                 }
14071                 mutex_exit(&dtrace_lock);
14073         } else if (dofhp != NULL) {
14074                 /*
14075                  * If the dtrace module is loaded and we have a particular
14076                  * helper provider description, pass that off to the
14077                  * meta provider.
14078                  */
14080                 mutex_exit(&dtrace_lock);
14082                 dtrace_helper_provide(dofhp, p->p_pid);
14084         } else {
14085                 /*
14086                  * Otherwise, just pass all the helper provider descriptions
14087                  * off to the meta provider.
14088                  */
14090                 int i;
14091                 mutex_exit(&dtrace_lock);
14093                 for (i = 0; i < help->dthps_nprovs; i++) {
14094                         dtrace_helper_provide(&help->dthps_provs[i]->dthp_prov,
14095                             p->p_pid);
14096                 }
14097         }
14099         mutex_exit(&dtrace_meta_lock);
14100 }
14102 static int
14103 dtrace_helper_provider_add(dof_helper_t *dofhp, int gen)
14104 {
14105         dtrace_helpers_t *help;
14106         dtrace_helper_provider_t *hprov, **tmp_provs;
14107         uint_t tmp_maxprovs, i;
14109         ASSERT(MUTEX_HELD(&dtrace_lock));
14111         help = curproc->p_dtrace_helpers;
14112         ASSERT(help != NULL);
14114         /*
14115          * If we already have dtrace_helper_providers_max helper providers,
14116          * we're refuse to add a new one.
14117          */
14118         if (help->dthps_nprovs >= dtrace_helper_providers_max)
14119                 return (ENOSPC);
```



```
14121         /*
14122          * Check to make sure this isn't a duplicate.
14123          */
14124         for (i = 0; i < help->dthps_nprovs; i++) {
14125                 if (dofhp->dofhp_dof ==
14126                     help->dthps_provs[i]->dthp_prov.dofhp_dof)
14127                         return (EALREADY);
14128         }
14130         hprov = kmem_zalloc(sizeof (dtrace_helper_provider_t), KM_SLEEP);
14131         hprov->dthp_prov = *dofhp;
14132         hprov->dthp_ref = 1;
14133         hprov->dthp_generation = gen;
14135         /*
14136          * Allocate a bigger table for helper providers if it's already full.
14137          */
14138         if (help->dthps_maxprovs == help->dthps_nprovs) {
14139                 tmp_maxprovs = help->dthps_maxprovs;
14140                 tmp_provs = help->dthps_provs;
14142                 if (help->dthps_maxprovs == 0)
14143                         help->dthps_maxprovs = 2;
14144                 else
14145                         help->dthps_maxprovs *= 2;
14146                 if (help->dthps_maxprovs > dtrace_helper_providers_max)
14147                         help->dthps_maxprovs = dtrace_helper_providers_max;
14149                 ASSERT(tmp_maxprovs < help->dthps_maxprovs);
14151                 help->dthps_provs = kmem_zalloc(help->dthps_maxprovs *
14152                     sizeof (dtrace_helper_provider_t *), KM_SLEEP);
14154                 if (tmp_provs != NULL) {
14155                         bcopy(tmp_provs, help->dthps_provs, tmp_maxprovs *
14156                             sizeof (dtrace_helper_provider_t *));
14157                         kmem_free(tmp_provs, tmp_maxprovs *
14158                             sizeof (dtrace_helper_provider_t *));
14159                 }
14160         }
14162         help->dthps_provs[help->dthps_nprovs] = hprov;
14163         help->dthps_nprovs++;
14165         return (0);
14166 }
14168 static void
14169 dtrace_helper_provider_destroy(dtrace_helper_provider_t *hprov)
14170 {
14171         mutex_enter(&dtrace_lock);
14173         if (--hprov->dthp_ref == 0) {
14174                 dof_hdr_t *dof;
14175                 mutex_exit(&dtrace_lock);
14176                 dof = (dof_hdr_t *)(uintptr_t)hprov->dthp_prov.dofhp_dof;
14177                 dtrace_dof_destroy(dof);
14178                 kmem_free(hprov, sizeof (dtrace_helper_provider_t));
14179         } else {
14180                 mutex_exit(&dtrace_lock);
14181         }
14182 }
14184 static int
14185 dtrace_helper_provider_validate(dof_hdr_t *dof, dof_sec_t *sec)
```

```
14186 {
14187         uintptr_t daddr = (uintptr_t)dof;
14188         dof_sec_t *str_sec, *prb_sec, *arg_sec, *off_sec, *enoff_sec;
14189         dof_provider_t *provider;
14190         dof_probe_t *probe;
14191         uint8_t *arg;
14192         char *strtab, *typestr;
14193         dof_stridx_t typeidx;
14194         size_t typesz;
14195         uint_t nprobes, j, k;

14197         ASSERT(sec->dofs_type == DOF_SECT_PROVIDER);

14199         if (sec->dofs_offset & (sizeof (uint_t) - 1)) {
14200                 dtrace_dof_error(dof, "misaligned section offset");
14201                 return (-1);
14202         }

14204         /*
14205          * The section needs to be large enough to contain the DOF provider
14206          * structure appropriate for the given version.
14207          */
14208         if (sec->dofs_size <
14209             ((dof->dofh_ident[DOF_ID_VERSION] == DOF_VERSION_1) ?
14210             offsetof(dof_provider_t, dofpv_prenoffs) :
14211             sizeof (dof_provider_t))) {
14212                 dtrace_dof_error(dof, "provider section too small");
14213                 return (-1);
14214         }

14216         provider = (dof_provider_t *)(uintptr_t)(daddr + sec->dofs_offset);
14217         str_sec = dtrace_dof_sect(dof, DOF_SECT_STRTAB, provider->dofpv_strtab);
14218         prb_sec = dtrace_dof_sect(dof, DOF_SECT_PROBES, provider->dofpv_probes);
14219         arg_sec = dtrace_dof_sect(dof, DOF_SECT_PRARGS, provider->dofpv_prargs);
14220         off_sec = dtrace_dof_sect(dof, DOF_SECT_PROFFS, provider->dofpv_proffs);

14222         if (str_sec == NULL || prb_sec == NULL ||
14223             arg_sec == NULL || off_sec == NULL)
14224                 return (-1);

14226         enoff_sec = NULL;

14228         if (dof->dofh_ident[DOF_ID_VERSION] != DOF_VERSION_1 &&
14229             provider->dofpv_prenoffs != DOF_SECT_NONE &&
14230             (enoff_sec = dtrace_dof_sect(dof, DOF_SECT_PRENOFFS,
14231             provider->dofpv_prenoffs)) == NULL)
14232                 return (-1);

14234         strtab = (char *)(uintptr_t)(daddr + str_sec->dofs_offset);

14236         if (provider->dofpv_name >= str_sec->dofs_size ||
14237             strlen(strtab + provider->dofpv_name) >= DTRACE_PROVNAMELEN) {
14238                 dtrace_dof_error(dof, "invalid provider name");
14239                 return (-1);
14240         }

14242         if (prb_sec->dofs_entsize == 0 ||
14243             prb_sec->dofs_entsize > prb_sec->dofs_size) {
14244                 dtrace_dof_error(dof, "invalid entry size");
14245                 return (-1);
14246         }

14248         if (prb_sec->dofs_entsize & (sizeof (uintptr_t) - 1)) {
14249                 dtrace_dof_error(dof, "misaligned entry size");
14250                 return (-1);
14251         }
```



```
14253         if (off_sec->dofs_entsize != sizeof (uint32_t)) {
14254                 dtrace_dof_error(dof, "invalid entry size");
14255                 return (-1);
14256         }

14258         if (off_sec->dofs_offset & (sizeof (uint32_t) - 1)) {
14259                 dtrace_dof_error(dof, "misaligned section offset");
14260                 return (-1);
14261         }

14263         if (arg_sec->dofs_entsize != sizeof (uint8_t)) {
14264                 dtrace_dof_error(dof, "invalid entry size");
14265                 return (-1);
14266         }

14268         arg = (uint8_t *)(uintptr_t)(daddr + arg_sec->dofs_offset);

14270         nprobes = prb_sec->dofs_size / prb_sec->dofs_entsize;

14272         /*
14273          * Take a pass through the probes to check for errors.
14274          */
14275         for (j = 0; j < nprobes; j++) {
14276                 probe = (dof_probe_t *)(uintptr_t)(daddr +
14277                     prb_sec->dofs_offset + j * prb_sec->dofs_entsize);

14279                 if (probe->dofpr_func >= str_sec->dofs_size) {
14280                         dtrace_dof_error(dof, "invalid function name");
14281                         return (-1);
14282                 }

14284                 if (strlen(strtab + probe->dofpr_func) >= DTRACE_FUNCNAMELEN) {
14285                         dtrace_dof_error(dof, "function name too long");
14286                         return (-1);
14287                 }

14289                 if (probe->dofpr_name >= str_sec->dofs_size ||
14290                     strlen(strtab + probe->dofpr_name) >= DTRACE_NAMELEN) {
14291                         dtrace_dof_error(dof, "invalid probe name");
14292                         return (-1);
14293                 }

14295                 /*
14296                  * The offset count must not wrap the index, and the offsets
14297                  * must also not overflow the section's data.
14298                  */
14299                 if (probe->dofpr_offidx + probe->dofpr_noffs <
14300                     probe->dofpr_offidx ||
14301                     (probe->dofpr_offidx + probe->dofpr_noffs) *
14302                     off_sec->dofs_entsize > off_sec->dofs_size) {
14303                         dtrace_dof_error(dof, "invalid probe offset");
14304                         return (-1);
14305                 }

14307                 if (dof->dofh_ident[DOF_ID_VERSION] != DOF_VERSION_1) {
14308                         /*
14309                          * If there's no is-enabled offset section, make sure
14310                          * there aren't any is-enabled offsets. Otherwise
14311                          * perform the same checks as for probe offsets
14312                          * (immediately above).
14313                          */
14314                         if (enoff_sec == NULL) {
14315                                 if (probe->dofpr_enoffidx != 0 ||
14316                                     probe->dofpr_nenoffs != 0) {
14317                                         dtrace_dof_error(dof, "is-enabled "
```

```
                                        "offsets with null section");
                                return (-1);
                        }
                } else if (probe->dofpr_enoffidx +
                    probe->dofpr_nenoffs < probe->dofpr_enoffidx ||
                    (probe->dofpr_enoffidx + probe->dofpr_nenoffs) *
                    enoff_sec->dofs_entsize > enoff_sec->dofs_size) {
                        dtrace_dof_error(dof, "invalid is-enabled "
                            "offset");
                        return (-1);
                }

                if (probe->dofpr_noffs + probe->dofpr_nenoffs == 0) {
                        dtrace_dof_error(dof, "zero probe and "
                            "is-enabled offsets");
                        return (-1);
                }
        } else if (probe->dofpr_noffs == 0) {
                dtrace_dof_error(dof, "zero probe offsets");
                return (-1);
        }

        if (probe->dofpr_argidx + probe->dofpr_xargc <
            probe->dofpr_argidx ||
            (probe->dofpr_argidx + probe->dofpr_xargc) *
            arg_sec->dofs_entsize > arg_sec->dofs_size) {
                dtrace_dof_error(dof, "invalid args");
                return (-1);
        }

        typeidx = probe->dofpr_nargv;
        typestr = strtab + probe->dofpr_nargv;
        for (k = 0; k < probe->dofpr_nargc; k++) {
                if (typeidx >= str_sec->dofs_size) {
                        dtrace_dof_error(dof, "bad "
                            "native argument type");
                        return (-1);
                }

                typesz = strlen(typestr) + 1;
                if (typesz > DTRACE_ARGTYPELEN) {
                        dtrace_dof_error(dof, "native "
                            "argument type too long");
                        return (-1);
                }
                typeidx += typesz;
                typestr += typesz;
        }

        typeidx = probe->dofpr_xargv;
        typestr = strtab + probe->dofpr_xargv;
        for (k = 0; k < probe->dofpr_xargc; k++) {
                if (arg[probe->dofpr_argidx + k] > probe->dofpr_nargc) {
                        dtrace_dof_error(dof, "bad "
                            "native argument index");
                        return (-1);
                }

                if (typeidx >= str_sec->dofs_size) {
                        dtrace_dof_error(dof, "bad "
                            "translated argument type");
                        return (-1);
                }

                typesz = strlen(typestr) + 1;
                if (typesz > DTRACE_ARGTYPELEN) {
```



```
                        dtrace_dof_error(dof, "translated argument "
                            "type too long");
                        return (-1);
                }

                typeidx += typesz;
                typestr += typesz;
        }
}

        return (0);
}

static int
dtrace_helper_slurp(dof_hdr_t *dof, dof_helper_t *dhp)
{
        dtrace_helpers_t *help;
        dtrace_vstate_t *vstate;
        dtrace_enabling_t *enab = NULL;
        int i, gen, rv, nhelpers = 0, nprovs = 0, destroy = 1;
        uintptr_t daddr = (uintptr_t)dof;

        ASSERT(MUTEX_HELD(&dtrace_lock));

        if ((help = curproc->p_dtrace_helpers) == NULL)
                help = dtrace_helpers_create(curproc);

        vstate = &help->dthps_vstate;

        if ((rv = dtrace_dof_slurp(dof, vstate, NULL, &enab,
            dhp != NULL ? dhp->dofhp_addr : 0, B_FALSE)) != 0) {
                dtrace_dof_destroy(dof);
                return (rv);
        }

        /*
         * Look for helper providers and validate their descriptions.
         */
        if (dhp != NULL) {
                for (i = 0; i < dof->dofh_secnum; i++) {
                        dof_sec_t *sec = (dof_sec_t *)(uintptr_t)(daddr +
                            dof->dofh_secoff + i * dof->dofh_secsize);

                        if (sec->dofs_type != DOF_SECT_PROVIDER)
                                continue;

                        if (dtrace_helper_provider_validate(dof, sec) != 0) {
                                dtrace_enabling_destroy(enab);
                                dtrace_dof_destroy(dof);
                                return (-1);
                        }

                        nprovs++;
                }
        }

        /*
         * Now we need to walk through the ECB descriptions in the enabling.
         */
        for (i = 0; i < enab->dten_ndesc; i++) {
                dtrace_ecbdesc_t *ep = enab->dten_desc[i];
                dtrace_probedesc_t *desc = &ep->dted_probe;

                if (strcmp(desc->dtpd_provider, "dtrace") != 0)
                        continue;
```

```
14450                 if (strcmp(desc->dtpd_mod, "helper") != 0)
14451                         continue;

14453                 if (strcmp(desc->dtpd_func, "ustack") != 0)
14454                         continue;

14456                 if ((rv = dtrace_helper_action_add(DTRACE_HELPER_ACTION_USTACK,
14457                     ep)) != 0) {
14458                         /*
14459                          * Adding this helper action failed -- we are now going
14460                          * to rip out the entire generation and return failure.
14461                          */
14462                         (void) dtrace_helper_destroygen(help->dthps_generation);
14463                         dtrace_enabling_destroy(enab);
14464                         dtrace_dof_destroy(dof);
14465                         return (-1);
14466                 }

14468                 nhelpers++;
14469         }

14471         if (nhelpers < enab->dten_ndesc)
14472                 dtrace_dof_error(dof, "unmatched helpers");

14474         gen = help->dthps_generation++;
14475         dtrace_enabling_destroy(enab);

14477         if (dhp != NULL && nprovs > 0) {
14478                 dhp->dofhp_dof = (uint64_t)(uintptr_t)dof;
14479                 if (dtrace_helper_provider_add(dhp, gen) == 0) {
14480                         mutex_exit(&dtrace_lock);
14481                         dtrace_helper_provider_register(curproc, help, dhp);
14482                         mutex_enter(&dtrace_lock);

14484                         destroy = 0;
14485                 }
14486         }

14488         if (destroy)
14489                 dtrace_dof_destroy(dof);

14491         return (gen);
14492 }

14494 static dtrace_helpers_t *
14495 dtrace_helpers_create(proc_t *p)
14496 {
14497         dtrace_helpers_t *help;

14499         ASSERT(MUTEX_HELD(&dtrace_lock));
14500         ASSERT(p->p_dtrace_helpers == NULL);

14502         help = kmem_zalloc(sizeof (dtrace_helpers_t), KM_SLEEP);
14503         help->dthps_actions = kmem_zalloc(sizeof (dtrace_helper_action_t *) *
14504             DTRACE_NHELPER_ACTIONS, KM_SLEEP);

14506         p->p_dtrace_helpers = help;
14507         dtrace_helpers++;

14509         return (help);
14510 }

14512 static void
14513 dtrace_helpers_destroy(void)
14514 {
14515         dtrace_helpers_t *help;
```



```
14516         dtrace_vstate_t *vstate;
14517         proc_t *p = curproc;
14518         int i;

14520         mutex_enter(&dtrace_lock);

14522         ASSERT(p->p_dtrace_helpers != NULL);
14523         ASSERT(dtrace_helpers > 0);

14525         help = p->p_dtrace_helpers;
14526         vstate = &help->dthps_vstate;

14528         /*
14529          * We're now going to lose the help from this process.
14530          */
14531         p->p_dtrace_helpers = NULL;
14532         dtrace_sync();

14534         /*
14535          * Destory the helper actions.
14536          */
14537         for (i = 0; i < DTRACE_NHELPER_ACTIONS; i++) {
14538                 dtrace_helper_action_t *h, *next;

14540                 for (h = help->dthps_actions[i]; h != NULL; h = next) {
14541                         next = h->dtha_next;
14542                         dtrace_helper_action_destroy(h, vstate);
14543                         h = next;
14544                 }
14545         }

14547         mutex_exit(&dtrace_lock);

14549         /*
14550          * Destroy the helper providers.
14551          */
14552         if (help->dthps_maxprovs > 0) {
14553                 mutex_enter(&dtrace_meta_lock);
14554                 if (dtrace_meta_pid != NULL) {
14555                         ASSERT(dtrace_deferred_pid == NULL);

14557                         for (i = 0; i < help->dthps_nprovs; i++) {
14558                                 dtrace_helper_provider_remove(
14559                                     &help->dthps_provs[i]->dthp_prov, p->p_pid);
14560                         }
14561                 } else {
14562                         mutex_enter(&dtrace_lock);
14563                         ASSERT(help->dthps_deferred == 0 ||
14564                             help->dthps_next != NULL ||
14565                             help->dthps_prev != NULL ||
14566                             help == dtrace_deferred_pid);

14568                         /*
14569                          * Remove the helper from the deferred list.
14570                          */
14571                         if (help->dthps_next != NULL)
14572                                 help->dthps_next->dthps_prev = help->dthps_prev;
14573                         if (help->dthps_prev != NULL)
14574                                 help->dthps_prev->dthps_next = help->dthps_next;
14575                         if (dtrace_deferred_pid == help) {
14576                                 dtrace_deferred_pid = help->dthps_next;
14577                                 ASSERT(help->dthps_prev == NULL);
14578                         }

14580                         mutex_exit(&dtrace_lock);
14581                 }
```

```
14583                 mutex_exit(&dtrace_meta_lock);

14585                 for (i = 0; i < help->dthps_nprovs; i++) {
14586                         dtrace_helper_provider_destroy(help->dthps_provs[i]);
14587                 }

14589                 kmem_free(help->dthps_provs, help->dthps_maxprovs *
14590                     sizeof (dtrace_helper_provider_t *));
14591         }

14593         mutex_enter(&dtrace_lock);

14595         dtrace_vstate_fini(&help->dthps_vstate);
14596         kmem_free(help->dthps_actions,
14597             sizeof (dtrace_helper_action_t *) * DTRACE_NHELPER_ACTIONS);
14598         kmem_free(help, sizeof (dtrace_helpers_t));

14600         --dtrace_helpers;
14601         mutex_exit(&dtrace_lock);
14602 }

14604 static void
14605 dtrace_helpers_duplicate(proc_t *from, proc_t *to)
14606 {
14607         dtrace_helpers_t *help, *newhelp;
14608         dtrace_helper_action_t *helper, *new, *last;
14609         dtrace_difo_t *dp;
14610         dtrace_vstate_t *vstate;
14611         int i, j, sz, hasprovs = 0;

14613         mutex_enter(&dtrace_lock);
14614         ASSERT(from->p_dtrace_helpers != NULL);
14615         ASSERT(dtrace_helpers > 0);

14617         help = from->p_dtrace_helpers;
14618         newhelp = dtrace_helpers_create(to);
14619         ASSERT(to->p_dtrace_helpers != NULL);

14621         newhelp->dthps_generation = help->dthps_generation;
14622         vstate = &newhelp->dthps_vstate;

14624         /*
14625          * Duplicate the helper actions.
14626          */
14627         for (i = 0; i < DTRACE_NHELPER_ACTIONS; i++) {
14628                 if ((helper = help->dthps_actions[i]) == NULL)
14629                         continue;

14631                 for (last = NULL; helper != NULL; helper = helper->dtha_next) {
14632                         new = kmem_zalloc(sizeof (dtrace_helper_action_t),
14633                             KM_SLEEP);
14634                         new->dtha_generation = helper->dtha_generation;

14636                         if ((dp = helper->dtha_predicate) != NULL) {
14637                                 dp = dtrace_difo_duplicate(dp, vstate);
14638                                 new->dtha_predicate = dp;
14639                         }

14641                         new->dtha_nactions = helper->dtha_nactions;
14642                         sz = sizeof (dtrace_difo_t *) * new->dtha_nactions;
14643                         new->dtha_actions = kmem_alloc(sz, KM_SLEEP);

14645                         for (j = 0; j < new->dtha_nactions; j++) {
14646                                 dtrace_difo_t *dp = helper->dtha_actions[j];
```



```
14648                                 ASSERT(dp != NULL);
14649                                 dp = dtrace_difo_duplicate(dp, vstate);
14650                                 new->dtha_actions[j] = dp;
14651                         }

14653                         if (last != NULL) {
14654                                 last->dtha_next = new;
14655                         } else {
14656                                 newhelp->dthps_actions[i] = new;
14657                         }

14659                         last = new;
14660                 }
14661         }

14663         /*
14664          * Duplicate the helper providers and register them with the
14665          * DTrace framework.
14666          */
14667         if (help->dthps_nprovs > 0) {
14668                 newhelp->dthps_nprovs = help->dthps_nprovs;
14669                 newhelp->dthps_maxprovs = help->dthps_nprovs;
14670                 newhelp->dthps_provs = kmem_alloc(newhelp->dthps_nprovs *
14671                     sizeof (dtrace_helper_provider_t *), KM_SLEEP);
14672                 for (i = 0; i < newhelp->dthps_nprovs; i++) {
14673                         newhelp->dthps_provs[i] = help->dthps_provs[i];
14674                         newhelp->dthps_provs[i]->dthp_ref++;
14675                 }

14677                 hasprovs = 1;
14678         }

14680         mutex_exit(&dtrace_lock);

14682         if (hasprovs)
14683                 dtrace_helper_provider_register(to, newhelp, NULL);
14684 }

14686 /*
14687  * DTrace Hook Functions
14688  */
14689 static void
14690 dtrace_module_loaded(struct modctl *ctl)
14691 {
14692         dtrace_provider_t *prv;

14694         mutex_enter(&dtrace_provider_lock);
14695         mutex_enter(&mod_lock);

14697         ASSERT(ctl->mod_busy);

14699         /*
14700          * We're going to call each providers per-module provide operation
14701          * specifying only this module.
14702          */
14703         for (prv = dtrace_provider; prv != NULL; prv = prv->dtpv_next)
14704                 prv->dtpv_pops.dtps_provide_module(prv->dtpv_arg, ctl);

14706         mutex_exit(&mod_lock);
14707         mutex_exit(&dtrace_provider_lock);

14709         /*
14710          * If we have any retained enablings, we need to match against them.
14711          * Enabling probes requires that cpu_lock be held, and we cannot hold
14712          * cpu_lock here -- it is legal for cpu_lock to be held when loading a
14713          * module.  (In particular, this happens when loading scheduling
```

```
14714          * classes.)  So if we have any retained enablings, we need to dispatch
14715          * our task queue to do the match for us.
14716          */
14717         mutex_enter(&dtrace_lock);

14719         if (dtrace_retained == NULL) {
14720                 mutex_exit(&dtrace_lock);
14721                 return;
14722         }

14724         (void) taskq_dispatch(dtrace_taskq,
14725             (task_func_t *)dtrace_enabling_matchall, NULL, TQ_SLEEP);

14727         mutex_exit(&dtrace_lock);

14729         /*
14730          * And now, for a little heuristic sleaze:  in general, we want to
14731          * match modules as soon as they load.  However, we cannot guarantee
14732          * this, because it would lead us to the lock ordering violation
14733          * outlined above.  The common case, of course, is that cpu_lock is
14734          * _not_ held -- so we delay here for a clock tick, hoping that that's
14735          * long enough for the task queue to do its work.  If it's not, it's
14736          * not a serious problem -- it just means that the module that we
14737          * just loaded may not be immediately instrumentable.
14738          */
14739         delay(1);
14740 }

14742 static void
14743 dtrace_module_unloaded(struct modctl *ctl)
14744 {
14745         dtrace_probe_t template, *probe, *first, *next;
14746         dtrace_provider_t *prov;

14748         template.dtpr_mod = ctl->mod_modname;

14750         mutex_enter(&dtrace_provider_lock);
14751         mutex_enter(&mod_lock);
14752         mutex_enter(&dtrace_lock);

14754         if (dtrace_bymod == NULL) {
14755                 /*
14756                  * The DTrace module is loaded (obviously) but not attached;
14757                  * we don't have any work to do.
14758                  */
14759                 mutex_exit(&dtrace_provider_lock);
14760                 mutex_exit(&mod_lock);
14761                 mutex_exit(&dtrace_lock);
14762                 return;
14763         }

14765         for (probe = first = dtrace_hash_lookup(dtrace_bymod, &template);
14766             probe != NULL; probe = probe->dtpr_nextmod) {
14767                 if (probe->dtpr_ecb != NULL) {
14768                         mutex_exit(&dtrace_provider_lock);
14769                         mutex_exit(&mod_lock);
14770                         mutex_exit(&dtrace_lock);

14772                         /*
14773                          * This shouldn't _actually_ be possible -- we're
14774                          * unloading a module that has an enabled probe in it.
14775                          * (It's normally up to the provider to make sure that
14776                          * this can't happen.)  However, because dtps_enable()
14777                          * doesn't have a failure mode, there can be an
14778                          * enable/unload race.  Upshot:  we don't want to
14779                          * assert, but we're not going to disable the
```



```
14780                          * probe, either.
14781                          */
14782                         if (dtrace_err_verbose) {
14783                                 cmn_err(CE_WARN, "unloaded module '%s' had "
14784                                     "enabled probes", ctl->mod_modname);
14785                         }

14787                         return;
14788                 }
14789         }

14791         probe = first;

14793         for (first = NULL; probe != NULL; probe = next) {
14794                 ASSERT(dtrace_probes[probe->dtpr_id - 1] == probe);

14796                 dtrace_probes[probe->dtpr_id - 1] = NULL;

14798                 next = probe->dtpr_nextmod;
14799                 dtrace_hash_remove(dtrace_bymod, probe);
14800                 dtrace_hash_remove(dtrace_byfunc, probe);
14801                 dtrace_hash_remove(dtrace_byname, probe);

14803                 if (first == NULL) {
14804                         first = probe;
14805                         probe->dtpr_nextmod = NULL;
14806                 } else {
14807                         probe->dtpr_nextmod = first;
14808                         first = probe;
14809                 }
14810         }

14812         /*
14813          * We've removed all of the module's probes from the hash chains and
14814          * from the probe array.  Now issue a dtrace_sync() to be sure that
14815          * everyone has cleared out from any probe array processing.
14816          */
14817         dtrace_sync();

14819         for (probe = first; probe != NULL; probe = first) {
14820                 first = probe->dtpr_nextmod;
14821                 prov = probe->dtpr_provider;
14822                 prov->dtpv_pops.dtps_destroy(prov->dtpv_arg, probe->dtpr_id,
14823                     probe->dtpr_arg);
14824                 kmem_free(probe->dtpr_mod, strlen(probe->dtpr_mod) + 1);
14825                 kmem_free(probe->dtpr_func, strlen(probe->dtpr_func) + 1);
14826                 kmem_free(probe->dtpr_name, strlen(probe->dtpr_name) + 1);
14827                 vmem_free(dtrace_arena, (void *)(uintptr_t)probe->dtpr_id, 1);
14828                 kmem_free(probe, sizeof (dtrace_probe_t));
14829         }

14831         mutex_exit(&dtrace_lock);
14832         mutex_exit(&mod_lock);
14833         mutex_exit(&dtrace_provider_lock);
14834 }

14836 void
14837 dtrace_suspend(void)
14838 {
14839         dtrace_probe_foreach(offsetof(dtrace_pops_t, dtps_suspend));
14840 }

14842 void
14843 dtrace_resume(void)
14844 {
14845         dtrace_probe_foreach(offsetof(dtrace_pops_t, dtps_resume));
```

```
14846 }

14848 static int
14849 dtrace_cpu_setup(cpu_setup_t what, processorid_t cpu)
14850 {
14851         ASSERT(MUTEX_HELD(&cpu_lock));
14852         mutex_enter(&dtrace_lock);

14854         switch (what) {
14855         case CPU_CONFIG: {
14856                 dtrace_state_t *state;
14857                 dtrace_optval_t *opt, rs, c;

14859                 /*
14860                  * For now, we only allocate a new buffer for anonymous state.
14861                  */
14862                 if ((state = dtrace_anon.dta_state) == NULL)
14863                         break;

14865                 if (state->dts_activity != DTRACE_ACTIVITY_ACTIVE)
14866                         break;

14868                 opt = state->dts_options;
14869                 c = opt[DTRACEOPT_CPU];

14871                 if (c != DTRACE_CPUALL && c != DTRACEOPT_UNSET && c != cpu)
14872                         break;

14874                 /*
14875                  * Regardless of what the actual policy is, we're going to
14876                  * temporarily set our resize policy to be manual.  We're
14877                  * also going to temporarily set our CPU option to denote
14878                  * the newly configured CPU.
14879                  */
14880                 rs = opt[DTRACEOPT_BUFRESIZE];
14881                 opt[DTRACEOPT_BUFRESIZE] = DTRACEOPT_BUFRESIZE_MANUAL;
14882                 opt[DTRACEOPT_CPU] = (dtrace_optval_t)cpu;

14884                 (void) dtrace_state_buffers(state);

14886                 opt[DTRACEOPT_BUFRESIZE] = rs;
14887                 opt[DTRACEOPT_CPU] = c;

14889                 break;
14890         }

14892         case CPU_UNCONFIG:
14893                 /*
14894                  * We don't free the buffer in the CPU_UNCONFIG case.  (The
14895                  * buffer will be freed when the consumer exits.)
14896                  */
14897                 break;

14899         default:
14900                 break;
14901         }

14903         mutex_exit(&dtrace_lock);
14904         return (0);
14905 }

14907 static void
14908 dtrace_cpu_setup_initial(processorid_t cpu)
14909 {
14910         (void) dtrace_cpu_setup(CPU_CONFIG, cpu);
14911 }
```



```
14913 static void
14914 dtrace_toxrange_add(uintptr_t base, uintptr_t limit)
14915 {
14916         if (dtrace_toxranges >= dtrace_toxranges_max) {
14917                 int osize, nsize;
14918                 dtrace_toxrange_t *range;

14920                 osize = dtrace_toxranges_max * sizeof (dtrace_toxrange_t);

14922                 if (osize == 0) {
14923                         ASSERT(dtrace_toxrange == NULL);
14924                         ASSERT(dtrace_toxranges_max == 0);
14925                         dtrace_toxranges_max = 1;
14926                 } else {
14927                         dtrace_toxranges_max <<= 1;
14928                 }

14930                 nsize = dtrace_toxranges_max * sizeof (dtrace_toxrange_t);
14931                 range = kmem_zalloc(nsize, KM_SLEEP);

14933                 if (dtrace_toxrange != NULL) {
14934                         ASSERT(osize != 0);
14935                         bcopy(dtrace_toxrange, range, osize);
14936                         kmem_free(dtrace_toxrange, osize);
14937                 }

14939                 dtrace_toxrange = range;
14940         }

14942         ASSERT(dtrace_toxrange[dtrace_toxranges].dtt_base == NULL);
14943         ASSERT(dtrace_toxrange[dtrace_toxranges].dtt_limit == NULL);

14945         dtrace_toxrange[dtrace_toxranges].dtt_base = base;
14946         dtrace_toxrange[dtrace_toxranges].dtt_limit = limit;
14947         dtrace_toxranges++;
14948 }

14950 static void
14951 dtrace_getf_barrier()
14952 {
14953         /*
14954          * When we have unprivileged (that is, non-DTRACE_CRV_KERNEL) enablings
14955          * that contain calls to getf(), this routine will be called on every
14956          * closef() before either the underlying vnode is released or the
14957          * file_t itself is freed.  By the time we are here, it is essential
14958          * that the file_t can no longer be accessed from a call to getf()
14959          * in probe context -- that assures that a dtrace_sync() can be used
14960          * to clear out any enablings referring to the old structures.
14961          */
14962         if (curthread->t_procp->p_zone->zone_dtrace_getf != 0 ||
14963             kcred->cr_zone->zone_dtrace_getf != 0)
14964                 dtrace_sync();
14965 }

14967 #endif /* ! codereview */
14968 /*
14969  * DTrace Driver Cookbook Functions
14970  */
14971 /*ARGSUSED*/
14972 static int
14973 dtrace_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
14974 {
14975         dtrace_provider_id_t id;
14976         dtrace_state_t *state = NULL;
14977         dtrace_enabling_t *enab;
```

```
14979         mutex_enter(&cpu_lock);
14980         mutex_enter(&dtrace_provider_lock);
14981         mutex_enter(&dtrace_lock);
14983         if (ddi_soft_state_init(&dtrace_softstate,
14984             sizeof (dtrace_state_t), 0) != 0) {
14985                 cmn_err(CE_NOTE, "/dev/dtrace failed to initialize soft state");
14986                 mutex_exit(&cpu_lock);
14987                 mutex_exit(&dtrace_provider_lock);
14988                 mutex_exit(&dtrace_lock);
14989                 return (DDI_FAILURE);
14990         }
14992         if (ddi_create_minor_node(devi, DTRACEMNR_DTRACE, S_IFCHR,
14993             DTRACEMNRN_DTRACE, DDI_PSEUDO, NULL) == DDI_FAILURE ||
14994             ddi_create_minor_node(devi, DTRACEMNR_HELPER, S_IFCHR,
14995             DTRACEMNRN_HELPER, DDI_PSEUDO, NULL) == DDI_FAILURE) {
14996                 cmn_err(CE_NOTE, "/dev/dtrace couldn't create minor nodes");
14997                 ddi_remove_minor_node(devi, NULL);
14998                 ddi_soft_state_fini(&dtrace_softstate);
14999                 mutex_exit(&cpu_lock);
15000                 mutex_exit(&dtrace_provider_lock);
15001                 mutex_exit(&dtrace_lock);
15002                 return (DDI_FAILURE);
15003         }
15005         ddi_report_dev(devi);
15006         dtrace_devi = devi;
15008         dtrace_modload = dtrace_module_loaded;
15009         dtrace_modunload = dtrace_module_unloaded;
15010         dtrace_cpu_init = dtrace_cpu_setup_initial;
15011         dtrace_helpers_cleanup = dtrace_helpers_destroy;
15012         dtrace_helpers_fork = dtrace_helpers_duplicate;
15013         dtrace_cpustart_init = dtrace_suspend;
15014         dtrace_cpustart_fini = dtrace_resume;
15015         dtrace_debugger_init = dtrace_suspend;
15016         dtrace_debugger_fini = dtrace_resume;
15018         register_cpu_setup_func((cpu_setup_func_t *)dtrace_cpu_setup, NULL);
15020         ASSERT(MUTEX_HELD(&cpu_lock));
15022         dtrace_arena = vmem_create("dtrace", (void *)1, UINT32_MAX, 1,
15023             NULL, NULL, NULL, 0, VM_SLEEP | VMC_IDENTIFIER);
15024         dtrace_minor = vmem_create("dtrace_minor", (void *)DTRACEMNRN_CLONE,
15025             UINT32_MAX - DTRACEMNRN_CLONE, 1, NULL, NULL, NULL, 0,
15026             VM_SLEEP | VMC_IDENTIFIER);
15027         dtrace_taskq = taskq_create("dtrace_taskq", 1, maxclsyspri,
15028             1, INT_MAX, 0);
15030         dtrace_state_cache = kmem_cache_create("dtrace_state_cache",
15031             sizeof (dtrace_dstate_percpu_t) * NCPU, DTRACE_STATE_ALIGN,
15032             NULL, NULL, NULL, NULL, NULL, 0);
15034         ASSERT(MUTEX_HELD(&cpu_lock));
15035         dtrace_bymod = dtrace_hash_create(offsetof(dtrace_probe_t, dtpr_mod),
15036             offsetof(dtrace_probe_t, dtpr_nextmod),
15037             offsetof(dtrace_probe_t, dtpr_prevmod));
15039         dtrace_byfunc = dtrace_hash_create(offsetof(dtrace_probe_t, dtpr_func),
15040             offsetof(dtrace_probe_t, dtpr_nextfunc),
15041             offsetof(dtrace_probe_t, dtpr_prevfunc));
15043         dtrace_byname = dtrace_hash_create(offsetof(dtrace_probe_t, dtpr_name),
```



```
15044             offsetof(dtrace_probe_t, dtpr_nextname),
15045             offsetof(dtrace_probe_t, dtpr_prevname));
15047         if (dtrace_retain_max < 1) {
15048                 cmn_err(CE_WARN, "illegal value (%lu) for dtrace_retain_max; "
15049                     "setting to 1", dtrace_retain_max);
15050                 dtrace_retain_max = 1;
15051         }
15053         /*
15054          * Now discover our toxic ranges.
15055          */
15056         dtrace_toxic_ranges(dtrace_toxrange_add);
15058         /*
15059          * Before we register ourselves as a provider to our own framework,
15060          * we would like to assert that dtrace_provider is NULL -- but that's
15061          * not true if we were loaded as a dependency of a DTrace provider.
15062          * Once we've registered, we can assert that dtrace_provider is our
15063          * pseudo provider.
15064          */
15065         (void) dtrace_register("dtrace", &dtrace_provider_attr,
15066             DTRACE_PRIV_NONE, 0, &dtrace_provider_ops, NULL, &id);
15068         ASSERT(dtrace_provider != NULL);
15069         ASSERT((dtrace_provider_id_t)dtrace_provider == id);
15071         dtrace_probeid_begin = dtrace_probe_create((dtrace_provider_id_t)
15072             dtrace_provider, NULL, NULL, "BEGIN", 0, NULL);
15073         dtrace_probeid_end = dtrace_probe_create((dtrace_provider_id_t)
15074             dtrace_provider, NULL, NULL, "END", 0, NULL);
15075         dtrace_probeid_error = dtrace_probe_create((dtrace_provider_id_t)
15076             dtrace_provider, NULL, NULL, "ERROR", 1, NULL);
15078         dtrace_anon_property();
15079         mutex_exit(&cpu_lock);
15081         /*
15082          * If DTrace helper tracing is enabled, we need to allocate the
15083          * trace buffer and initialize the values.
15084          */
15085         if (dtrace_helptrace_enabled) {
15086                 ASSERT(dtrace_helptrace_buffer == NULL);
15087                 dtrace_helptrace_buffer =
15088                     kmem_zalloc(dtrace_helptrace_bufsize, KM_SLEEP);
15089                 dtrace_helptrace_next = 0;
15090         }
15092         /*
15093          * If there are already providers, we must ask them to provide their
15094          * probes, and then match any anonymous enabling against them.  Note
15095          * that there should be no other retained enablings at this time:
15096          * the only retained enablings at this time should be the anonymous
15097          * enabling.
15098          */
15099         if (dtrace_anon.dta_enabling != NULL) {
15100                 ASSERT(dtrace_retained == dtrace_anon.dta_enabling);
15102                 dtrace_enabling_provide(NULL);
15103                 state = dtrace_anon.dta_state;
15105                 /*
15106                  * We couldn't hold cpu_lock across the above call to
15107                  * dtrace_enabling_provide(), but we must hold it to actually
15108                  * enable the probes.  We have to drop all of our locks, pick
15109                  * up cpu_lock, and regain our locks before matching the
```

```
15110                  * retained anonymous enabling.
15111                  */
15112                 mutex_exit(&dtrace_lock);
15113                 mutex_exit(&dtrace_provider_lock);
15114
15115                 mutex_enter(&cpu_lock);
15116                 mutex_enter(&dtrace_provider_lock);
15117                 mutex_enter(&dtrace_lock);
15118
15119                 if ((enab = dtrace_anon.dta_enabling) != NULL)
15120                         (void) dtrace_enabling_match(enab, NULL);
15121
15122                 mutex_exit(&cpu_lock);
15123         }
15124
15125         mutex_exit(&dtrace_lock);
15126         mutex_exit(&dtrace_provider_lock);
15127
15128         if (state != NULL) {
15129                 /*
15130                  * If we created any anonymous state, set it going now.
15131                  */
15132                 (void) dtrace_state_go(state, &dtrace_anon.dta_beganon);
15133         }
15134
15135         return (DDI_SUCCESS);
15136 }
15137
15138 /*ARGSUSED*/
15139 static int
15140 dtrace_open(dev_t *devp, int flag, int otyp, cred_t *cred_p)
15141 {
15142         dtrace_state_t *state;
15143         uint32_t priv;
15144         uid_t uid;
15145         zoneid_t zoneid;
15146
15147         if (getminor(*devp) == DTRACEMNRN_HELPER)
15148                 return (0);
15149
15150         /*
15151          * If this wasn't an open with the "helper" minor, then it must be
15152          * the "dtrace" minor.
15153          */
15154         if (getminor(*devp) != DTRACEMNRN_DTRACE)
15155                 return (ENXIO);
15156
15157         /*
15158          * If no DTRACE_PRIV_* bits are set in the credential, then the
15159          * caller lacks sufficient permission to do anything with DTrace.
15160          */
15161         dtrace_cred2priv(cred_p, &priv, &uid, &zoneid);
15162         if (priv == DTRACE_PRIV_NONE)
15163                 return (EACCES);
15164
15165         /*
15166          * Ask all providers to provide all their probes.
15167          */
15168         mutex_enter(&dtrace_provider_lock);
15169         dtrace_probe_provide(NULL, NULL);
15170         mutex_exit(&dtrace_provider_lock);
15171
15172         mutex_enter(&cpu_lock);
15173         mutex_enter(&dtrace_lock);
15174         dtrace_opens++;
15175         dtrace_membar_producer();
```



```
15176
15177         /*
15178          * If the kernel debugger is active (that is, if the kernel debugger
15179          * modified text in some way), we won't allow the open.
15180          */
15181         if (kdi_dtrace_set(KDI_DTSET_DTRACE_ACTIVATE) != 0) {
15182                 dtrace_opens--;
15183                 mutex_exit(&cpu_lock);
15184                 mutex_exit(&dtrace_lock);
15185                 return (EBUSY);
15186         }
15187
15188         state = dtrace_state_create(devp, cred_p);
15189         mutex_exit(&cpu_lock);
15190
15191         if (state == NULL) {
15192                 if (--dtrace_opens == 0 && dtrace_anon.dta_enabling == NULL)
15193                         (void) kdi_dtrace_set(KDI_DTSET_DTRACE_DEACTIVATE);
15194                 mutex_exit(&dtrace_lock);
15195                 return (EAGAIN);
15196         }
15197
15198         mutex_exit(&dtrace_lock);
15199
15200         return (0);
15201 }
15202
15203 /*ARGSUSED*/
15204 static int
15205 dtrace_close(dev_t dev, int flag, int otyp, cred_t *cred_p)
15206 {
15207         minor_t minor = getminor(dev);
15208         dtrace_state_t *state;
15209
15210         if (minor == DTRACEMNRN_HELPER)
15211                 return (0);
15212
15213         state = ddi_get_soft_state(dtrace_softstate, minor);
15214
15215         mutex_enter(&cpu_lock);
15216         mutex_enter(&dtrace_lock);
15217
15218         if (state->dts_anon) {
15219                 /*
15220                  * There is anonymous state. Destroy that first.
15221                  */
15222                 ASSERT(dtrace_anon.dta_state == NULL);
15223                 dtrace_state_destroy(state->dts_anon);
15224         }
15225
15226         dtrace_state_destroy(state);
15227         ASSERT(dtrace_opens > 0);
15228
15229         /*
15230          * Only relinquish control of the kernel debugger interface when there
15231          * are no consumers and no anonymous enablings.
15232          */
15233         if (--dtrace_opens == 0 && dtrace_anon.dta_enabling == NULL)
15234                 (void) kdi_dtrace_set(KDI_DTSET_DTRACE_DEACTIVATE);
15235
15236         mutex_exit(&dtrace_lock);
15237         mutex_exit(&cpu_lock);
15238
15239         return (0);
15240 }
```

```
15242 /*ARGSUSED*/
15243 static int
15244 dtrace_ioctl_helper(int cmd, intptr_t arg, int *rv)
15245 {
15246         int rval;
15247         dof_helper_t help, *dhp = NULL;

15249         switch (cmd) {
15250         case DTRACEHIOC_ADDDOF:
15251                 if (copyin((void *)arg, &help, sizeof (help)) != 0) {
15252                         dtrace_dof_error(NULL, "failed to copyin DOF helper");
15253                         return (EFAULT);
15254                 }

15256                 dhp = &help;
15257                 arg = (intptr_t)help.dofhp_dof;
15258                 /*FALLTHROUGH*/

15260         case DTRACEHIOC_ADD: {
15261                 dof_hdr_t *dof = dtrace_dof_copyin(arg, &rval);

15263                 if (dof == NULL)
15264                         return (rval);

15266                 mutex_enter(&dtrace_lock);

15268                 /*
15269                  * dtrace_helper_slurp() takes responsibility for the dof --
15270                  * it may free it now or it may save it and free it later.
15271                  */
15272                 if ((rval = dtrace_helper_slurp(dof, dhp)) != -1) {
15273                         *rv = rval;
15274                         rval = 0;
15275                 } else {
15276                         rval = EINVAL;
15277                 }

15279                 mutex_exit(&dtrace_lock);
15280                 return (rval);
15281         }

15283         case DTRACEHIOC_REMOVE: {
15284                 mutex_enter(&dtrace_lock);
15285                 rval = dtrace_helper_destroygen(arg);
15286                 mutex_exit(&dtrace_lock);

15288                 return (rval);
15289         }

15291         default:
15292                 break;
15293         }

15295         return (ENOTTY);
15296 }

15298 /*ARGSUSED*/
15299 static int
15300 dtrace_ioctl(dev_t dev, int cmd, intptr_t arg, int md, cred_t *cr, int *rv)
15301 {
15302         minor_t minor = getminor(dev);
15303         dtrace_state_t *state;
15304         int rval;

15306         if (minor == DTRACEMNRN_HELPER)
15307                 return (dtrace_ioctl_helper(cmd, arg, rv));
```



```
15309         state = ddi_get_soft_state(dtrace_softstate, minor);

15311         if (state->dts_anon) {
15312                 ASSERT(dtrace_anon.dta_state == NULL);
15313                 state = state->dts_anon;
15314         }

15316         switch (cmd) {
15317         case DTRACEIOC_PROVIDER: {
15318                 dtrace_providerdesc_t pvd;
15319                 dtrace_provider_t *pvp;

15321                 if (copyin((void *)arg, &pvd, sizeof (pvd)) != 0)
15322                         return (EFAULT);

15324                 pvd.dtvd_name[DTRACE_PROVNAMELEN - 1] = '\0';
15325                 mutex_enter(&dtrace_provider_lock);

15327                 for (pvp = dtrace_provider; pvp != NULL; pvp = pvp->dtpv_next) {
15328                         if (strcmp(pvp->dtpv_name, pvd.dtvd_name) == 0)
15329                                 break;
15330                 }

15332                 mutex_exit(&dtrace_provider_lock);

15334                 if (pvp == NULL)
15335                         return (ESRCH);

15337                 bcopy(&pvp->dtpv_priv, &pvd.dtvd_priv, sizeof (dtrace_ppriv_t));
15338                 bcopy(&pvp->dtpv_attr, &pvd.dtvd_attr, sizeof (dtrace_pattr_t));
15339                 if (copyout(&pvd, (void *)arg, sizeof (pvd)) != 0)
15340                         return (EFAULT);

15342                 return (0);
15343         }

15345         case DTRACEIOC_EPROBE: {
15346                 dtrace_eprobedesc_t epdesc;
15347                 dtrace_ecb_t *ecb;
15348                 dtrace_action_t *act;
15349                 void *buf;
15350                 size_t size;
15351                 uintptr_t dest;
15352                 int nrecs;

15354                 if (copyin((void *)arg, &epdesc, sizeof (epdesc)) != 0)
15355                         return (EFAULT);

15357                 mutex_enter(&dtrace_lock);

15359                 if ((ecb = dtrace_epid2ecb(state, epdesc.dtepd_epid)) == NULL) {
15360                         mutex_exit(&dtrace_lock);
15361                         return (EINVAL);
15362                 }

15364                 if (ecb->dte_probe == NULL) {
15365                         mutex_exit(&dtrace_lock);
15366                         return (EINVAL);
15367                 }

15369                 epdesc.dtepd_probeid = ecb->dte_probe->dtpr_id;
15370                 epdesc.dtepd_uarg = ecb->dte_uarg;
15371                 epdesc.dtepd_size = ecb->dte_size;

15373                 nrecs = epdesc.dtepd_nrecs;
```

```
15374                 epdesc.dtepd_nrecs = 0;
15375                 for (act = ecb->dte_action; act != NULL; act = act->dta_next) {
15376                         if (DTRACEACT_ISAGG(act->dta_kind) || act->dta_intuple)
15377                                 continue;
15379                         epdesc.dtepd_nrecs++;
15380                 }
15382                 /*
15383                  * Now that we have the size, we need to allocate a temporary
15384                  * buffer in which to store the complete description.  We need
15385                  * the temporary buffer to be able to drop dtrace_lock()
15386                  * across the copyout(), below.
15387                  */
15388                 size = sizeof (dtrace_eprobedesc_t) +
15389                     (epdesc.dtepd_nrecs * sizeof (dtrace_recdesc_t));
15391                 buf = kmem_alloc(size, KM_SLEEP);
15392                 dest = (uintptr_t)buf;
15394                 bcopy(&epdesc, (void *)dest, sizeof (epdesc));
15395                 dest += offsetof(dtrace_eprobedesc_t, dtepd_rec[0]);
15397                 for (act = ecb->dte_action; act != NULL; act = act->dta_next) {
15398                         if (DTRACEACT_ISAGG(act->dta_kind) || act->dta_intuple)
15399                                 continue;
15401                         if (nrecs-- == 0)
15402                                 break;
15404                         bcopy(&act->dta_rec, (void *)dest,
15405                             sizeof (dtrace_recdesc_t));
15406                         dest += sizeof (dtrace_recdesc_t);
15407                 }
15409                 mutex_exit(&dtrace_lock);
15411                 if (copyout(buf, (void *)arg, dest - (uintptr_t)buf) != 0) {
15412                         kmem_free(buf, size);
15413                         return (EFAULT);
15414                 }
15416                 kmem_free(buf, size);
15417                 return (0);
15418         }
15420         case DTRACEIOC_AGGDESC: {
15421                 dtrace_aggdesc_t aggdesc;
15422                 dtrace_action_t *act;
15423                 dtrace_aggregation_t *agg;
15424                 int nrecs;
15425                 uint32_t offs;
15426                 dtrace_recdesc_t *lrec;
15427                 void *buf;
15428                 size_t size;
15429                 uintptr_t dest;
15431                 if (copyin((void *)arg, &aggdesc, sizeof (aggdesc)) != 0)
15432                         return (EFAULT);
15434                 mutex_enter(&dtrace_lock);
15436                 if ((agg = dtrace_aggid2agg(state, aggdesc.dtagd_id)) == NULL) {
15437                         mutex_exit(&dtrace_lock);
15438                         return (EINVAL);
15439                 }
```



```
15441                 aggdesc.dtagd_epid = agg->dtag_ecb->dte_epid;
15443                 nrecs = aggdesc.dtagd_nrecs;
15444                 aggdesc.dtagd_nrecs = 0;
15446                 offs = agg->dtag_base;
15447                 lrec = &agg->dtag_action.dta_rec;
15448                 aggdesc.dtagd_size = lrec->dtrd_offset + lrec->dtrd_size - offs;
15450                 for (act = agg->dtag_first; ; act = act->dta_next) {
15451                         ASSERT(act->dta_intuple ||
15452                             DTRACEACT_ISAGG(act->dta_kind));
15454                         /*
15455                          * If this action has a record size of zero, it
15456                          * denotes an argument to the aggregating action.
15457                          * Because the presence of this record doesn't (or
15458                          * shouldn't) affect the way the data is interpreted,
15459                          * we don't copy it out to save user-level the
15460                          * confusion of dealing with a zero-length record.
15461                          */
15462                         if (act->dta_rec.dtrd_size == 0) {
15463                                 ASSERT(agg->dtag_hasarg);
15464                                 continue;
15465                         }
15467                         aggdesc.dtagd_nrecs++;
15469                         if (act == &agg->dtag_action)
15470                                 break;
15471                 }
15473                 /*
15474                  * Now that we have the size, we need to allocate a temporary
15475                  * buffer in which to store the complete description.  We need
15476                  * the temporary buffer to be able to drop dtrace_lock()
15477                  * across the copyout(), below.
15478                  */
15479                 size = sizeof (dtrace_aggdesc_t) +
15480                     (aggdesc.dtagd_nrecs * sizeof (dtrace_recdesc_t));
15482                 buf = kmem_alloc(size, KM_SLEEP);
15483                 dest = (uintptr_t)buf;
15485                 bcopy(&aggdesc, (void *)dest, sizeof (aggdesc));
15486                 dest += offsetof(dtrace_aggdesc_t, dtagd_rec[0]);
15488                 for (act = agg->dtag_first; ; act = act->dta_next) {
15489                         dtrace_recdesc_t rec = act->dta_rec;
15491                         /*
15492                          * See the comment in the above loop for why we pass
15493                          * over zero-length records.
15494                          */
15495                         if (rec.dtrd_size == 0) {
15496                                 ASSERT(agg->dtag_hasarg);
15497                                 continue;
15498                         }
15500                         if (nrecs-- == 0)
15501                                 break;
15503                         rec.dtrd_offset -= offs;
15504                         bcopy(&rec, (void *)dest, sizeof (rec));
15505                         dest += sizeof (dtrace_recdesc_t);
```

```
15507                           if (act == &agg->dtag_action)
15508                                   break;
15509                   }
15511                   mutex_exit(&dtrace_lock);
15513                   if (copyout(buf, (void *)arg, dest - (uintptr_t)buf) != 0) {
15514                           kmem_free(buf, size);
15515                           return (EFAULT);
15516                   }
15518                   kmem_free(buf, size);
15519                   return (0);
15520           }
15522           case DTRACEIOC_ENABLE: {
15523                   dof_hdr_t *dof;
15524                   dtrace_enabling_t *enab = NULL;
15525                   dtrace_vstate_t *vstate;
15526                   int err = 0;
15528                   *rv = 0;
15530                   /*
15531                    * If a NULL argument has been passed, we take this as our
15532                    * cue to reevaluate our enablings.
15533                    */
15534                   if (arg == NULL) {
15535                           dtrace_enabling_matchall();
15537                           return (0);
15538                   }
15540                   if ((dof = dtrace_dof_copyin(arg, &rval)) == NULL)
15541                           return (rval);
15543                   mutex_enter(&cpu_lock);
15544                   mutex_enter(&dtrace_lock);
15545                   vstate = &state->dts_vstate;
15547                   if (state->dts_activity != DTRACE_ACTIVITY_INACTIVE) {
15548                           mutex_exit(&dtrace_lock);
15549                           mutex_exit(&cpu_lock);
15550                           dtrace_dof_destroy(dof);
15551                           return (EBUSY);
15552                   }
15554                   if (dtrace_dof_slurp(dof, vstate, cr, &enab, 0, B_TRUE) != 0) {
15555                           mutex_exit(&dtrace_lock);
15556                           mutex_exit(&cpu_lock);
15557                           dtrace_dof_destroy(dof);
15558                           return (EINVAL);
15559                   }
15561                   if ((rval = dtrace_dof_options(dof, state)) != 0) {
15562                           dtrace_enabling_destroy(enab);
15563                           mutex_exit(&dtrace_lock);
15564                           mutex_exit(&cpu_lock);
15565                           dtrace_dof_destroy(dof);
15566                           return (rval);
15567                   }
15569                   if ((err = dtrace_enabling_match(enab, rv)) == 0) {
15570                           err = dtrace_enabling_retain(enab);
15571                   } else {
```



```
15572                           dtrace_enabling_destroy(enab);
15573                   }
15575                   mutex_exit(&cpu_lock);
15576                   mutex_exit(&dtrace_lock);
15577                   dtrace_dof_destroy(dof);
15579                   return (err);
15580           }
15582           case DTRACEIOC_REPLICATE: {
15583                   dtrace_repldesc_t desc;
15584                   dtrace_probedesc_t *match = &desc.dtrpd_match;
15585                   dtrace_probedesc_t *create = &desc.dtrpd_create;
15586                   int err;
15588                   if (copyin((void *)arg, &desc, sizeof (desc)) != 0)
15589                           return (EFAULT);
15591                   match->dtpd_provider[DTRACE_PROVNAMELEN - 1] = '\0';
15592                   match->dtpd_mod[DTRACE_MODNAMELEN - 1] = '\0';
15593                   match->dtpd_func[DTRACE_FUNCNAMELEN - 1] = '\0';
15594                   match->dtpd_name[DTRACE_NAMELEN - 1] = '\0';
15596                   create->dtpd_provider[DTRACE_PROVNAMELEN - 1] = '\0';
15597                   create->dtpd_mod[DTRACE_MODNAMELEN - 1] = '\0';
15598                   create->dtpd_func[DTRACE_FUNCNAMELEN - 1] = '\0';
15599                   create->dtpd_name[DTRACE_NAMELEN - 1] = '\0';
15601                   mutex_enter(&dtrace_lock);
15602                   err = dtrace_enabling_replicate(state, match, create);
15603                   mutex_exit(&dtrace_lock);
15605                   return (err);
15606           }
15608           case DTRACEIOC_PROBEMATCH:
15609           case DTRACEIOC_PROBES: {
15610                   dtrace_probe_t *probe = NULL;
15611                   dtrace_probedesc_t desc;
15612                   dtrace_probekey_t pkey;
15613                   dtrace_id_t i;
15614                   int m = 0;
15615                   uint32_t priv;
15616                   uid_t uid;
15617                   zoneid_t zoneid;
15619                   if (copyin((void *)arg, &desc, sizeof (desc)) != 0)
15620                           return (EFAULT);
15622                   desc.dtpd_provider[DTRACE_PROVNAMELEN - 1] = '\0';
15623                   desc.dtpd_mod[DTRACE_MODNAMELEN - 1] = '\0';
15624                   desc.dtpd_func[DTRACE_FUNCNAMELEN - 1] = '\0';
15625                   desc.dtpd_name[DTRACE_NAMELEN - 1] = '\0';
15627                   /*
15628                    * Before we attempt to match this probe, we want to give
15629                    * all providers the opportunity to provide it.
15630                    */
15631                   if (desc.dtpd_id == DTRACE_IDNONE) {
15632                           mutex_enter(&dtrace_provider_lock);
15633                           dtrace_probe_provide(&desc, NULL);
15634                           mutex_exit(&dtrace_provider_lock);
15635                           desc.dtpd_id++;
15636                   }
```

```
15638                 if (cmd == DTRACEIOC_PROBEMATCH)  {
15639                         dtrace_probekey(&desc, &pkey);
15640                         pkey.dtpk_id = DTRACE_IDNONE;
15641                 }

15643                 dtrace_cred2priv(cr, &priv, &uid, &zoneid);

15645                 mutex_enter(&dtrace_lock);

15647                 if (cmd == DTRACEIOC_PROBEMATCH) {
15648                         for (i = desc.dtpd_id; i <= dtrace_nprobes; i++) {
15649                                 if ((probe = dtrace_probes[i - 1]) != NULL &&
15650                                     (m = dtrace_match_probe(probe, &pkey,
15651                                     priv, uid, zoneid)) != 0)
15652                                         break;
15653                         }

15655                         if (m < 0) {
15656                                 mutex_exit(&dtrace_lock);
15657                                 return (EINVAL);
15658                         }

15660                 } else {
15661                         for (i = desc.dtpd_id; i <= dtrace_nprobes; i++) {
15662                                 if ((probe = dtrace_probes[i - 1]) != NULL &&
15663                                     dtrace_match_priv(probe, priv, uid, zoneid))
15664                                         break;
15665                         }
15666                 }

15668                 if (probe == NULL) {
15669                         mutex_exit(&dtrace_lock);
15670                         return (ESRCH);
15671                 }

15673                 dtrace_probe_description(probe, &desc);
15674                 mutex_exit(&dtrace_lock);

15676                 if (copyout(&desc, (void *)arg, sizeof (desc)) != 0)
15677                         return (EFAULT);

15679                 return (0);
15680         }

15682         case DTRACEIOC_PROBEARG: {
15683                 dtrace_argdesc_t desc;
15684                 dtrace_probe_t *probe;
15685                 dtrace_provider_t *prov;

15687                 if (copyin((void *)arg, &desc, sizeof (desc)) != 0)
15688                         return (EFAULT);

15690                 if (desc.dtargd_id == DTRACE_IDNONE)
15691                         return (EINVAL);

15693                 if (desc.dtargd_ndx == DTRACE_ARGNONE)
15694                         return (EINVAL);

15696                 mutex_enter(&dtrace_provider_lock);
15697                 mutex_enter(&mod_lock);
15698                 mutex_enter(&dtrace_lock);

15700                 if (desc.dtargd_id > dtrace_nprobes) {
15701                         mutex_exit(&dtrace_lock);
15702                         mutex_exit(&mod_lock);
15703                         mutex_exit(&dtrace_provider_lock);
```



```
15704                         return (EINVAL);
15705                 }

15707                 if ((probe = dtrace_probes[desc.dtargd_id - 1]) == NULL) {
15708                         mutex_exit(&dtrace_lock);
15709                         mutex_exit(&mod_lock);
15710                         mutex_exit(&dtrace_provider_lock);
15711                         return (EINVAL);
15712                 }

15714                 mutex_exit(&dtrace_lock);

15716                 prov = probe->dtpr_provider;

15718                 if (prov->dtpv_pops.dtps_getargdesc == NULL) {
15719                         /*
15720                          * There isn't any typed information for this probe.
15721                          * Set the argument number to DTRACE_ARGNONE.
15722                          */
15723                         desc.dtargd_ndx = DTRACE_ARGNONE;
15724                 } else {
15725                         desc.dtargd_native[0] = '\0';
15726                         desc.dtargd_xlate[0] = '\0';
15727                         desc.dtargd_mapping = desc.dtargd_ndx;

15729                         prov->dtpv_pops.dtps_getargdesc(prov->dtpv_arg,
15730                             probe->dtpr_id, probe->dtpr_arg, &desc);
15731                 }

15733                 mutex_exit(&mod_lock);
15734                 mutex_exit(&dtrace_provider_lock);

15736                 if (copyout(&desc, (void *)arg, sizeof (desc)) != 0)
15737                         return (EFAULT);

15739                 return (0);
15740         }

15742         case DTRACEIOC_GO: {
15743                 processorid_t cpuid;
15744                 rval = dtrace_state_go(state, &cpuid);

15746                 if (rval != 0)
15747                         return (rval);

15749                 if (copyout(&cpuid, (void *)arg, sizeof (cpuid)) != 0)
15750                         return (EFAULT);

15752                 return (0);
15753         }

15755         case DTRACEIOC_STOP: {
15756                 processorid_t cpuid;

15758                 mutex_enter(&dtrace_lock);
15759                 rval = dtrace_state_stop(state, &cpuid);
15760                 mutex_exit(&dtrace_lock);

15762                 if (rval != 0)
15763                         return (rval);

15765                 if (copyout(&cpuid, (void *)arg, sizeof (cpuid)) != 0)
15766                         return (EFAULT);

15768                 return (0);
15769         }
```

```
15771         case DTRACEIOC_DOFGET: {
15772                 dof_hdr_t hdr, *dof;
15773                 uint64_t len;

15775                 if (copyin((void *)arg, &hdr, sizeof (hdr)) != 0)
15776                         return (EFAULT);

15778                 mutex_enter(&dtrace_lock);
15779                 dof = dtrace_dof_create(state);
15780                 mutex_exit(&dtrace_lock);

15782                 len = MIN(hdr.dofh_loadsz, dof->dofh_loadsz);
15783                 rval = copyout(dof, (void *)arg, len);
15784                 dtrace_dof_destroy(dof);

15786                 return (rval == 0 ? 0 : EFAULT);
15787         }

15789         case DTRACEIOC_AGGSNAP:
15790         case DTRACEIOC_BUFSNAP: {
15791                 dtrace_bufdesc_t desc;
15792                 caddr_t cached;
15793                 dtrace_buffer_t *buf;

15795                 if (copyin((void *)arg, &desc, sizeof (desc)) != 0)
15796                         return (EFAULT);

15798                 if (desc.dtbd_cpu < 0 || desc.dtbd_cpu >= NCPU)
15799                         return (EINVAL);

15801                 mutex_enter(&dtrace_lock);

15803                 if (cmd == DTRACEIOC_BUFSNAP) {
15804                         buf = &state->dts_buffer[desc.dtbd_cpu];
15805                 } else {
15806                         buf = &state->dts_aggbuffer[desc.dtbd_cpu];
15807                 }

15809                 if (buf->dtb_flags & (DTRACEBUF_RING | DTRACEBUF_FILL)) {
15810                         size_t sz = buf->dtb_offset;

15812                         if (state->dts_activity != DTRACE_ACTIVITY_STOPPED) {
15813                                 mutex_exit(&dtrace_lock);
15814                                 return (EBUSY);
15815                         }

15817                         /*
15818                          * If this buffer has already been consumed, we're
15819                          * going to indicate that there's nothing left here
15820                          * to consume.
15821                          */
15822                         if (buf->dtb_flags & DTRACEBUF_CONSUMED) {
15823                                 mutex_exit(&dtrace_lock);

15825                                 desc.dtbd_size = 0;
15826                                 desc.dtbd_drops = 0;
15827                                 desc.dtbd_errors = 0;
15828                                 desc.dtbd_oldest = 0;
15829                                 sz = sizeof (desc);

15831                                 if (copyout(&desc, (void *)arg, sz) != 0)
15832                                         return (EFAULT);

15834                                 return (0);
15835                         }
```



```
15837                         /*
15838                          * If this is a ring buffer that has wrapped, we want
15839                          * to copy the whole thing out.
15840                          */
15841                         if (buf->dtb_flags & DTRACEBUF_WRAPPED) {
15842                                 dtrace_buffer_polish(buf);
15843                                 sz = buf->dtb_size;
15844                         }

15846                         if (copyout(buf->dtb_tomax, desc.dtbd_data, sz) != 0) {
15847                                 mutex_exit(&dtrace_lock);
15848                                 return (EFAULT);
15849                         }

15851                         desc.dtbd_size = sz;
15852                         desc.dtbd_drops = buf->dtb_drops;
15853                         desc.dtbd_errors = buf->dtb_errors;
15854                         desc.dtbd_oldest = buf->dtb_xamot_offset;
15855                         desc.dtbd_timestamp = dtrace_gethrtime();

15857                         mutex_exit(&dtrace_lock);

15859                         if (copyout(&desc, (void *)arg, sizeof (desc)) != 0)
15860                                 return (EFAULT);

15862                         buf->dtb_flags |= DTRACEBUF_CONSUMED;

15864                         return (0);
15865                 }

15867                 if (buf->dtb_tomax == NULL) {
15868                         ASSERT(buf->dtb_xamot == NULL);
15869                         mutex_exit(&dtrace_lock);
15870                         return (ENOENT);
15871                 }

15873                 cached = buf->dtb_tomax;
15874                 ASSERT(!(buf->dtb_flags & DTRACEBUF_NOSWITCH));

15876                 dtrace_xcall(desc.dtbd_cpu,
15877                     (dtrace_xcall_t)dtrace_buffer_switch, buf);

15879                 state->dts_errors += buf->dtb_xamot_errors;

15881                 /*
15882                  * If the buffers did not actually switch, then the cross call
15883                  * did not take place -- presumably because the given CPU is
15884                  * not in the ready set.  If this is the case, we'll return
15885                  * ENOENT.
15886                  */
15887                 if (buf->dtb_tomax == cached) {
15888                         ASSERT(buf->dtb_xamot != cached);
15889                         mutex_exit(&dtrace_lock);
15890                         return (ENOENT);
15891                 }

15893                 ASSERT(cached == buf->dtb_xamot);

15895                 /*
15896                  * We have our snapshot; now copy it out.
15897                  */
15898                 if (copyout(buf->dtb_xamot, desc.dtbd_data,
15899                     buf->dtb_xamot_offset) != 0) {
15900                         mutex_exit(&dtrace_lock);
15901                         return (EFAULT);
```

```
15902                 }

15904                 desc.dtbd_size = buf->dtb_xamot_offset;
15905                 desc.dtbd_drops = buf->dtb_xamot_drops;
15906                 desc.dtbd_errors = buf->dtb_xamot_errors;
15907                 desc.dtbd_oldest = 0;
15908                 desc.dtbd_timestamp = buf->dtb_switched;

15910                 mutex_exit(&dtrace_lock);

15912                 /*
15913                  * Finally, copy out the buffer description.
15914                  */
15915                 if (copyout(&desc, (void *)arg, sizeof (desc)) != 0)
15916                         return (EFAULT);

15918                 return (0);
15919         }

15921         case DTRACEIOC_CONF: {
15922                 dtrace_conf_t conf;

15924                 bzero(&conf, sizeof (conf));
15925                 conf.dtc_difversion = DIF_VERSION;
15926                 conf.dtc_difintregs = DIF_DIR_NREGS;
15927                 conf.dtc_diftupregs = DIF_DTR_NREGS;
15928                 conf.dtc_ctfmodel = CTF_MODEL_NATIVE;

15930                 if (copyout(&conf, (void *)arg, sizeof (conf)) != 0)
15931                         return (EFAULT);

15933                 return (0);
15934         }

15936         case DTRACEIOC_STATUS: {
15937                 dtrace_status_t stat;
15938                 dtrace_dstate_t *dstate;
15939                 int i, j;
15940                 uint64_t nerrs;

15942                 /*
15943                  * See the comment in dtrace_state_deadman() for the reason
15944                  * for setting dts_laststatus to INT64_MAX before setting
15945                  * it to the correct value.
15946                  */
15947                 state->dts_laststatus = INT64_MAX;
15948                 dtrace_membar_producer();
15949                 state->dts_laststatus = dtrace_gethrtime();

15951                 bzero(&stat, sizeof (stat));

15953                 mutex_enter(&dtrace_lock);

15955                 if (state->dts_activity == DTRACE_ACTIVITY_INACTIVE) {
15956                         mutex_exit(&dtrace_lock);
15957                         return (ENOENT);
15958                 }

15960                 if (state->dts_activity == DTRACE_ACTIVITY_DRAINING)
15961                         stat.dtst_exiting = 1;

15963                 nerrs = state->dts_errors;
15964                 dstate = &state->dts_vstate.dtvs_dynvars;

15966                 for (i = 0; i < NCPU; i++) {
15967                         dtrace_dstate_percpu_t *dcpu = &dstate->dtds_percpu[i];
```



```
15969                         stat.dtst_dyndrops += dcpu->dtdsc_drops;
15970                         stat.dtst_dyndrops_dirty += dcpu->dtdsc_dirty_drops;
15971                         stat.dtst_dyndrops_rinsing += dcpu->dtdsc_rinsing_drops;

15973                         if (state->dts_buffer[i].dtb_flags & DTRACEBUF_FULL)
15974                                 stat.dtst_filled++;

15976                         nerrs += state->dts_buffer[i].dtb_errors;

15978                         for (j = 0; j < state->dts_nspeculations; j++) {
15979                                 dtrace_speculation_t *spec;
15980                                 dtrace_buffer_t *buf;

15982                                 spec = &state->dts_speculations[j];
15983                                 buf = &spec->dtsp_buffer[i];
15984                                 stat.dtst_specdrops += buf->dtb_xamot_drops;
15985                         }
15986                 }

15988                 stat.dtst_specdrops_busy = state->dts_speculations_busy;
15989                 stat.dtst_specdrops_unavail = state->dts_speculations_unavail;
15990                 stat.dtst_stkstroverflows = state->dts_stkstroverflows;
15991                 stat.dtst_dblerrors = state->dts_dblerrors;
15992                 stat.dtst_killed =
15993                     (state->dts_activity == DTRACE_ACTIVITY_KILLED);
15994                 stat.dtst_errors = nerrs;

15996                 mutex_exit(&dtrace_lock);

15998                 if (copyout(&stat, (void *)arg, sizeof (stat)) != 0)
15999                         return (EFAULT);

16001                 return (0);
16002         }

16004         case DTRACEIOC_FORMAT: {
16005                 dtrace_fmtdesc_t fmt;
16006                 char *str;
16007                 int len;

16009                 if (copyin((void *)arg, &fmt, sizeof (fmt)) != 0)
16010                         return (EFAULT);

16012                 mutex_enter(&dtrace_lock);

16014                 if (fmt.dtfd_format == 0 ||
16015                     fmt.dtfd_format > state->dts_nformats) {
16016                         mutex_exit(&dtrace_lock);
16017                         return (EINVAL);
16018                 }

16020                 /*
16021                  * Format strings are allocated contiguously and they are
16022                  * never freed; if a format index is less than the number
16023                  * of formats, we can assert that the format map is non-NULL
16024                  * and that the format for the specified index is non-NULL.
16025                  */
16026                 ASSERT(state->dts_formats != NULL);
16027                 str = state->dts_formats[fmt.dtfd_format - 1];
16028                 ASSERT(str != NULL);

16030                 len = strlen(str) + 1;

16032                 if (len > fmt.dtfd_length) {
16033                         fmt.dtfd_length = len;
```

```
16035                         if (copyout(&fmt, (void *)arg, sizeof (fmt)) != 0) {
16036                                 mutex_exit(&dtrace_lock);
16037                                 return (EINVAL);
16038                         }
16039                 } else {
16040                         if (copyout(str, fmt.dtfd_string, len) != 0) {
16041                                 mutex_exit(&dtrace_lock);
16042                                 return (EINVAL);
16043                         }
16044                 }

16046                 mutex_exit(&dtrace_lock);
16047                 return (0);
16048         }

16050         default:
16051                 break;
16052         }

16054         return (ENOTTY);
16055 }

16057 /*ARGSUSED*/
16058 static int
16059 dtrace_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
16060 {
16061         dtrace_state_t *state;

16063         switch (cmd) {
16064         case DDI_DETACH:
16065                 break;

16067         case DDI_SUSPEND:
16068                 return (DDI_SUCCESS);

16070         default:
16071                 return (DDI_FAILURE);
16072         }

16074         mutex_enter(&cpu_lock);
16075         mutex_enter(&dtrace_provider_lock);
16076         mutex_enter(&dtrace_lock);

16078         ASSERT(dtrace_opens == 0);

16080         if (dtrace_helpers > 0) {
16081                 mutex_exit(&dtrace_provider_lock);
16082                 mutex_exit(&dtrace_lock);
16083                 mutex_exit(&cpu_lock);
16084                 return (DDI_FAILURE);
16085         }

16087         if (dtrace_unregister((dtrace_provider_id_t)dtrace_provider) != 0) {
16088                 mutex_exit(&dtrace_provider_lock);
16089                 mutex_exit(&dtrace_lock);
16090                 mutex_exit(&cpu_lock);
16091                 return (DDI_FAILURE);
16092         }

16094         dtrace_provider = NULL;

16096         if ((state = dtrace_anon_grab()) != NULL) {
16097                 /*
16098                  * If there were ECBs on this state, the provider should
16099                  * have not been allowed to detach; assert that there is
```



```
16100                  * none.
16101                  */
16102                 ASSERT(state->dts_necbs == 0);
16103                 dtrace_state_destroy(state);

16105                 /*
16106                  * If we're being detached with anonymous state, we need to
16107                  * indicate to the kernel debugger that DTrace is now inactive.
16108                  */
16109                 (void) kdi_dtrace_set(KDI_DTSET_DTRACE_DEACTIVATE);
16110         }

16112         bzero(&dtrace_anon, sizeof (dtrace_anon_t));
16113         unregister_cpu_setup_func((cpu_setup_func_t *)dtrace_cpu_setup, NULL);
16114         dtrace_cpu_init = NULL;
16115         dtrace_helpers_cleanup = NULL;
16116         dtrace_helpers_fork = NULL;
16117         dtrace_cpustart_init = NULL;
16118         dtrace_cpustart_fini = NULL;
16119         dtrace_debugger_init = NULL;
16120         dtrace_debugger_fini = NULL;
16121         dtrace_modload = NULL;
16122         dtrace_modunload = NULL;

16124         ASSERT(dtrace_getf == 0);
16125         ASSERT(dtrace_closef == NULL);

16127 #endif /* ! codereview */
16128         mutex_exit(&cpu_lock);

16130         if (dtrace_helptrace_enabled) {
16131                 kmem_free(dtrace_helptrace_buffer, dtrace_helptrace_bufsize);
16132                 dtrace_helptrace_buffer = NULL;
16133         }

16135         kmem_free(dtrace_probes, dtrace_nprobes * sizeof (dtrace_probe_t *));
16136         dtrace_probes = NULL;
16137         dtrace_nprobes = 0;

16139         dtrace_hash_destroy(dtrace_bymod);
16140         dtrace_hash_destroy(dtrace_byfunc);
16141         dtrace_hash_destroy(dtrace_byname);
16142         dtrace_bymod = NULL;
16143         dtrace_byfunc = NULL;
16144         dtrace_byname = NULL;

16146         kmem_cache_destroy(dtrace_state_cache);
16147         vmem_destroy(dtrace_minor);
16148         vmem_destroy(dtrace_arena);

16150         if (dtrace_toxrange != NULL) {
16151                 kmem_free(dtrace_toxrange,
16152                     dtrace_toxranges_max * sizeof (dtrace_toxrange_t));
16153                 dtrace_toxrange = NULL;
16154                 dtrace_toxranges = 0;
16155                 dtrace_toxranges_max = 0;
16156         }

16158         ddi_remove_minor_node(dtrace_devi, NULL);
16159         dtrace_devi = NULL;

16161         ddi_soft_state_fini(&dtrace_softstate);

16163         ASSERT(dtrace_vtime_references == 0);
16164         ASSERT(dtrace_opens == 0);
16165         ASSERT(dtrace_retained == NULL);
```

```
16167         mutex_exit(&dtrace_lock);
16168         mutex_exit(&dtrace_provider_lock);

16170         /*
16171          * We don't destroy the task queue until after we have dropped our
16172          * locks (taskq_destroy() may block on running tasks).  To prevent
16173          * attempting to do work after we have effectively detached but before
16174          * the task queue has been destroyed, all tasks dispatched via the
16175          * task queue must check that DTrace is still attached before
16176          * performing any operation.
16177          */
16178         taskq_destroy(dtrace_taskq);
16179         dtrace_taskq = NULL;

16181         return (DDI_SUCCESS);
16182 }

16184 /*ARGSUSED*/
16185 static int
16186 dtrace_info(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
16187 {
16188         int error;

16190         switch (infocmd) {
16191         case DDI_INFO_DEVT2DEVINFO:
16192                 *result = (void *)dtrace_devi;
16193                 error = DDI_SUCCESS;
16194                 break;
16195         case DDI_INFO_DEVT2INSTANCE:
16196                 *result = (void *)0;
16197                 error = DDI_SUCCESS;
16198                 break;
16199         default:
16200                 error = DDI_FAILURE;
16201         }
16202         return (error);
16203 }

16205 static struct cb_ops dtrace_cb_ops = {
16206         dtrace_open,            /* open */
16207         dtrace_close,           /* close */
16208         nulldev,                /* strategy */
16209         nulldev,                /* print */
16210         nodev,                  /* dump */
16211         nodev,                  /* read */
16212         nodev,                  /* write */
16213         dtrace_ioctl,           /* ioctl */
16214         nodev,                  /* devmap */
16215         nodev,                  /* mmap */
16216         nodev,                  /* segmap */
16217         nochpoll,               /* poll */
16218         ddi_prop_op,            /* cb_prop_op */
16219         0,                      /* streamtab  */
16220         D_NEW | D_MP            /* Driver compatibility flag */
16221 };

16223 static struct dev_ops dtrace_ops = {
16224         DEVO_REV,               /* devo_rev */
16225         0,                      /* refcnt */
16226         dtrace_info,            /* get_dev_info */
16227         nulldev,                /* identify */
16228         nulldev,                /* probe */
16229         dtrace_attach,          /* attach */
16230         dtrace_detach,          /* detach */
16231         nodev,                  /* reset */
```



```
16232         &dtrace_cb_ops,         /* driver operations */
16233         NULL,                   /* bus operations */
16234         nodev,                  /* dev power */
16235         ddi_quiesce_not_needed,         /* quiesce */
16236 };

16238 static struct modldrv modldrv = {
16239         &mod_driverops,         /* module type (this is a pseudo driver) */
16240         "Dynamic Tracing",      /* name of module */
16241         &dtrace_ops,            /* driver ops */
16242 };

16244 static struct modlinkage modlinkage = {
16245         MODREV_1,
16246         (void *)&modldrv,
16247         NULL
16248 };

16250 int
16251 _init(void)
16252 {
16253         return (mod_install(&modlinkage));
16254 }

16256 int
16257 _info(struct modinfo *modinfop)
16258 {
16259         return (mod_info(&modlinkage, modinfop));
16260 }

16262 int
16263 _fini(void)
16264 {
16265         return (mod_remove(&modlinkage));
16266 }
```

```
**********************************************************
   55331 Tue Jan 14 16:50:02 2014
new/usr/src/uts/common/dtrace/sdt_subr.c
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2004, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
#endif /* ! codereview */
 */

#include <sys/sdt_impl.h>

static dtrace_pattr_t vtrace_attr = {
{ DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_ISA },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_UNSTABLE, DTRACE_STABILITY_UNSTABLE, DTRACE_CLASS_ISA },
};

static dtrace_pattr_t info_attr = {
{ DTRACE_STABILITY_EVOLVING, DTRACE_STABILITY_EVOLVING, DTRACE_CLASS_ISA },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_EVOLVING, DTRACE_STABILITY_EVOLVING, DTRACE_CLASS_ISA },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_ISA },
};

static dtrace_pattr_t fc_attr = {
{ DTRACE_STABILITY_EVOLVING, DTRACE_STABILITY_EVOLVING, DTRACE_CLASS_ISA },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_ISA },
{ DTRACE_STABILITY_EVOLVING, DTRACE_STABILITY_EVOLVING, DTRACE_CLASS_ISA },
};

static dtrace_pattr_t fpu_attr = {
{ DTRACE_STABILITY_EVOLVING, DTRACE_STABILITY_EVOLVING, DTRACE_CLASS_ISA },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_EVOLVING, DTRACE_STABILITY_EVOLVING, DTRACE_CLASS_CPU },
```



```
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_ISA },
};

static dtrace_pattr_t fsinfo_attr = {
{ DTRACE_STABILITY_EVOLVING, DTRACE_STABILITY_EVOLVING, DTRACE_CLASS_ISA },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_EVOLVING, DTRACE_STABILITY_EVOLVING, DTRACE_CLASS_ISA },
};

static dtrace_pattr_t stab_attr = {
{ DTRACE_STABILITY_EVOLVING, DTRACE_STABILITY_EVOLVING, DTRACE_CLASS_ISA },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_EVOLVING, DTRACE_STABILITY_EVOLVING, DTRACE_CLASS_ISA },
{ DTRACE_STABILITY_EVOLVING, DTRACE_STABILITY_EVOLVING, DTRACE_CLASS_ISA },
};

static dtrace_pattr_t sdt_attr = {
{ DTRACE_STABILITY_EVOLVING, DTRACE_STABILITY_EVOLVING, DTRACE_CLASS_ISA },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_ISA },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_ISA },
};

static dtrace_pattr_t xpv_attr = {
{ DTRACE_STABILITY_EVOLVING, DTRACE_STABILITY_EVOLVING, DTRACE_CLASS_PLATFORM },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_PLATFORM },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_PLATFORM },
};

static dtrace_pattr_t iscsi_attr = {
{ DTRACE_STABILITY_EVOLVING, DTRACE_STABILITY_EVOLVING, DTRACE_CLASS_ISA },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
{ DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_ISA },
{ DTRACE_STABILITY_EVOLVING, DTRACE_STABILITY_EVOLVING, DTRACE_CLASS_ISA },
};

sdt_provider_t sdt_providers[] = {
        { "vtrace", "__vtrace_", &vtrace_attr },
        { "sysinfo", "__cpu_sysinfo_", &info_attr, DTRACE_PRIV_USER },
        { "vminfo", "__cpu_vminfo_", &info_attr, DTRACE_PRIV_USER },
        { "fpuinfo", "__fpuinfo_", &fpu_attr },
        { "sched", "__sched_", &stab_attr, DTRACE_PRIV_USER },
        { "proc", "__proc_", &stab_attr, DTRACE_PRIV_USER },
        { "io", "__io_", &stab_attr },
        { "ip", "__ip_", &stab_attr },
        { "tcp", "__tcp_", &stab_attr },
        { "udp", "__udp_", &stab_attr },
        { "mib", "__mib_", &stab_attr },
        { "fsinfo", "__fsinfo_", &fsinfo_attr },
        { "iscsi", "__iscsi_", &iscsi_attr },
        { "nfsv3", "__nfsv3_", &stab_attr },
        { "nfsv4", "__nfsv4_", &stab_attr },
        { "xpv", "__xpv_", &xpv_attr },
        { "fc", "__fc_", &fc_attr },
        { "srp", "__srp_", &fc_attr },
        { "sysevent", "__sysevent_", &stab_attr },
        { "sdt", NULL, &sdt_attr },
        { "vtrace", "__vtrace_", &vtrace_attr, 0 },
        { "sysinfo", "__cpu_sysinfo_", &info_attr, 0 },
```

```
  25         { "vminfo", "__cpu_vminfo_", &info_attr, 0 },
  26         { "fpuinfo", "__fpuinfo_", &fpu_attr, 0 },
  27         { "sched", "__sched_", &stab_attr, 0 },
  28         { "proc", "__proc_", &stab_attr, 0 },
  29         { "io", "__io_", &stab_attr, 0 },
  30         { "ip", "__ip_", &stab_attr, 0 },
  31         { "tcp", "__tcp_", &stab_attr, 0 },
  32         { "udp", "__udp_", &stab_attr, 0 },
  33         { "mib", "__mib_", &stab_attr, 0 },
  34         { "fsinfo", "__fsinfo_", &fsinfo_attr, 0 },
  35         { "iscsi", "__iscsi_", &iscsi_attr, 0 },
  36         { "nfsv3", "__nfsv3_", &stab_attr, 0 },
  37         { "nfsv4", "__nfsv4_", &stab_attr, 0 },
  38         { "xpv", "__xpv_", &xpv_attr, 0 },
  39         { "fc", "__fc_", &fc_attr, 0 },
  40         { "srp", "__srp_", &fc_attr, 0 },
  41         { "sysevent", "__sysevent_", &stab_attr, 0 },
  42         { "sdt", NULL, &sdt_attr, 0 },
 122         { NULL }
 123 };
```

*unchanged_portion_omitted_*

```
1159 /*ARGSUSED*/
1160 int
1161 sdt_mode(void *arg, dtrace_id_t id, void *parg)
1162 {
1163         /*
1164          * We tell DTrace that we're in kernel mode, that the firing needs to
1165          * be dropped for anything that doesn't have necessary privileges, and
1166          * that it needs to be restricted for anything that has restricted
1167          * (i.e., not all-zone) privileges.
1168          */
1169         return (DTRACE_MODE_KERNEL | DTRACE_MODE_NOPRIV_DROP |
1170             DTRACE_MODE_LIMITEDPRIV_RESTRICT);
1171 }

1173 /*ARGSUSED*/
1174 #endif /* ! codereview */
1175 void
1176 sdt_getargdesc(void *arg, dtrace_id_t id, void *parg, dtrace_argdesc_t *desc)
1177 {
1178         sdt_probe_t *sdp = parg;
1179         int i;

1181         desc->dtargd_native[0] = '\0';
1182         desc->dtargd_xlate[0] = '\0';

1184         for (i = 0; sdt_args[i].sda_provider != NULL; i++) {
1185                 sdt_argdesc_t *a = &sdt_args[i];

1187                 if (strcmp(sdp->sdp_provider->sdtp_name, a->sda_provider) != 0)
1188                         continue;

1190                 if (a->sda_name != NULL &&
1191                     strcmp(sdp->sdp_name, a->sda_name) != 0)
1192                         continue;

1194                 if (desc->dtargd_ndx != a->sda_ndx)
1195                         continue;

1197                 if (a->sda_native != NULL)
1198                         (void) strcpy(desc->dtargd_native, a->sda_native);

1200                 if (a->sda_xlate != NULL)
1201                         (void) strcpy(desc->dtargd_xlate, a->sda_xlate);
```



```
1203                 desc->dtargd_mapping = a->sda_mapping;
1204                 return;
1205         }

1207         desc->dtargd_ndx = DTRACE_ARGNONE;
1208 }
```

```
**********************************************************
    9558 Tue Jan 14 16:50:03 2014
new/usr/src/uts/common/os/dtrace_subr.c
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

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

#include <sys/dtrace.h>
#include <sys/cmn_err.h>
#include <sys/tnf.h>
#include <sys/atomic.h>
#include <sys/prsystm.h>
#include <sys/modctl.h>
#include <sys/aio_impl.h>

#ifdef __sparc
#include <sys/privregs.h>
#endif

void (*dtrace_cpu_init)(processorid_t);
void (*dtrace_modload)(struct modctl *);
void (*dtrace_modunload)(struct modctl *);
void (*dtrace_helpers_cleanup)(void);
void (*dtrace_helpers_fork)(proc_t *, proc_t *);
void (*dtrace_cpustart_init)(void);
void (*dtrace_cpustart_fini)(void);
void (*dtrace_cpc_fire)(uint64_t);
void (*dtrace_closef)(void);
#endif /* ! codereview */

void (*dtrace_debugger_init)(void);
void (*dtrace_debugger_fini)(void);

dtrace_vtime_state_t dtrace_vtime_active = 0;
dtrace_cacheid_t dtrace_predcache_id = DTRACE_CACHEIDNONE + 1;

/*
 * dtrace_cpc_in_use usage statement: this global variable is used by the cpc
```



```
 * hardware overflow interrupt handler and the kernel cpc framework to check
 * whether or not the DTrace cpc provider is currently in use. The variable is
 * set before counters are enabled with the first enabling and cleared when
 * the last enabling is disabled. Its value at any given time indicates the
 * number of active dcpc based enablings. The global 'kcpc_cpuctx_lock' rwlock
 * is held during initial setting to protect races between kcpc_open() and the
 * first enabling. The locking provided by the DTrace subsystem, the kernel
 * cpc framework and the cpu management framework protect consumers from race
 * conditions on enabling and disabling probes.
 */
uint32_t dtrace_cpc_in_use = 0;

typedef struct dtrace_hrestime {
        lock_t          dthr_lock;              /* lock for this element */
        timestruc_t     dthr_hrestime;          /* hrestime value */
        int64_t         dthr_adj;               /* hrestime_adj value */
        hrtime_t        dthr_hrtime;            /* hrtime value */
} dtrace_hrestime_t;

static dtrace_hrestime_t dtrace_hrestime[2];

/*
 * Making available adjustable high-resolution time in DTrace is regrettably
 * more complicated than one might think it should be.  The problem is that
 * the variables related to adjusted high-resolution time (hrestime,
 * hrestime_adj and friends) are adjusted under hres_lock -- and this lock may
 * be held when we enter probe context.  One might think that we could address
 * this by having a single snapshot copy that is stored under a different lock
 * from hres_tick(), using the snapshot iff hres_lock is locked in probe
 * context.  Unfortunately, this too won't work:  because hres_lock is grabbed
 * in more than just hres_tick() context, we could enter probe context
 * concurrently on two different CPUs with both locks (hres_lock and the
 * snapshot lock) held.  As this implies, the fundamental problem is that we
 * need to have access to a snapshot of these variables that we _know_ will
 * not be locked in probe context.  To effect this, we have two snapshots
 * protected by two different locks, and we mandate that these snapshots are
 * recorded in succession by a single thread calling dtrace_hres_tick().  (We
 * assure this by calling it out of the same CY_HIGH_LEVEL cyclic that calls
 * hres_tick().)  A single thread can't be in two places at once:  one of the
 * snapshot locks is guaranteed to be unheld at all times.  The
 * dtrace_gethrestime() algorithm is thus to check first one snapshot and then
 * the other to find the unlocked snapshot.
 */
void
dtrace_hres_tick(void)
{
        int i;
        ushort_t spl;

        for (i = 0; i < 2; i++) {
                dtrace_hrestime_t tmp;

                spl = hr_clock_lock();
                tmp.dthr_hrestime = hrestime;
                tmp.dthr_adj = hrestime_adj;
                tmp.dthr_hrtime = dtrace_gethrtime();
                hr_clock_unlock(spl);

                lock_set(&dtrace_hrestime[i].dthr_lock);
                dtrace_hrestime[i].dthr_hrestime = tmp.dthr_hrestime;
                dtrace_hrestime[i].dthr_adj = tmp.dthr_adj;
                dtrace_hrestime[i].dthr_hrtime = tmp.dthr_hrtime;
                dtrace_membar_producer();

                /*
                 * To allow for lock-free examination of this lock, we use
```

```
124                  * the same trick that is used hres_lock; for more details,
125                  * see the description of this technique in sun4u/sys/clock.h.
126                  */
127                 dtrace_hrestime[i].dthr_lock++;
128         }
129 }

131 hrtime_t
132 dtrace_gethrestime(void)
133 {
134         dtrace_hrestime_t snap;
135         hrtime_t now;
136         int i = 0, adj, nslt;

138         for (;;) {
139                 snap.dthr_lock = dtrace_hrestime[i].dthr_lock;
140                 dtrace_membar_consumer();
141                 snap.dthr_hrestime = dtrace_hrestime[i].dthr_hrestime;
142                 snap.dthr_hrtime = dtrace_hrestime[i].dthr_hrtime;
143                 snap.dthr_adj = dtrace_hrestime[i].dthr_adj;
144                 dtrace_membar_consumer();

146                 if ((snap.dthr_lock & ~1) == dtrace_hrestime[i].dthr_lock)
147                         break;

149                 /*
150                  * If we're here, the lock was either locked, or it
151                  * transitioned while we were taking the snapshot.  Either
152                  * way, we're going to try the other dtrace_hrestime element;
153                  * we know that it isn't possible for both to be locked
154                  * simultaneously, so we will ultimately get a good snapshot.
155                  */
156                 i ^= 1;
157         }

159         /*
160          * We have a good snapshot.  Now perform any necessary adjustments.
161          */
162         nslt = dtrace_gethrtime() - snap.dthr_hrtime;
163         ASSERT(nslt >= 0);

165         now = ((hrtime_t)snap.dthr_hrestime.tv_sec * (hrtime_t)NANOSEC) +
166             snap.dthr_hrestime.tv_nsec;

168         if (snap.dthr_adj != 0) {
169                 if (snap.dthr_adj > 0) {
170                         adj = (nslt >> adj_shift);
171                         if (adj > snap.dthr_adj)
172                                 adj = (int)snap.dthr_adj;
173                 } else {
174                         adj = -(nslt >> adj_shift);
175                         if (adj < snap.dthr_adj)
176                                 adj = (int)snap.dthr_adj;
177                 }
178                 now += adj;
179         }

181         return (now);
182 }

184 void
185 dtrace_vtime_enable(void)
186 {
187         dtrace_vtime_state_t state, nstate;

189         do {
```



```
190                 state = dtrace_vtime_active;

192                 switch (state) {
193                 case DTRACE_VTIME_INACTIVE:
194                         nstate = DTRACE_VTIME_ACTIVE;
195                         break;

197                 case DTRACE_VTIME_INACTIVE_TNF:
198                         nstate = DTRACE_VTIME_ACTIVE_TNF;
199                         break;

201                 case DTRACE_VTIME_ACTIVE:
202                 case DTRACE_VTIME_ACTIVE_TNF:
203                         panic("DTrace virtual time already enabled");
204                         /*NOTREACHED*/
205                 }

207         } while (cas32((uint32_t *)&dtrace_vtime_active,
208             state, nstate) != state);
209 }

211 void
212 dtrace_vtime_disable(void)
213 {
214         dtrace_vtime_state_t state, nstate;

216         do {
217                 state = dtrace_vtime_active;

219                 switch (state) {
220                 case DTRACE_VTIME_ACTIVE:
221                         nstate = DTRACE_VTIME_INACTIVE;
222                         break;

224                 case DTRACE_VTIME_ACTIVE_TNF:
225                         nstate = DTRACE_VTIME_INACTIVE_TNF;
226                         break;

228                 case DTRACE_VTIME_INACTIVE:
229                 case DTRACE_VTIME_INACTIVE_TNF:
230                         panic("DTrace virtual time already disabled");
231                         /*NOTREACHED*/
232                 }

234         } while (cas32((uint32_t *)&dtrace_vtime_active,
235             state, nstate) != state);
236 }

238 void
239 dtrace_vtime_enable_tnf(void)
240 {
241         dtrace_vtime_state_t state, nstate;

243         do {
244                 state = dtrace_vtime_active;

246                 switch (state) {
247                 case DTRACE_VTIME_ACTIVE:
248                         nstate = DTRACE_VTIME_ACTIVE_TNF;
249                         break;

251                 case DTRACE_VTIME_INACTIVE:
252                         nstate = DTRACE_VTIME_INACTIVE_TNF;
253                         break;

255                 case DTRACE_VTIME_ACTIVE_TNF:
```

```
256                 case DTRACE_VTIME_INACTIVE_TNF:
257                         panic("TNF already active");
258                         /*NOTREACHED*/
259                 }
260
261         } while (cas32((uint32_t *)&dtrace_vtime_active,
262             state, nstate) != state);
263 }
264
265 void
266 dtrace_vtime_disable_tnf(void)
267 {
268         dtrace_vtime_state_t state, nstate;
269
270         do {
271                 state = dtrace_vtime_active;
272
273                 switch (state) {
274                 case DTRACE_VTIME_ACTIVE_TNF:
275                         nstate = DTRACE_VTIME_ACTIVE;
276                         break;
277
278                 case DTRACE_VTIME_INACTIVE_TNF:
279                         nstate = DTRACE_VTIME_INACTIVE;
280                         break;
281
282                 case DTRACE_VTIME_ACTIVE:
283                 case DTRACE_VTIME_INACTIVE:
284                         panic("TNF already inactive");
285                         /*NOTREACHED*/
286                 }
287
288         } while (cas32((uint32_t *)&dtrace_vtime_active,
289             state, nstate) != state);
290 }
291
292 void
293 dtrace_vtime_switch(kthread_t *next)
294 {
295         dtrace_icookie_t cookie;
296         hrtime_t ts;
297
298         if (tnf_tracing_active) {
299                 tnf_thread_switch(next);
300
301                 if (dtrace_vtime_active == DTRACE_VTIME_INACTIVE_TNF)
302                         return;
303         }
304
305         cookie = dtrace_interrupt_disable();
306         ts = dtrace_gethrtime();
307
308         if (curthread->t_dtrace_start != 0) {
309                 curthread->t_dtrace_vtime += ts - curthread->t_dtrace_start;
310                 curthread->t_dtrace_start = 0;
311         }
312
313         next->t_dtrace_start = ts;
314
315         dtrace_interrupt_enable(cookie);
316 }
317
318 void (*dtrace_fasttrap_fork_ptr)(proc_t *, proc_t *);
319 void (*dtrace_fasttrap_exec_ptr)(proc_t *);
320 void (*dtrace_fasttrap_exit_ptr)(proc_t *);
```



```
322 /*
323  * This function is called by cfork() in the event that it appears that
324  * there may be dtrace tracepoints active in the parent process's address
325  * space. This first confirms the existence of dtrace tracepoints in the
326  * parent process and calls into the fasttrap module to remove the
327  * corresponding tracepoints from the child. By knowing that there are
328  * existing tracepoints, and ensuring they can't be removed, we can rely
329  * on the fasttrap module remaining loaded.
330  */
331 void
332 dtrace_fasttrap_fork(proc_t *p, proc_t *cp)
333 {
334         ASSERT(p->p_proc_flag & P_PR_LOCK);
335         ASSERT(p->p_dtrace_count > 0);
336         ASSERT(dtrace_fasttrap_fork_ptr != NULL);
337
338         dtrace_fasttrap_fork_ptr(p, cp);
339 }
```

```
**********************************************************
   46744 Tue Jan 14 16:50:04 2014
new/usr/src/uts/common/os/fio.c
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

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

/*
 * Copyright (c) 1989, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012, Joyent Inc. All rights reserved.
#endif /* ! codereview */
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*      All Rights Reserved */

#include <sys/types.h>
#include <sys/sysmacros.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/errno.h>
#include <sys/signal.h>
#include <sys/cred.h>
#include <sys/user.h>
#include <sys/conf.h>
#include <sys/vfs.h>
#include <sys/vnode.h>
#include <sys/pathname.h>
#include <sys/file.h>
#include <sys/proc.h>
#include <sys/var.h>
#include <sys/cpuvar.h>
#include <sys/open.h>
#include <sys/cmn_err.h>
#include <sys/priocntl.h>
#include <sys/procset.h>
#include <sys/prsystm.h>
#include <sys/debug.h>
#include <sys/kmem.h>
#include <sys/atomic.h>
#include <sys/fcntl.h>
#include <sys/poll.h>
#include <sys/rctl.h>
```



```
#include <sys/port_impl.h>
#include <sys/dtrace.h>
#endif /* ! codereview */

#include <c2/audit.h>
#include <sys/nbmlock.h>

#ifdef DEBUG

static uint32_t afd_maxfd;      /* # of entries in maximum allocated array */
static uint32_t afd_alloc;      /* count of kmem_alloc()s */
static uint32_t afd_free;       /* count of kmem_free()s */
static uint32_t afd_wait;       /* count of waits on non-zero ref count */
#define MAXFD(x)        (afd_maxfd = ((afd_maxfd >= (x))? afd_maxfd : (x)))
#define COUNT(x)        atomic_add_32(&x, 1)

#else   /* DEBUG */

#define MAXFD(x)
#define COUNT(x)

#endif  /* DEBUG */

kmem_cache_t *file_cache;

static void port_close_fd(portfd_t *);

/*
 * File descriptor allocation.
 *
 * fd_find(fip, minfd) finds the first available descriptor >= minfd.
 * The most common case is open(2), in which minfd = 0, but we must also
 * support fcntl(fd, F_DUPFD, minfd).
 *
 * The algorithm is as follows: we keep all file descriptors in an infix
 * binary tree in which each node records the number of descriptors
 * allocated in its right subtree, including itself.  Starting at minfd,
 * we ascend the tree until we find a non-fully allocated right subtree.
 * We then descend that subtree in a binary search for the smallest fd.
 * Finally, we ascend the tree again to increment the allocation count
 * of every subtree containing the newly-allocated fd.  Freeing an fd
 * requires only the last step: we ascend the tree to decrement allocation
 * counts.  Each of these three steps (ascent to find non-full subtree,
 * descent to find lowest fd, ascent to update allocation counts) is
 * O(log n), thus the algorithm as a whole is O(log n).
 *
 * We don't implement the fd tree using the customary left/right/parent
 * pointers, but instead take advantage of the glorious mathematics of
 * full infix binary trees.  For reference, here's an illustration of the
 * logical structure of such a tree, rooted at 4 (binary 100), covering
 * the range 1-7 (binary 001-111).  Our canonical trees do not include
 * fd 0; we'll deal with that later.
 *
 *            100
 *           /   \
 *          /     \
 *        010     110
 *        / \     / \
 *      001 011 101 111
 *
 * We make the following observations, all of which are easily proven by
 * induction on the depth of the tree:
 *
 * (T1) The least-significant bit (LSB) of any node is equal to its level
 *      in the tree.  In our example, nodes 001, 011, 101 and 111 are at
 *      level 0; nodes 010 and 110 are at level 1; and node 100 is at level 2.
```

```
 *
 * (T2) The child size (CSIZE) of node N -- that is, the total number of
 *      right-branch descendants in a child of node N, including itself -- is
 *      given by clearing all but the least significant bit of N.  This
 *      follows immediately from (T1).  Applying this rule to our example, we
 *      see that CSIZE(100) = 100, CSIZE(x10) = 10, and CSIZE(xx1) = 1.
 *
 * (T3) The nearest left ancestor (LPARENT) of node N -- that is, the nearest
 *      ancestor containing node N in its right child -- is given by clearing
 *      the LSB of N.  For example, LPARENT(111) = 110 and LPARENT(110) = 100.
 *      Clearing the LSB of nodes 001, 010 or 100 yields zero, reflecting
 *      the fact that these are leftmost nodes.  Note that this algorithm
 *      automatically skips generations as necessary.  For example, the parent
 *      of node 101 is 110, which is a *right* ancestor (not what we want);
 *      but its grandparent is 100, which is a left ancestor. Clearing the LSB
 *      of 101 gets us to 100 directly, skipping right past the uninteresting
 *      generation (110).
 *
 *      Note that since LPARENT clears the LSB, whereas CSIZE clears all *but*
 *      the LSB, we can express LPARENT() nicely in terms of CSIZE():
 *
 *      LPARENT(N) = N - CSIZE(N)
 *
 * (T4) The nearest right ancestor (RPARENT) of node N is given by:
 *
 *      RPARENT(N) = N + CSIZE(N)
 *
 * (T5) For every interior node, the children differ from their parent by
 *      CSIZE(parent) / 2.  In our example, CSIZE(100) / 2 = 2 = 10 binary,
 *      and indeed, the children of 100 are 100 +/- 10 = 010 and 110.
 *
 * Next, we'll need a few two's-complement math tricks.  Suppose a number,
 * N, has the following form:
 *
 *              N = xxxx10...0
 *
 * That is, the binary representation of N consists of some string of bits,
 * then a 1, then all zeroes.  This amounts to nothing more than saying that
 * N has a least-significant bit, which is true for any N != 0.  If we look
 * at N and N - 1 together, we see that we can combine them in useful ways:
 *
 *                N = xxxx10...0
 *            N - 1 = xxxx01...1
 *      ------------------------
 *      N & (N - 1) = xxxx000000
 *      N | (N - 1) = xxxx111111
 *      N ^ (N - 1) =     111111
 *
 * In particular, this suggests several easy ways to clear all but the LSB,
 * which by (T2) is exactly what we need to determine CSIZE(N) = 10...0.
 * We'll opt for this formulation:
 *
 *      (C1) CSIZE(N) = (N - 1) ^ (N | (N - 1))
 *
 * Similarly, we have an easy way to determine LPARENT(N), which requires
 * that we clear the LSB of N:
 *
 *      (L1) LPARENT(N) = N & (N - 1)
 *
 * We note in the above relations that (N | (N - 1)) - N = CSIZE(N) - 1.
 * When combined with (T4), this yields an easy way to compute RPARENT(N):
 *
 *      (R1) RPARENT(N) = (N | (N - 1)) + 1
 *
 * Finally, to accommodate fd 0 we must adjust all of our results by +/-1 to
 * move the fd range from [1, 2^n) to [0, 2^n - 1).  This is straightforward,
```



```
 * so there's no need to belabor the algebra; the revised relations become:
 *
 *      (C1a) CSIZE(N) = N ^ (N | (N + 1))
 *
 *      (L1a) LPARENT(N) = (N & (N + 1)) - 1
 *
 *      (R1a) RPARENT(N) = N | (N + 1)
 *
 * This completes the mathematical framework.  We now have all the tools
 * we need to implement fd_find() and fd_reserve().
 *
 * fd_find(fip, minfd) finds the smallest available file descriptor >= minfd.
 * It does not actually allocate the descriptor; that's done by fd_reserve().
 * fd_find() proceeds in two steps:
 *
 * (1) Find the leftmost subtree that contains a descriptor >= minfd.
 *     We start at the right subtree rooted at minfd.  If this subtree is
 *     not full -- if fip->fi_list[minfd].uf_alloc != CSIZE(minfd) -- then
 *     step 1 is done.  Otherwise, we know that all fds in this subtree
 *     are taken, so we ascend to RPARENT(minfd) using (R1a).  We repeat
 *     this process until we either find a candidate subtree or exceed
 *     fip->fi_nfiles.  We use (C1a) to compute CSIZE().
 *
 * (2) Find the smallest fd in the subtree discovered by step 1.
 *     Starting at the root of this subtree, we descend to find the
 *     smallest available fd.  Since the left children have the smaller
 *     fds, we will descend rightward only when the left child is full.
 *
 *     We begin by comparing the number of allocated fds in the root
 *     to the number of allocated fds in its right child; if they differ
 *     by exactly CSIZE(child), we know the left subtree is full, so we
 *     descend right; that is, the right child becomes the search root.
 *     Otherwise we leave the root alone and start following the right
 *     child's left children.  As fortune would have it, this is very
 *     simple computationally: by (T5), the right child of fd is just
 *     fd + size, where size = CSIZE(fd) / 2.  Applying (T5) again,
 *     we find that the right child's left child is fd + size - (size / 2) =
 *     fd + (size / 2); *its* left child is fd + (size / 2) - (size / 4) =
 *     fd + (size / 4), and so on.  In general, fd's right child's
 *     leftmost nth descendant is fd + (size >> n).  Thus, to follow
 *     the right child's left descendants, we just halve the size in
 *     each iteration of the search.
 *
 *     When we descend leftward, we must keep track of the number of fds
 *     that were allocated in all the right subtrees we rejected, so we
 *     know how many of the root fd's allocations are in the remaining
 *     (as yet unexplored) leftmost part of its right subtree.  When we
 *     encounter a fully-allocated left child -- that is, when we find
 *     that fip->fi_list[fd].uf_alloc == ralloc + size -- we descend right
 *     (as described earlier), resetting ralloc to zero.
 *
 * fd_reserve(fip, fd, incr) either allocates or frees fd, depending
 * on whether incr is 1 or -1.  Starting at fd, fd_reserve() ascends
 * the leftmost ancestors (see (T3)) and updates the allocation counts.
 * At each step we use (L1a) to compute LPARENT(), the next left ancestor.
 *
 * flist_minsize() finds the minimal tree that still covers all
 * used fds; as long as the allocation count of a root node is zero, we
 * don't need that node or its right subtree.
 *
 * flist_nalloc() counts the number of allocated fds in the tree, by starting
 * at the top of the tree and summing the right-subtree allocation counts as
 * it descends leftwards.
 *
 * Note: we assume that flist_grow() will keep fip->fi_nfiles of the form
 * 2^n - 1.  This ensures that the fd trees are always full, which saves
```

```
 * quite a bit of boundary checking.
 */
static int
fd_find(uf_info_t *fip, int minfd)
{
        int size, ralloc, fd;

        ASSERT(MUTEX_HELD(&fip->fi_lock));
        ASSERT((fip->fi_nfiles & (fip->fi_nfiles + 1)) == 0);

        for (fd = minfd; (uint_t)fd < fip->fi_nfiles; fd |= fd + 1) {
                size = fd ^ (fd | (fd + 1));
                if (fip->fi_list[fd].uf_alloc == size)
                        continue;
                for (ralloc = 0, size >>= 1; size != 0; size >>= 1) {
                        ralloc += fip->fi_list[fd + size].uf_alloc;
                        if (fip->fi_list[fd].uf_alloc == ralloc + size) {
                                fd += size;
                                ralloc = 0;
                        }
                }
                return (fd);
        }
        return (-1);
}

static void
fd_reserve(uf_info_t *fip, int fd, int incr)
{
        int pfd;
        uf_entry_t *ufp = &fip->fi_list[fd];

        ASSERT((uint_t)fd < fip->fi_nfiles);
        ASSERT((ufp->uf_busy == 0 && incr == 1) ||
            (ufp->uf_busy == 1 && incr == -1));
        ASSERT(MUTEX_HELD(&ufp->uf_lock));
        ASSERT(MUTEX_HELD(&fip->fi_lock));

        for (pfd = fd; pfd >= 0; pfd = (pfd & (pfd + 1)) - 1)
                fip->fi_list[pfd].uf_alloc += incr;

        ufp->uf_busy += incr;
}

static int
flist_minsize(uf_info_t *fip)
{
        int fd;

        /*
         * We'd like to ASSERT(MUTEX_HELD(&fip->fi_lock)), but we're called
         * by flist_fork(), which relies on other mechanisms for mutual
         * exclusion.
         */
        ASSERT((fip->fi_nfiles & (fip->fi_nfiles + 1)) == 0);

        for (fd = fip->fi_nfiles; fd != 0; fd >>= 1)
                if (fip->fi_list[fd >> 1].uf_alloc != 0)
                        break;

        return (fd);
}

static int
flist_nalloc(uf_info_t *fip)
{
```



```
        int fd;
        int nalloc = 0;

        ASSERT(MUTEX_HELD(&fip->fi_lock));
        ASSERT((fip->fi_nfiles & (fip->fi_nfiles + 1)) == 0);

        for (fd = fip->fi_nfiles; fd != 0; fd >>= 1)
                nalloc += fip->fi_list[fd >> 1].uf_alloc;

        return (nalloc);
}

/*
 * Increase size of the fi_list array to accommodate at least maxfd.
 * We keep the size of the form 2^n - 1 for benefit of fd_find().
 */
static void
flist_grow(int maxfd)
{
        uf_info_t *fip = P_FINFO(curproc);
        int newcnt, oldcnt;
        uf_entry_t *src, *dst, *newlist, *oldlist, *newend, *oldend;
        uf_rlist_t *urp;

        for (newcnt = 1; newcnt <= maxfd; newcnt = (newcnt << 1) | 1)
                continue;

        newlist = kmem_zalloc(newcnt * sizeof (uf_entry_t), KM_SLEEP);

        mutex_enter(&fip->fi_lock);
        oldcnt = fip->fi_nfiles;
        if (newcnt <= oldcnt) {
                mutex_exit(&fip->fi_lock);
                kmem_free(newlist, newcnt * sizeof (uf_entry_t));
                return;
        }
        ASSERT((newcnt & (newcnt + 1)) == 0);
        oldlist = fip->fi_list;
        oldend = oldlist + oldcnt;
        newend = newlist + oldcnt;      /* no need to lock beyond old end */

        /*
         * fi_list and fi_nfiles cannot change while any uf_lock is held,
         * so we must grab all the old locks *and* the new locks up to oldcnt.
         * (Locks beyond the end of oldcnt aren't visible until we store
         * the new fi_nfiles, which is the last thing we do before dropping
         * all the locks, so there's no need to acquire these locks).
         * Holding the new locks is necessary because when fi_list changes
         * to point to the new list, fi_nfiles won't have been stored yet.
         * If we *didn't* hold the new locks, someone doing a UF_ENTER()
         * could see the new fi_list, grab the new uf_lock, and then see
         * fi_nfiles change while the lock is held -- in violation of
         * UF_ENTER() semantics.
         */
        for (src = oldlist; src < oldend; src++)
                mutex_enter(&src->uf_lock);

        for (dst = newlist; dst < newend; dst++)
                mutex_enter(&dst->uf_lock);

        for (src = oldlist, dst = newlist; src < oldend; src++, dst++) {
                dst->uf_file = src->uf_file;
                dst->uf_fpollinfo = src->uf_fpollinfo;
                dst->uf_refcnt = src->uf_refcnt;
                dst->uf_alloc = src->uf_alloc;
                dst->uf_flag = src->uf_flag;
```

```
388                 dst->uf_busy = src->uf_busy;
389                 dst->uf_portfd = src->uf_portfd;
390         }

392         /*
393          * As soon as we store the new flist, future locking operations
394          * will use it.  Therefore, we must ensure that all the state
395          * we've just established reaches global visibility before the
396          * new flist does.
397          */
398         membar_producer();
399         fip->fi_list = newlist;

401         /*
402          * Routines like getf() make an optimistic check on the validity
403          * of the supplied file descriptor: if it's less than the current
404          * value of fi_nfiles -- examined without any locks -- then it's
405          * safe to attempt a UF_ENTER() on that fd (which is a valid
406          * assumption because fi_nfiles only increases).  Therefore, it
407          * is critical that the new value of fi_nfiles not reach global
408          * visibility until after the new fi_list: if it happened the
409          * other way around, getf() could see the new fi_nfiles and attempt
410          * a UF_ENTER() on the old fi_list, which would write beyond its
411          * end if the fd exceeded the old fi_nfiles.
412          */
413         membar_producer();
414         fip->fi_nfiles = newcnt;

416         /*
417          * The new state is consistent now, so we can drop all the locks.
418          */
419         for (dst = newlist; dst < newend; dst++)
420                 mutex_exit(&dst->uf_lock);

422         for (src = oldlist; src < oldend; src++) {
423                 /*
424                  * If any threads are blocked on the old cvs, wake them.
425                  * This will force them to wake up, discover that fi_list
426                  * has changed, and go back to sleep on the new cvs.
427                  */
428                 cv_broadcast(&src->uf_wanted_cv);
429                 cv_broadcast(&src->uf_closing_cv);
430                 mutex_exit(&src->uf_lock);
431         }

433         mutex_exit(&fip->fi_lock);

435         /*
436          * Retire the old flist.  We can't actually kmem_free() it now
437          * because someone may still have a pointer to it.  Instead,
438          * we link it onto a list of retired flists.  The new flist
439          * is at least double the size of the previous flist, so the
440          * total size of all retired flists will be less than the size
441          * of the current one (to prove, consider the sum of a geometric
442          * series in powers of 2).  exit() frees the retired flists.
443          */
444         urp = kmem_zalloc(sizeof (uf_rlist_t), KM_SLEEP);
445         urp->ur_list = oldlist;
446         urp->ur_nfiles = oldcnt;

448         mutex_enter(&fip->fi_lock);
449         urp->ur_next = fip->fi_rlist;
450         fip->fi_rlist = urp;
451         mutex_exit(&fip->fi_lock);
452 }
```



```
454 /*
455  * Utility functions for keeping track of the active file descriptors.
456  */
457 void
458 clear_stale_fd()                /* called from post_syscall() */
459 {
460         afd_t *afd = &curthread->t_activefd;
461         int i;

463         /* uninitialized is ok here, a_nfd is then zero */
464         for (i = 0; i < afd->a_nfd; i++) {
465                 /* assert that this should not be necessary */
466                 ASSERT(afd->a_fd[i] == -1);
467                 afd->a_fd[i] = -1;
468         }
469         afd->a_stale = 0;
470 }

472 void
473 free_afd(afd_t *afd)            /* called below and from thread_free() */
474 {
475         int i;

477         /* free the buffer if it was kmem_alloc()ed */
478         if (afd->a_nfd > sizeof (afd->a_buf) / sizeof (afd->a_buf[0])) {
479                 COUNT(afd_free);
480                 kmem_free(afd->a_fd, afd->a_nfd * sizeof (afd->a_fd[0]));
481         }

483         /* (re)initialize the structure */
484         afd->a_fd = &afd->a_buf[0];
485         afd->a_nfd = sizeof (afd->a_buf) / sizeof (afd->a_buf[0]);
486         afd->a_stale = 0;
487         for (i = 0; i < afd->a_nfd; i++)
488                 afd->a_fd[i] = -1;
489 }

491 static void
492 set_active_fd(int fd)
493 {
494         afd_t *afd = &curthread->t_activefd;
495         int i;
496         int *old_fd;
497         int old_nfd;
498         int *new_fd;
499         int new_nfd;

501         if (afd->a_nfd == 0) {  /* first time initialization */
502                 ASSERT(fd == -1);
503                 mutex_enter(&afd->a_fdlock);
504                 free_afd(afd);
505                 mutex_exit(&afd->a_fdlock);
506         }

508         /* insert fd into vacant slot, if any */
509         for (i = 0; i < afd->a_nfd; i++) {
510                 if (afd->a_fd[i] == -1) {
511                         afd->a_fd[i] = fd;
512                         return;
513                 }
514         }

516         /*
517          * Reallocate the a_fd[] array to add one more slot.
518          */
519         ASSERT(fd == -1);
```

```
520         old_nfd = afd->a_nfd;
521         old_fd = afd->a_fd;
522         new_nfd = old_nfd + 1;
523         new_fd = kmem_alloc(new_nfd * sizeof (afd->a_fd[0]), KM_SLEEP);
524         MAXFD(new_nfd);
525         COUNT(afd_alloc);

527         mutex_enter(&afd->a_fdlock);
528         afd->a_fd = new_fd;
529         afd->a_nfd = new_nfd;
530         for (i = 0; i < old_nfd; i++)
531                 afd->a_fd[i] = old_fd[i];
532         afd->a_fd[i] = fd;
533         mutex_exit(&afd->a_fdlock);

535         if (old_nfd > sizeof (afd->a_buf) / sizeof (afd->a_buf[0])) {
536                 COUNT(afd_free);
537                 kmem_free(old_fd, old_nfd * sizeof (afd->a_fd[0]));
538         }
539 }

541 void
542 clear_active_fd(int fd)         /* called below and from aio.c */
543 {
544         afd_t *afd = &curthread->t_activefd;
545         int i;

547         for (i = 0; i < afd->a_nfd; i++) {
548                 if (afd->a_fd[i] == fd) {
549                         afd->a_fd[i] = -1;
550                         break;
551                 }
552         }
553         ASSERT(i < afd->a_nfd);         /* not found is not ok */
554 }

556 /*
557  * Does this thread have this fd active?
558  */
559 static int
560 is_active_fd(kthread_t *t, int fd)
561 {
562         afd_t *afd = &t->t_activefd;
563         int i;

565         ASSERT(t != curthread);
566         mutex_enter(&afd->a_fdlock);
567         /* uninitialized is ok here, a_nfd is then zero */
568         for (i = 0; i < afd->a_nfd; i++) {
569                 if (afd->a_fd[i] == fd) {
570                         mutex_exit(&afd->a_fdlock);
571                         return (1);
572                 }
573         }
574         mutex_exit(&afd->a_fdlock);
575         return (0);
576 }

578 /*
579  * Convert a user supplied file descriptor into a pointer to a file
580  * structure.  Only task is to check range of the descriptor (soft
581  * resource limit was enforced at open time and shouldn't be checked
582  * here).
583  */
584 file_t *
585 getf(int fd)
```



```
586 {
587         uf_info_t *fip = P_FINFO(curproc);
588         uf_entry_t *ufp;
589         file_t *fp;

591         if ((uint_t)fd >= fip->fi_nfiles)
592                 return (NULL);

594         /*
595          * Reserve a slot in the active fd array now so we can call
596          * set_active_fd(fd) for real below, while still inside UF_ENTER().
597          */
598         set_active_fd(-1);

600         UF_ENTER(ufp, fip, fd);

602         if ((fp = ufp->uf_file) == NULL) {
603                 UF_EXIT(ufp);

605                 if (fd == fip->fi_badfd && fip->fi_action > 0)
606                         tsignal(curthread, fip->fi_action);

608                 return (NULL);
609         }
610         ufp->uf_refcnt++;

612         set_active_fd(fd);      /* record the active file descriptor */

614         UF_EXIT(ufp);

616         return (fp);
617 }

619 /*
620  * Close whatever file currently occupies the file descriptor slot
621  * and install the new file, usually NULL, in the file descriptor slot.
622  * The close must complete before we release the file descriptor slot.
623  * If newfp != NULL we only return an error if we can't allocate the
624  * slot so the caller knows that it needs to free the filep;
625  * in the other cases we return the error number from closef().
626  */
627 int
628 closeandsetf(int fd, file_t *newfp)
629 {
630         proc_t *p = curproc;
631         uf_info_t *fip = P_FINFO(p);
632         uf_entry_t *ufp;
633         file_t *fp;
634         fpollinfo_t *fpip;
635         portfd_t *pfd;
636         int error;

638         if ((uint_t)fd >= fip->fi_nfiles) {
639                 if (newfp == NULL)
640                         return (EBADF);
641                 flist_grow(fd);
642         }

644         if (newfp != NULL) {
645                 /*
646                  * If ufp is reserved but has no file pointer, it's in the
647                  * transition between ufalloc() and setf().  We must wait
648                  * for this transition to complete before assigning the
649                  * new non-NULL file pointer.
650                  */
651                 mutex_enter(&fip->fi_lock);
```

```
652                 if (fd == fip->fi_badfd) {
653                         mutex_exit(&fip->fi_lock);
654                         if (fip->fi_action > 0)
655                                 tsignal(curthread, fip->fi_action);
656                         return (EBADF);
657                 }
658                 UF_ENTER(ufp, fip, fd);
659                 while (ufp->uf_busy && ufp->uf_file == NULL) {
660                         mutex_exit(&fip->fi_lock);
661                         cv_wait_stop(&ufp->uf_wanted_cv, &ufp->uf_lock, 250);
662                         UF_EXIT(ufp);
663                         mutex_enter(&fip->fi_lock);
664                         UF_ENTER(ufp, fip, fd);
665                 }
666                 if ((fp = ufp->uf_file) == NULL) {
667                         ASSERT(ufp->uf_fpollinfo == NULL);
668                         ASSERT(ufp->uf_flag == 0);
669                         fd_reserve(fip, fd, 1);
670                         ufp->uf_file = newfp;
671                         UF_EXIT(ufp);
672                         mutex_exit(&fip->fi_lock);
673                         return (0);
674                 }
675                 mutex_exit(&fip->fi_lock);
676         } else {
677                 UF_ENTER(ufp, fip, fd);
678                 if ((fp = ufp->uf_file) == NULL) {
679                         UF_EXIT(ufp);
680                         return (EBADF);
681                 }
682         }

684         ASSERT(ufp->uf_busy);
685         ufp->uf_file = NULL;
686         ufp->uf_flag = 0;

688         /*
689          * If the file descriptor reference count is non-zero, then
690          * some other lwp in the process is performing system call
691          * activity on the file.  To avoid blocking here for a long
692          * time (the other lwp might be in a long term sleep in its
693          * system call), we scan all other lwps in the process to
694          * find the ones with this fd as one of their active fds,
695          * set their a_stale flag, and set them running if they
696          * are in an interruptible sleep so they will emerge from
697          * their system calls immediately.  post_syscall() will
698          * test the a_stale flag and set errno to EBADF.
699          */
700         ASSERT(ufp->uf_refcnt == 0 || p->p_lwpcnt > 1);
701         if (ufp->uf_refcnt > 0) {
702                 kthread_t *t;

704                 /*
705                  * We call sprlock_proc(p) to ensure that the thread
706                  * list will not change while we are scanning it.
707                  * To do this, we must drop ufp->uf_lock and then
708                  * reacquire it (so we are not holding both p->p_lock
709                  * and ufp->uf_lock at the same time).  ufp->uf_lock
710                  * must be held for is_active_fd() to be correct
711                  * (set_active_fd() is called while holding ufp->uf_lock).
712                  *
713                  * This is a convoluted dance, but it is better than
714                  * the old brute-force method of stopping every thread
715                  * in the process by calling holdlwps(SHOLDFORK1).
716                  */
```



```
718                 UF_EXIT(ufp);
719                 COUNT(afd_wait);

721                 mutex_enter(&p->p_lock);
722                 sprlock_proc(p);
723                 mutex_exit(&p->p_lock);

725                 UF_ENTER(ufp, fip, fd);
726                 ASSERT(ufp->uf_file == NULL);

728                 if (ufp->uf_refcnt > 0) {
729                         for (t = curthread->t_forw;
730                             t != curthread;
731                             t = t->t_forw) {
732                                 if (is_active_fd(t, fd)) {
733                                         thread_lock(t);
734                                         t->t_activefd.a_stale = 1;
735                                         t->t_post_sys = 1;
736                                         if (ISWAKEABLE(t))
737                                                 setrun_locked(t);
738                                         thread_unlock(t);
739                                 }
740                         }
741                 }

743                 UF_EXIT(ufp);

745                 mutex_enter(&p->p_lock);
746                 sprunlock(p);

748                 UF_ENTER(ufp, fip, fd);
749                 ASSERT(ufp->uf_file == NULL);
750         }

752         /*
753          * Wait for other lwps to stop using this file descriptor.
754          */
755         while (ufp->uf_refcnt > 0) {
756                 cv_wait_stop(&ufp->uf_closing_cv, &ufp->uf_lock, 250);
757                 /*
758                  * cv_wait_stop() drops ufp->uf_lock, so the file list
759                  * can change.  Drop the lock on our (possibly) stale
760                  * ufp and let UF_ENTER() find and lock the current ufp.
761                  */
762                 UF_EXIT(ufp);
763                 UF_ENTER(ufp, fip, fd);
764         }

766 #ifdef DEBUG
767         /*
768          * catch a watchfd on device's pollhead list but not on fpollinfo list
769          */
770         if (ufp->uf_fpollinfo != NULL)
771                 checkwfdlist(fp->f_vnode, ufp->uf_fpollinfo);
772 #endif  /* DEBUG */

774         /*
775          * We may need to cleanup some cached poll states in t_pollstate
776          * before the fd can be reused. It is important that we don't
777          * access a stale thread structure. We will do the cleanup in two
778          * phases to avoid deadlock and holding uf_lock for too long.
779          * In phase 1, hold the uf_lock and call pollblockexit() to set
780          * state in t_pollstate struct so that a thread does not exit on
781          * us. In phase 2, we drop the uf_lock and call pollcacheclean().
782          */
783         pfd = ufp->uf_portfd;
```

```
784         ufp->uf_portfd = NULL;
785         fpip = ufp->uf_fpollinfo;
786         ufp->uf_fpollinfo = NULL;
787         if (fpip != NULL)
788                 pollblockexit(fpip);
789         UF_EXIT(ufp);
790         if (fpip != NULL)
791                 pollcacheclean(fpip, fd);
792         if (pfd)
793                 port_close_fd(pfd);

795         /*
796          * Keep the file descriptor entry reserved across the closef().
797          */
798         error = closef(fp);

800         setf(fd, newfp);

802         /* Only return closef() error when closing is all we do */
803         return (newfp == NULL ? error : 0);
804 }

806 /*
807  * Decrement uf_refcnt; wakeup anyone waiting to close the file.
808  */
809 void
810 releasef(int fd)
811 {
812         uf_info_t *fip = P_FINFO(curproc);
813         uf_entry_t *ufp;

815         UF_ENTER(ufp, fip, fd);
816         ASSERT(ufp->uf_refcnt > 0);
817         clear_active_fd(fd);    /* clear the active file descriptor */
818         if (--ufp->uf_refcnt == 0)
819                 cv_broadcast(&ufp->uf_closing_cv);
820         UF_EXIT(ufp);
821 }

823 /*
824  * Identical to releasef() but can be called from another process.
825  */
826 void
827 areleasef(int fd, uf_info_t *fip)
828 {
829         uf_entry_t *ufp;

831         UF_ENTER(ufp, fip, fd);
832         ASSERT(ufp->uf_refcnt > 0);
833         if (--ufp->uf_refcnt == 0)
834                 cv_broadcast(&ufp->uf_closing_cv);
835         UF_EXIT(ufp);
836 }

838 /*
839  * Duplicate all file descriptors across a fork.
840  */
841 void
842 flist_fork(uf_info_t *pfip, uf_info_t *cfip)
843 {
844         int fd, nfiles;
845         uf_entry_t *pufp, *cufp;

847         mutex_init(&cfip->fi_lock, NULL, MUTEX_DEFAULT, NULL);
848         cfip->fi_rlist = NULL;
```



```
850         /*
851          * We don't need to hold fi_lock because all other lwp's in the
852          * parent have been held.
853          */
854         cfip->fi_nfiles = nfiles = flist_minsize(pfip);

856         cfip->fi_list = kmem_zalloc(nfiles * sizeof (uf_entry_t), KM_SLEEP);

858         for (fd = 0, pufp = pfip->fi_list, cufp = cfip->fi_list; fd < nfiles;
859             fd++, pufp++, cufp++) {
860                 cufp->uf_file = pufp->uf_file;
861                 cufp->uf_alloc = pufp->uf_alloc;
862                 cufp->uf_flag = pufp->uf_flag;
863                 cufp->uf_busy = pufp->uf_busy;
864                 if (pufp->uf_file == NULL) {
865                         ASSERT(pufp->uf_flag == 0);
866                         if (pufp->uf_busy) {
867                                 /*
868                                  * Grab locks to appease ASSERTs in fd_reserve
869                                  */
870                                 mutex_enter(&cfip->fi_lock);
871                                 mutex_enter(&cufp->uf_lock);
872                                 fd_reserve(cfip, fd, -1);
873                                 mutex_exit(&cufp->uf_lock);
874                                 mutex_exit(&cfip->fi_lock);
875                         }
876                 }
877         }
878 }

880 /*
881  * Close all open file descriptors for the current process.
882  * This is only called from exit(), which is single-threaded,
883  * so we don't need any locking.
884  */
885 void
886 closeall(uf_info_t *fip)
887 {
888         int fd;
889         file_t *fp;
890         uf_entry_t *ufp;

892         ufp = fip->fi_list;
893         for (fd = 0; fd < fip->fi_nfiles; fd++, ufp++) {
894                 if ((fp = ufp->uf_file) != NULL) {
895                         ufp->uf_file = NULL;
896                         if (ufp->uf_portfd != NULL) {
897                                 portfd_t *pfd;
898                                 /* remove event port association */
899                                 pfd = ufp->uf_portfd;
900                                 ufp->uf_portfd = NULL;
901                                 port_close_fd(pfd);
902                         }
903                         ASSERT(ufp->uf_fpollinfo == NULL);
904                         (void) closef(fp);
905                 }
906         }

908         kmem_free(fip->fi_list, fip->fi_nfiles * sizeof (uf_entry_t));
909         fip->fi_list = NULL;
910         fip->fi_nfiles = 0;
911         while (fip->fi_rlist != NULL) {
912                 uf_rlist_t *urp = fip->fi_rlist;
913                 fip->fi_rlist = urp->ur_next;
914                 kmem_free(urp->ur_list, urp->ur_nfiles * sizeof (uf_entry_t));
915                 kmem_free(urp, sizeof (uf_rlist_t));
```

```
916         }
917 }

919 /*
920  * Internal form of close.  Decrement reference count on file
921  * structure.  Decrement reference count on the vnode following
922  * removal of the referencing file structure.
923  */
924 int
925 closef(file_t *fp)
926 {
927         vnode_t *vp;
928         int error;
929         int count;
930         int flag;
931         offset_t offset;

933         /*
934          * audit close of file (may be exit)
935          */
936         if (AU_AUDITING())
937                 audit_closef(fp);
938         ASSERT(MUTEX_NOT_HELD(&P_FINFO(curproc)->fi_lock));

940         mutex_enter(&fp->f_tlock);

942         ASSERT(fp->f_count > 0);

944         count = fp->f_count--;
945         flag = fp->f_flag;
946         offset = fp->f_offset;

948         vp = fp->f_vnode;

950         error = VOP_CLOSE(vp, flag, count, offset, fp->f_cred, NULL);

952         if (count > 1) {
953                 mutex_exit(&fp->f_tlock);
954                 return (error);
955         }
956         ASSERT(fp->f_count == 0);
957         mutex_exit(&fp->f_tlock);

959         /*
960          * If DTrace has getf() subroutines active, it will set dtrace_closef
961          * to point to code that implements a barrier with respect to probe
962          * context.  This must be called before the file_t is freed (and the
963          * vnode that it refers to is released) -- but it must be after the
964          * file_t has been removed from the uf_entry_t.  That is, there must
965          * be no way for a racing getf() in probe context to yield the fp that
966          * we're operating upon.
967          */
968         if (dtrace_closef != NULL)
969                 (*dtrace_closef)();

971 #endif /* ! codereview */
972         VN_RELE(vp);
973         /*
974          * deallocate resources to audit_data
975          */
976         if (audit_active)
977                 audit_unfalloc(fp);
978         crfree(fp->f_cred);
979         kmem_cache_free(file_cache, fp);
980         return (error);
981 }
```



```
983 /*
984  * This is a combination of ufalloc() and setf().
985  */
986 int
987 ufalloc_file(int start, file_t *fp)
988 {
989         proc_t *p = curproc;
990         uf_info_t *fip = P_FINFO(p);
991         int filelimit;
992         uf_entry_t *ufp;
993         int nfiles;
994         int fd;

996         /*
997          * Assertion is to convince the correctness of the following
998          * assignment for filelimit after casting to int.
999          */
1000        ASSERT(p->p_fno_ctl <= INT_MAX);
1001        filelimit = (int)p->p_fno_ctl;

1003        for (;;) {
1004                mutex_enter(&fip->fi_lock);
1005                fd = fd_find(fip, start);
1006                if (fd >= 0 && fd == fip->fi_badfd) {
1007                        start = fd + 1;
1008                        mutex_exit(&fip->fi_lock);
1009                        continue;
1010                }
1011                if ((uint_t)fd < filelimit)
1012                        break;
1013                if (fd >= filelimit) {
1014                        mutex_exit(&fip->fi_lock);
1015                        mutex_enter(&p->p_lock);
1016                        (void) rctl_action(rctlproc_legacy[RLIMIT_NOFILE],
1017                            p->p_rctls, p, RCA_SAFE);
1018                        mutex_exit(&p->p_lock);
1019                        return (-1);
1020                }
1021                /* fd_find() returned -1 */
1022                nfiles = fip->fi_nfiles;
1023                mutex_exit(&fip->fi_lock);
1024                flist_grow(MAX(start, nfiles));
1025        }

1027        UF_ENTER(ufp, fip, fd);
1028        fd_reserve(fip, fd, 1);
1029        ASSERT(ufp->uf_file == NULL);
1030        ufp->uf_file = fp;
1031        UF_EXIT(ufp);
1032        mutex_exit(&fip->fi_lock);
1033        return (fd);
1034 }

1036 /*
1037  * Allocate a user file descriptor greater than or equal to "start".
1038  */
1039 int
1040 ufalloc(int start)
1041 {
1042         return (ufalloc_file(start, NULL));
1043 }

1045 /*
1046  * Check that a future allocation of count fds on proc p has a good
1047  * chance of succeeding.  If not, do rctl processing as if we'd failed
```

```
1048  * the allocation.
1049  *
1050  * Our caller must guarantee that p cannot disappear underneath us.
1051  */
1052 int
1053 ufcanalloc(proc_t *p, uint_t count)
1054 {
1055         uf_info_t *fip = P_FINFO(p);
1056         int filelimit;
1057         int current;

1059         if (count == 0)
1060                 return (1);

1062         ASSERT(p->p_fno_ctl <= INT_MAX);
1063         filelimit = (int)p->p_fno_ctl;

1065         mutex_enter(&fip->fi_lock);
1066         current = flist_nalloc(fip);            /* # of in-use descriptors */
1067         mutex_exit(&fip->fi_lock);

1069         /*
1070          * If count is a positive integer, the worst that can happen is
1071          * an overflow to a negative value, which is caught by the >= 0 check.
1072          */
1073         current += count;
1074         if (count <= INT_MAX && current >= 0 && current <= filelimit)
1075                 return (1);

1077         mutex_enter(&p->p_lock);
1078         (void) rctl_action(rctlproc_legacy[RLIMIT_NOFILE],
1079             p->p_rctls, p, RCA_SAFE);
1080         mutex_exit(&p->p_lock);
1081         return (0);
1082 }

1084 /*
1085  * Allocate a user file descriptor and a file structure.
1086  * Initialize the descriptor to point at the file structure.
1087  * If fdp is NULL, the user file descriptor will not be allocated.
1088  */
1089 int
1090 falloc(vnode_t *vp, int flag, file_t **fpp, int *fdp)
1091 {
1092         file_t *fp;
1093         int fd;

1095         if (fdp) {
1096                 if ((fd = ufalloc(0)) == -1)
1097                         return (EMFILE);
1098         }
1099         fp = kmem_cache_alloc(file_cache, KM_SLEEP);
1100         /*
1101          * Note: falloc returns the fp locked
1102          */
1103         mutex_enter(&fp->f_tlock);
1104         fp->f_count = 1;
1105         fp->f_flag = (ushort_t)flag;
1106         fp->f_flag2 = (flag & (FSEARCH|FEXEC)) >> 16;
1107         fp->f_vnode = vp;
1108         fp->f_offset = 0;
1109         fp->f_audit_data = 0;
1110         crhold(fp->f_cred = CRED());
1111         /*
1112          * allocate resources to audit_data
1113          */
```



```
1114         if (audit_active)
1115                 audit_falloc(fp);
1116         *fpp = fp;
1117         if (fdp)
1118                 *fdp = fd;
1119         return (0);
1120 }

1122 /*ARGSUSED*/
1123 static int
1124 file_cache_constructor(void *buf, void *cdrarg, int kmflags)
1125 {
1126         file_t *fp = buf;

1128         mutex_init(&fp->f_tlock, NULL, MUTEX_DEFAULT, NULL);
1129         return (0);
1130 }

1132 /*ARGSUSED*/
1133 static void
1134 file_cache_destructor(void *buf, void *cdrarg)
1135 {
1136         file_t *fp = buf;

1138         mutex_destroy(&fp->f_tlock);
1139 }

1141 void
1142 finit()
1143 {
1144         file_cache = kmem_cache_create("file_cache", sizeof (file_t), 0,
1145             file_cache_constructor, file_cache_destructor, NULL, NULL, NULL, 0);
1146 }

1148 void
1149 unfalloc(file_t *fp)
1150 {
1151         ASSERT(MUTEX_HELD(&fp->f_tlock));
1152         if (--fp->f_count <= 0) {
1153                 /*
1154                  * deallocate resources to audit_data
1155                  */
1156                 if (audit_active)
1157                         audit_unfalloc(fp);
1158                 crfree(fp->f_cred);
1159                 mutex_exit(&fp->f_tlock);
1160                 kmem_cache_free(file_cache, fp);
1161         } else
1162                 mutex_exit(&fp->f_tlock);
1163 }

1165 /*
1166  * Given a file descriptor, set the user's
1167  * file pointer to the given parameter.
1168  */
1169 void
1170 setf(int fd, file_t *fp)
1171 {
1172         uf_info_t *fip = P_FINFO(curproc);
1173         uf_entry_t *ufp;

1175         if (AU_AUDITING())
1176                 audit_setf(fp, fd);

1178         if (fp == NULL) {
1179                 mutex_enter(&fip->fi_lock);
```

```
1180                 UF_ENTER(ufp, fip, fd);
1181                 fd_reserve(fip, fd, -1);
1182                 mutex_exit(&fip->fi_lock);
1183         } else {
1184                 UF_ENTER(ufp, fip, fd);
1185                 ASSERT(ufp->uf_busy);
1186         }
1187         ASSERT(ufp->uf_fpollinfo == NULL);
1188         ASSERT(ufp->uf_flag == 0);
1189         ufp->uf_file = fp;
1190         cv_broadcast(&ufp->uf_wanted_cv);
1191         UF_EXIT(ufp);
1192 }

1194 /*
1195  * Given a file descriptor, return the file table flags, plus,
1196  * if this is a socket in asynchronous mode, the FASYNC flag.
1197  * getf() may or may not have been called before calling f_getfl().
1198  */
1199 int
1200 f_getfl(int fd, int *flagp)
1201 {
1202         uf_info_t *fip = P_FINFO(curproc);
1203         uf_entry_t *ufp;
1204         file_t *fp;
1205         int error;

1207         if ((uint_t)fd >= fip->fi_nfiles)
1208                 error = EBADF;
1209         else {
1210                 UF_ENTER(ufp, fip, fd);
1211                 if ((fp = ufp->uf_file) == NULL)
1212                         error = EBADF;
1213                 else {
1214                         vnode_t *vp = fp->f_vnode;
1215                         int flag = fp->f_flag | (fp->f_flag2 << 16);

1217                         /*
1218                          * BSD fcntl() FASYNC compatibility.
1219                          */
1220                         if (vp->v_type == VSOCK)
1221                                 flag |= sock_getfasync(vp);
1222                         *flagp = flag;
1223                         error = 0;
1224                 }
1225                 UF_EXIT(ufp);
1226         }

1228         return (error);
1229 }

1231 /*
1232  * Given a file descriptor, return the user's file flags.
1233  * Force the FD_CLOEXEC flag for writable self-open /proc files.
1234  * getf() may or may not have been called before calling f_getfd_error().
1235  */
1236 int
1237 f_getfd_error(int fd, int *flagp)
1238 {
1239         uf_info_t *fip = P_FINFO(curproc);
1240         uf_entry_t *ufp;
1241         file_t *fp;
1242         int flag;
1243         int error;

1245         if ((uint_t)fd >= fip->fi_nfiles)
```



```
1246                 error = EBADF;
1247         else {
1248                 UF_ENTER(ufp, fip, fd);
1249                 if ((fp = ufp->uf_file) == NULL)
1250                         error = EBADF;
1251                 else {
1252                         flag = ufp->uf_flag;
1253                         if ((fp->f_flag & FWRITE) && pr_isself(fp->f_vnode))
1254                                 flag |= FD_CLOEXEC;
1255                         *flagp = flag;
1256                         error = 0;
1257                 }
1258                 UF_EXIT(ufp);
1259         }

1261         return (error);
1262 }

1264 /*
1265  * getf() must have been called before calling f_getfd().
1266  */
1267 char
1268 f_getfd(int fd)
1269 {
1270         int flag = 0;
1271         (void) f_getfd_error(fd, &flag);
1272         return ((char)flag);
1273 }

1275 /*
1276  * Given a file descriptor and file flags, set the user's file flags.
1277  * At present, the only valid flag is FD_CLOEXEC.
1278  * getf() may or may not have been called before calling f_setfd_error().
1279  */
1280 int
1281 f_setfd_error(int fd, int flags)
1282 {
1283         uf_info_t *fip = P_FINFO(curproc);
1284         uf_entry_t *ufp;
1285         int error;

1287         if ((uint_t)fd >= fip->fi_nfiles)
1288                 error = EBADF;
1289         else {
1290                 UF_ENTER(ufp, fip, fd);
1291                 if (ufp->uf_file == NULL)
1292                         error = EBADF;
1293                 else {
1294                         ufp->uf_flag = flags & FD_CLOEXEC;
1295                         error = 0;
1296                 }
1297                 UF_EXIT(ufp);
1298         }
1299         return (error);
1300 }

1302 void
1303 f_setfd(int fd, char flags)
1304 {
1305         (void) f_setfd_error(fd, flags);
1306 }

1308 #define BADFD_MIN       3
1309 #define BADFD_MAX       255

1311 /*
```

```
 * Attempt to allocate a file descriptor which is bad and which
 * is "poison" to the application.  It cannot be closed (except
 * on exec), allocated for a different use, etc.
 */
int
f_badfd(int start, int *fdp, int action)
{
        int fdr;
        int badfd;
        uf_info_t *fip = P_FINFO(curproc);

#ifdef _LP64
        /* No restrictions on 64 bit _file */
        if (get_udatamodel() != DATAMODEL_ILP32)
                return (EINVAL);
#endif

        if (start > BADFD_MAX || start < BADFD_MIN)
                return (EINVAL);

        if (action >= NSIG || action < 0)
                return (EINVAL);

        mutex_enter(&fip->fi_lock);
        badfd = fip->fi_badfd;
        mutex_exit(&fip->fi_lock);

        if (badfd != -1)
                return (EAGAIN);

        fdr = ufalloc(start);

        if (fdr > BADFD_MAX) {
                setf(fdr, NULL);
                return (EMFILE);
        }
        if (fdr < 0)
                return (EMFILE);

        mutex_enter(&fip->fi_lock);
        if (fip->fi_badfd != -1) {
                /* Lost race */
                mutex_exit(&fip->fi_lock);
                setf(fdr, NULL);
                return (EAGAIN);
        }
        fip->fi_action = action;
        fip->fi_badfd = fdr;
        mutex_exit(&fip->fi_lock);
        setf(fdr, NULL);

        *fdp = fdr;

        return (0);
}

/*
 * Allocate a file descriptor and assign it to the vnode "*vpp",
 * performing the usual open protocol upon it and returning the
 * file descriptor allocated.  It is the responsibility of the
 * caller to dispose of "*vpp" if any error occurs.
 */
int
fassign(vnode_t **vpp, int mode, int *fdp)
{
        file_t *fp;
```



```
        int error;
        int fd;

        if (error = falloc((vnode_t *)NULL, mode, &fp, &fd))
                return (error);
        if (error = VOP_OPEN(vpp, mode, fp->f_cred, NULL)) {
                setf(fd, NULL);
                unfalloc(fp);
                return (error);
        }
        fp->f_vnode = *vpp;
        mutex_exit(&fp->f_tlock);
        /*
         * Fill in the slot falloc reserved.
         */
        setf(fd, fp);
        *fdp = fd;
        return (0);
}

/*
 * When a process forks it must increment the f_count of all file pointers
 * since there is a new process pointing at them.  fcnt_add(fip, 1) does this.
 * Since we are called when there is only 1 active lwp we don't need to
 * hold fi_lock or any uf_lock.  If the fork fails, fork_fail() calls
 * fcnt_add(fip, -1) to restore the counts.
 */
void
fcnt_add(uf_info_t *fip, int incr)
{
        int i;
        uf_entry_t *ufp;
        file_t *fp;

        ufp = fip->fi_list;
        for (i = 0; i < fip->fi_nfiles; i++, ufp++) {
                if ((fp = ufp->uf_file) != NULL) {
                        mutex_enter(&fp->f_tlock);
                        ASSERT((incr == 1 && fp->f_count >= 1) ||
                            (incr == -1 && fp->f_count >= 2));
                        fp->f_count += incr;
                        mutex_exit(&fp->f_tlock);
                }
        }
}

/*
 * This is called from exec to close all fd's that have the FD_CLOEXEC flag
 * set and also to close all self-open for write /proc file descriptors.
 */
void
close_exec(uf_info_t *fip)
{
        int fd;
        file_t *fp;
        fpollinfo_t *fpip;
        uf_entry_t *ufp;
        portfd_t *pfd;

        ufp = fip->fi_list;
        for (fd = 0; fd < fip->fi_nfiles; fd++, ufp++) {
                if ((fp = ufp->uf_file) != NULL &&
                    ((ufp->uf_flag & FD_CLOEXEC) ||
                    ((fp->f_flag & FWRITE) && pr_isself(fp->f_vnode)))) {
                        fpip = ufp->uf_fpollinfo;
                        mutex_enter(&fip->fi_lock);
```

```
                        mutex_enter(&ufp->uf_lock);
                        fd_reserve(fip, fd, -1);
                        mutex_exit(&fip->fi_lock);
                        ufp->uf_file = NULL;
                        ufp->uf_fpollinfo = NULL;
                        ufp->uf_flag = 0;
                        /*
                         * We may need to cleanup some cached poll states
                         * in t_pollstate before the fd can be reused. It
                         * is important that we don't access a stale thread
                         * structure. We will do the cleanup in two
                         * phases to avoid deadlock and holding uf_lock for
                         * too long. In phase 1, hold the uf_lock and call
                         * pollblockexit() to set state in t_pollstate struct
                         * so that a thread does not exit on us. In phase 2,
                         * we drop the uf_lock and call pollcacheclean().
                         */
                        pfd = ufp->uf_portfd;
                        ufp->uf_portfd = NULL;
                        if (fpip != NULL)
                                pollblockexit(fpip);
                        mutex_exit(&ufp->uf_lock);
                        if (fpip != NULL)
                                pollcacheclean(fpip, fd);
                        if (pfd)
                                port_close_fd(pfd);
                        (void) closef(fp);
                }
        }

        /* Reset bad fd */
        fip->fi_badfd = -1;
        fip->fi_action = -1;
}

/*
 * Utility function called by most of the *at() system call interfaces.
 *
 * Generate a starting vnode pointer for an (fd, path) pair where 'fd'
 * is an open file descriptor for a directory to be used as the starting
 * point for the lookup of the relative pathname 'path' (or, if path is
 * NULL, generate a vnode pointer for the direct target of the operation).
 *
 * If we successfully return a non-NULL startvp, it has been the target
 * of VN_HOLD() and the caller must call VN_RELE() on it.
 */
int
fgetstartvp(int fd, char *path, vnode_t **startvpp)
{
        vnode_t         *startvp;
        file_t          *startfp;
        char            startchar;

        if (fd == AT_FDCWD && path == NULL)
                return (EFAULT);

        if (fd == AT_FDCWD) {
                /*
                 * Start from the current working directory.
                 */
                startvp = NULL;
        } else {
                if (path == NULL)
                        startchar = '\0';
                else if (copyin(path, &startchar, sizeof (char)))
                        return (EFAULT);
```



```
                if (startchar == '/') {
                        /*
                         * 'path' is an absolute pathname.
                         */
                        startvp = NULL;
                } else {
                        /*
                         * 'path' is a relative pathname or we will
                         * be applying the operation to 'fd' itself.
                         */
                        if ((startfp = getf(fd)) == NULL)
                                return (EBADF);
                        startvp = startfp->f_vnode;
                        VN_HOLD(startvp);
                        releasef(fd);
                }
        }
        *startvpp = startvp;
        return (0);
}

/*
 * Called from fchownat() and fchmodat() to set ownership and mode.
 * The contents of *vap must be set before calling here.
 */
int
fsetattrat(int fd, char *path, int flags, struct vattr *vap)
{
        vnode_t         *startvp;
        vnode_t         *vp;
        int             error;

        /*
         * Since we are never called to set the size of a file, we don't
         * need to check for non-blocking locks (via nbl_need_check(vp)).
         */
        ASSERT(!(vap->va_mask & AT_SIZE));

        if ((error = fgetstartvp(fd, path, &startvp)) != 0)
                return (error);
        if (AU_AUDITING() && startvp != NULL)
                audit_setfsat_path(1);

        /*
         * Do lookup for fchownat/fchmodat when path not NULL
         */
        if (path != NULL) {
                if (error = lookupnameat(path, UIO_USERSPACE,
                    (flags == AT_SYMLINK_NOFOLLOW) ?
                    NO_FOLLOW : FOLLOW,
                    NULLVPP, &vp, startvp)) {
                        if (startvp != NULL)
                                VN_RELE(startvp);
                        return (error);
                }
        } else {
                vp = startvp;
                ASSERT(vp);
                VN_HOLD(vp);
        }

        if (vn_is_readonly(vp)) {
                error = EROFS;
        } else {
                error = VOP_SETATTR(vp, vap, 0, CRED(), NULL);
```

```
1576         }

1578         if (startvp != NULL)
1579                 VN_RELE(startvp);
1580         VN_RELE(vp);

1582         return (error);
1583 }

1585 /*
1586  * Return true if the given vnode is referenced by any
1587  * entry in the current process's file descriptor table.
1588  */
1589 int
1590 fisopen(vnode_t *vp)
1591 {
1592         int fd;
1593         file_t *fp;
1594         vnode_t *ovp;
1595         uf_info_t *fip = P_FINFO(curproc);
1596         uf_entry_t *ufp;

1598         mutex_enter(&fip->fi_lock);
1599         for (fd = 0; fd < fip->fi_nfiles; fd++) {
1600                 UF_ENTER(ufp, fip, fd);
1601                 if ((fp = ufp->uf_file) != NULL &&
1602                     (ovp = fp->f_vnode) != NULL && VN_CMP(vp, ovp)) {
1603                         UF_EXIT(ufp);
1604                         mutex_exit(&fip->fi_lock);
1605                         return (1);
1606                 }
1607                 UF_EXIT(ufp);
1608         }
1609         mutex_exit(&fip->fi_lock);
1610         return (0);
1611 }

1613 /*
1614  * Return zero if at least one file currently open (by curproc) shouldn't be
1615  * allowed to change zones.
1616  */
1617 int
1618 files_can_change_zones(void)
1619 {
1620         int fd;
1621         file_t *fp;
1622         uf_info_t *fip = P_FINFO(curproc);
1623         uf_entry_t *ufp;

1625         mutex_enter(&fip->fi_lock);
1626         for (fd = 0; fd < fip->fi_nfiles; fd++) {
1627                 UF_ENTER(ufp, fip, fd);
1628                 if ((fp = ufp->uf_file) != NULL &&
1629                     !vn_can_change_zones(fp->f_vnode)) {
1630                         UF_EXIT(ufp);
1631                         mutex_exit(&fip->fi_lock);
1632                         return (0);
1633                 }
1634                 UF_EXIT(ufp);
1635         }
1636         mutex_exit(&fip->fi_lock);
1637         return (1);
1638 }

1640 #ifdef DEBUG
```



```
1642 /*
1643  * The following functions are only used in ASSERT()s elsewhere.
1644  * They do not modify the state of the system.
1645  */

1647 /*
1648  * Return true (1) if the current thread is in the fpollinfo
1649  * list for this file descriptor, else false (0).
1650  */
1651 static int
1652 curthread_in_plist(uf_entry_t *ufp)
1653 {
1654         fpollinfo_t *fpip;

1656         ASSERT(MUTEX_HELD(&ufp->uf_lock));
1657         for (fpip = ufp->uf_fpollinfo; fpip; fpip = fpip->fp_next)
1658                 if (fpip->fp_thread == curthread)
1659                         return (1);
1660         return (0);
1661 }

1663 /*
1664  * Sanity check to make sure that after lwp_exit(),
1665  * curthread does not appear on any fd's fpollinfo list.
1666  */
1667 void
1668 checkfpollinfo(void)
1669 {
1670         int fd;
1671         uf_info_t *fip = P_FINFO(curproc);
1672         uf_entry_t *ufp;

1674         mutex_enter(&fip->fi_lock);
1675         for (fd = 0; fd < fip->fi_nfiles; fd++) {
1676                 UF_ENTER(ufp, fip, fd);
1677                 ASSERT(!curthread_in_plist(ufp));
1678                 UF_EXIT(ufp);
1679         }
1680         mutex_exit(&fip->fi_lock);
1681 }

1683 /*
1684  * Return true (1) if the current thread is in the fpollinfo
1685  * list for this file descriptor, else false (0).
1686  * This is the same as curthread_in_plist(),
1687  * but is called w/o holding uf_lock.
1688  */
1689 int
1690 infpollinfo(int fd)
1691 {
1692         uf_info_t *fip = P_FINFO(curproc);
1693         uf_entry_t *ufp;
1694         int rc;

1696         UF_ENTER(ufp, fip, fd);
1697         rc = curthread_in_plist(ufp);
1698         UF_EXIT(ufp);
1699         return (rc);
1700 }

1702 #endif  /* DEBUG */

1704 /*
1705  * Add the curthread to fpollinfo list, meaning this fd is currently in the
1706  * thread's poll cache. Each lwp polling this file descriptor should call
1707  * this routine once.
```

```
 */
void
addfpollinfo(int fd)
{
        struct uf_entry *ufp;
        fpollinfo_t *fpip;
        uf_info_t *fip = P_FINFO(curproc);

        fpip = kmem_zalloc(sizeof (fpollinfo_t), KM_SLEEP);
        fpip->fp_thread = curthread;
        UF_ENTER(ufp, fip, fd);
        /*
         * Assert we are not already on the list, that is, that
         * this lwp did not call addfpollinfo twice for the same fd.
         */
        ASSERT(!curthread_in_plist(ufp));
        /*
         * addfpollinfo is always done inside the getf/releasef pair.
         */
        ASSERT(ufp->uf_refcnt >= 1);
        fpip->fp_next = ufp->uf_fpollinfo;
        ufp->uf_fpollinfo = fpip;
        UF_EXIT(ufp);
}

/*
 * Delete curthread from fpollinfo list if it is there.
 */
void
delfpollinfo(int fd)
{
        struct uf_entry *ufp;
        struct fpollinfo *fpip;
        struct fpollinfo **fpipp;
        uf_info_t *fip = P_FINFO(curproc);

        UF_ENTER(ufp, fip, fd);
        for (fpipp = &ufp->uf_fpollinfo;
            (fpip = *fpipp) != NULL;
            fpipp = &fpip->fp_next) {
                if (fpip->fp_thread == curthread) {
                        *fpipp = fpip->fp_next;
                        kmem_free(fpip, sizeof (fpollinfo_t));
                        break;
                }
        }
        /*
         * Assert that we are not still on the list, that is, that
         * this lwp did not call addfpollinfo twice for the same fd.
         */
        ASSERT(!curthread_in_plist(ufp));
        UF_EXIT(ufp);
}

/*
 * fd is associated with a port. pfd is a pointer to the fd entry in the
 * cache of the port.
 */

void
addfd_port(int fd, portfd_t *pfd)
{
        struct uf_entry *ufp;
        uf_info_t *fip = P_FINFO(curproc);

        UF_ENTER(ufp, fip, fd);
```



```
        /*
         * addfd_port is always done inside the getf/releasef pair.
         */
        ASSERT(ufp->uf_refcnt >= 1);
        if (ufp->uf_portfd == NULL) {
                /* first entry */
                ufp->uf_portfd = pfd;
                pfd->pfd_next = NULL;
        } else {
                pfd->pfd_next = ufp->uf_portfd;
                ufp->uf_portfd = pfd;
                pfd->pfd_next->pfd_prev = pfd;
        }
        UF_EXIT(ufp);
}

void
delfd_port(int fd, portfd_t *pfd)
{
        struct uf_entry *ufp;
        uf_info_t *fip = P_FINFO(curproc);

        UF_ENTER(ufp, fip, fd);
        /*
         * delfd_port is always done inside the getf/releasef pair.
         */
        ASSERT(ufp->uf_refcnt >= 1);
        if (ufp->uf_portfd == pfd) {
                /* remove first entry */
                ufp->uf_portfd = pfd->pfd_next;
        } else {
                pfd->pfd_prev->pfd_next = pfd->pfd_next;
                if (pfd->pfd_next != NULL)
                        pfd->pfd_next->pfd_prev = pfd->pfd_prev;
        }
        UF_EXIT(ufp);
}

static void
port_close_fd(portfd_t *pfd)
{
        portfd_t        *pfdn;

        /*
         * At this point, no other thread should access
         * the portfd_t list for this fd. The uf_file, uf_portfd
         * pointers in the uf_entry_t struct for this fd would
         * be set to NULL.
         */
        for (; pfd != NULL; pfd = pfdn) {
                pfdn = pfd->pfd_next;
                port_close_pfd(pfd);
        }
}
```

```
**********************************************************
  101928 Tue Jan 14 16:50:04 2014
new/usr/src/uts/common/sys/dtrace.h
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

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

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

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

/*
 * DTrace Intermediate Format (DIF)
 *
 * The following definitions describe the DTrace Intermediate Format (DIF), a
 * a RISC-like instruction set and program encoding used to represent
 * predicates and actions that can be bound to DTrace probes.  The constants
 * below defining the number of available registers are suggested minimums; the
 * compiler should use DTRACEIOC_CONF to dynamically obtain the number of
 * registers provided by the current DTrace implementation.
 */
#define DIF_VERSION_1   1               /* DIF version 1: Solaris 10 Beta */
#define DIF_VERSION_2   2               /* DIF version 2: Solaris 10 FCS */
#define DIF_VERSION     DIF_VERSION_2   /* latest DIF instruction set version */
#define DIF_DIR_NREGS   8               /* number of DIF integer registers */
#define DIF_DTR_NREGS   8               /* number of DIF tuple registers */

#define DIF_OP_OR       1               /* or   r1, r2, rd */
#define DIF_OP_XOR      2               /* xor  r1, r2, rd */
#define DIF_OP_AND      3               /* and  r1, r2, rd */
#define DIF_OP_SLL      4               /* sll  r1, r2, rd */
#define DIF_OP_SRL      5               /* srl  r1, r2, rd */
#define DIF_OP_SUB      6               /* sub  r1, r2, rd */
#define DIF_OP_ADD      7               /* add  r1, r2, rd */
#define DIF_OP_MUL      8               /* mul  r1, r2, rd */
#define DIF_OP_SDIV     9               /* sdiv r1, r2, rd */
#define DIF_OP_UDIV     10              /* udiv r1, r2, rd */
```

```
#define DIF_OP_SREM     11              /* srem r1, r2, rd */
#define DIF_OP_UREM     12              /* urem r1, r2, rd */
#define DIF_OP_NOT      13              /* not  r1, rd */
#define DIF_OP_MOV      14              /* mov  r1, rd */
#define DIF_OP_CMP      15              /* cmp  r1, r2 */
#define DIF_OP_TST      16              /* tst  r1 */
#define DIF_OP_BA       17              /* ba   label */
#define DIF_OP_BE       18              /* be   label */
#define DIF_OP_BNE      19              /* bne  label */
#define DIF_OP_BG       20              /* bg   label */
#define DIF_OP_BGU      21              /* bgu  label */
#define DIF_OP_BGE      22              /* bge  label */
#define DIF_OP_BGEU     23              /* bgeu label */
#define DIF_OP_BL       24              /* bl   label */
#define DIF_OP_BLU      25              /* blu  label */
#define DIF_OP_BLE      26              /* ble  label */
#define DIF_OP_BLEU     27              /* bleu label */
#define DIF_OP_LDSB     28              /* ldsb [r1], rd */
#define DIF_OP_LDSH     29              /* ldsh [r1], rd */
#define DIF_OP_LDSW     30              /* ldsw [r1], rd */
#define DIF_OP_LDUB     31              /* ldub [r1], rd */
#define DIF_OP_LDUH     32              /* lduh [r1], rd */
#define DIF_OP_LDUW     33              /* lduw [r1], rd */
#define DIF_OP_LDX      34              /* ldx  [r1], rd */
#define DIF_OP_RET      35              /* ret  rd */
#define DIF_OP_NOP      36              /* nop */
#define DIF_OP_SETX     37              /* setx intindex, rd */
#define DIF_OP_SETS     38              /* sets strindex, rd */
#define DIF_OP_SCMP     39              /* scmp r1, r2 */
#define DIF_OP_LDGA     40              /* ldga var, ri, rd */
#define DIF_OP_LDGS     41              /* ldgs var, rd */
#define DIF_OP_STGS     42              /* stgs var, rs */
#define DIF_OP_LDTA     43              /* ldta var, ri, rd */
#define DIF_OP_LDTS     44              /* ldts var, rd */
#define DIF_OP_STTS     45              /* stts var, rs */
#define DIF_OP_SRA      46              /* sra  r1, r2, rd */
#define DIF_OP_CALL     47              /* call subr, rd */
#define DIF_OP_PUSHTR   48              /* pushtr type, rs, rr */
#define DIF_OP_PUSHTV   49              /* pushtv type, rs, rv */
#define DIF_OP_POPTS    50              /* popts */
#define DIF_OP_FLUSHTS  51              /* flushts */
#define DIF_OP_LDGAA    52              /* ldgaa var, rd */
#define DIF_OP_LDTAA    53              /* ldtaa var, rd */
#define DIF_OP_STGAA    54              /* stgaa var, rs */
#define DIF_OP_STTAA    55              /* sttaa var, rs */
#define DIF_OP_LDLS     56              /* ldls var, rd */
#define DIF_OP_STLS     57              /* stls var, rs */
#define DIF_OP_ALLOCS   58              /* allocs r1, rd */
#define DIF_OP_COPYS    59              /* copys  r1, r2, rd */
#define DIF_OP_STB      60              /* stb  r1, [rd] */
#define DIF_OP_STH      61              /* sth  r1, [rd] */
#define DIF_OP_STW      62              /* stw  r1, [rd] */
#define DIF_OP_STX      63              /* stx  r1, [rd] */
#define DIF_OP_ULDSB    64              /* uldsb [r1], rd */
#define DIF_OP_ULDSH    65              /* uldsh [r1], rd */
#define DIF_OP_ULDSW    66              /* uldsw [r1], rd */
#define DIF_OP_ULDUB    67              /* uldub [r1], rd */
#define DIF_OP_ULDUH    68              /* ulduh [r1], rd */
#define DIF_OP_ULDUW    69              /* ulduw [r1], rd */
#define DIF_OP_ULDX     70              /* uldx  [r1], rd */
#define DIF_OP_RLDSB    71              /* rldsb [r1], rd */
#define DIF_OP_RLDSH    72              /* rldsh [r1], rd */
#define DIF_OP_RLDSW    73              /* rldsw [r1], rd */
#define DIF_OP_RLDUB    74              /* rldub [r1], rd */
#define DIF_OP_RLDUH    75              /* rlduh [r1], rd */
#define DIF_OP_RLDUW    76              /* rlduw [r1], rd */
```



```
#define DIF_OP_RLDX     77              /* rldx  [r1], rd */
#define DIF_OP_XLATE    78              /* xlate xlrindex, rd */
#define DIF_OP_XLARG    79              /* xlarg xlrindex, rd */

#define DIF_INTOFF_MAX          0xffff  /* highest integer table offset */
#define DIF_STROFF_MAX          0xffff  /* highest string table offset */
#define DIF_REGISTER_MAX        0xff    /* highest register number */
#define DIF_VARIABLE_MAX        0xffff  /* highest variable identifier */
#define DIF_SUBROUTINE_MAX      0xffff  /* highest subroutine code */

#define DIF_VAR_ARRAY_MIN       0x0000  /* lowest numbered array variable */
#define DIF_VAR_ARRAY_UBASE     0x0080  /* lowest user-defined array */
#define DIF_VAR_ARRAY_MAX       0x00ff  /* highest numbered array variable */

#define DIF_VAR_OTHER_MIN       0x0100  /* lowest numbered scalar or assc */
#define DIF_VAR_OTHER_UBASE     0x0500  /* lowest user-defined scalar or assc */
#define DIF_VAR_OTHER_MAX       0xffff  /* highest numbered scalar or assc */

#define DIF_VAR_ARGS            0x0000  /* arguments array */
#define DIF_VAR_REGS            0x0001  /* registers array */
#define DIF_VAR_UREGS           0x0002  /* user registers array */
#define DIF_VAR_VMREGS          0x0003  /* virtual machine registers array */
#define DIF_VAR_CURTHREAD       0x0100  /* thread pointer */
#define DIF_VAR_TIMESTAMP       0x0101  /* timestamp */
#define DIF_VAR_VTIMESTAMP      0x0102  /* virtual timestamp */
#define DIF_VAR_IPL             0x0103  /* interrupt priority level */
#define DIF_VAR_EPID            0x0104  /* enabled probe ID */
#define DIF_VAR_ID              0x0105  /* probe ID */
#define DIF_VAR_ARG0            0x0106  /* first argument */
#define DIF_VAR_ARG1            0x0107  /* second argument */
#define DIF_VAR_ARG2            0x0108  /* third argument */
#define DIF_VAR_ARG3            0x0109  /* fourth argument */
#define DIF_VAR_ARG4            0x010a  /* fifth argument */
#define DIF_VAR_ARG5            0x010b  /* sixth argument */
#define DIF_VAR_ARG6            0x010c  /* seventh argument */
#define DIF_VAR_ARG7            0x010d  /* eighth argument */
#define DIF_VAR_ARG8            0x010e  /* ninth argument */
#define DIF_VAR_ARG9            0x010f  /* tenth argument */
#define DIF_VAR_STACKDEPTH      0x0110  /* stack depth */
#define DIF_VAR_CALLER          0x0111  /* caller */
#define DIF_VAR_PROBEPROV       0x0112  /* probe provider */
#define DIF_VAR_PROBEMOD        0x0113  /* probe module */
#define DIF_VAR_PROBEFUNC       0x0114  /* probe function */
#define DIF_VAR_PROBENAME       0x0115  /* probe name */
#define DIF_VAR_PID             0x0116  /* process ID */
#define DIF_VAR_TID             0x0117  /* (per-process) thread ID */
#define DIF_VAR_EXECNAME        0x0118  /* name of executable */
#define DIF_VAR_ZONENAME        0x0119  /* zone name associated with process */
#define DIF_VAR_WALLTIMESTAMP   0x011a  /* wall-clock timestamp */
#define DIF_VAR_USTACKDEPTH     0x011b  /* user-land stack depth */
#define DIF_VAR_UCALLER         0x011c  /* user-level caller */
#define DIF_VAR_PPID            0x011d  /* parent process ID */
#define DIF_VAR_UID             0x011e  /* process user ID */
#define DIF_VAR_GID             0x011f  /* process group ID */
#define DIF_VAR_ERRNO           0x0120  /* thread errno */

#define DIF_SUBR_RAND                   0
#define DIF_SUBR_MUTEX_OWNED            1
#define DIF_SUBR_MUTEX_OWNER            2
#define DIF_SUBR_MUTEX_TYPE_ADAPTIVE    3
#define DIF_SUBR_MUTEX_TYPE_SPIN        4
#define DIF_SUBR_RW_READ_HELD           5
#define DIF_SUBR_RW_WRITE_HELD          6
#define DIF_SUBR_RW_ISWRITER            7
#define DIF_SUBR_COPYIN                 8
#define DIF_SUBR_COPYINSTR              9
```

```
 255 #define DIF_SUBR_SPECULATION            10
 256 #define DIF_SUBR_PROGENYOF              11
 257 #define DIF_SUBR_STRLEN                 12
 258 #define DIF_SUBR_COPYOUT                13
 259 #define DIF_SUBR_COPYOUTSTR             14
 260 #define DIF_SUBR_ALLOCA                 15
 261 #define DIF_SUBR_BCOPY                  16
 262 #define DIF_SUBR_COPYINTO               17
 263 #define DIF_SUBR_MSGDSIZE               18
 264 #define DIF_SUBR_MSGSIZE                19
 265 #define DIF_SUBR_GETMAJOR               20
 266 #define DIF_SUBR_GETMINOR               21
 267 #define DIF_SUBR_DDI_PATHNAME           22
 268 #define DIF_SUBR_STRJOIN                23
 269 #define DIF_SUBR_LLTOSTR                24
 270 #define DIF_SUBR_BASENAME               25
 271 #define DIF_SUBR_DIRNAME                26
 272 #define DIF_SUBR_CLEANPATH              27
 273 #define DIF_SUBR_STRCHR                 28
 274 #define DIF_SUBR_STRRCHR                29
 275 #define DIF_SUBR_STRSTR                 30
 276 #define DIF_SUBR_STRTOK                 31
 277 #define DIF_SUBR_SUBSTR                 32
 278 #define DIF_SUBR_INDEX                  33
 279 #define DIF_SUBR_RINDEX                 34
 280 #define DIF_SUBR_HTONS                  35
 281 #define DIF_SUBR_HTONL                  36
 282 #define DIF_SUBR_HTONLL                 37
 283 #define DIF_SUBR_NTOHS                  38
 284 #define DIF_SUBR_NTOHL                  39
 285 #define DIF_SUBR_NTOHLL                 40
 286 #define DIF_SUBR_INET_NTOP              41
 287 #define DIF_SUBR_INET_NTOA              42
 288 #define DIF_SUBR_INET_NTOA6             43
 289 #define DIF_SUBR_TOUPPER                44
 290 #define DIF_SUBR_TOLOWER                45
 291 #define DIF_SUBR_GETF                   46
 292 #endif /* ! codereview */

 294 #define DIF_SUBR_MAX                    46      /* max subroutine value */
 291 #define DIF_SUBR_MAX                    45      /* max subroutine value */

 296 typedef uint32_t dif_instr_t;

 298 #define DIF_INSTR_OP(i)                 (((i) >> 24) & 0xff)
 299 #define DIF_INSTR_R1(i)                 (((i) >> 16) & 0xff)
 300 #define DIF_INSTR_R2(i)                 (((i) >>  8) & 0xff)
 301 #define DIF_INSTR_RD(i)                 ((i) & 0xff)
 302 #define DIF_INSTR_RS(i)                 ((i) & 0xff)
 303 #define DIF_INSTR_LABEL(i)              ((i) & 0xffffff)
 304 #define DIF_INSTR_VAR(i)                (((i) >>  8) & 0xffff)
 305 #define DIF_INSTR_INTEGER(i)            (((i) >>  8) & 0xffff)
 306 #define DIF_INSTR_STRING(i)             (((i) >>  8) & 0xffff)
 307 #define DIF_INSTR_SUBR(i)               (((i) >>  8) & 0xffff)
 308 #define DIF_INSTR_TYPE(i)               (((i) >> 16) & 0xff)
 309 #define DIF_INSTR_XLREF(i)              (((i) >>  8) & 0xffff)

 311 #define DIF_INSTR_FMT(op, r1, r2, d) \
 312         (((op) << 24) | ((r1) << 16) | ((r2) << 8) | (d))

 314 #define DIF_INSTR_NOT(r1, d)            (DIF_INSTR_FMT(DIF_OP_NOT, r1, 0, d))
 315 #define DIF_INSTR_MOV(r1, d)            (DIF_INSTR_FMT(DIF_OP_MOV, r1, 0, d))
 316 #define DIF_INSTR_CMP(op, r1, r2)       (DIF_INSTR_FMT(op, r1, r2, 0))
 317 #define DIF_INSTR_TST(r1)               (DIF_INSTR_FMT(DIF_OP_TST, r1, 0, 0))
 318 #define DIF_INSTR_BRANCH(op, label)     (((op) << 24) | (label))
 319 #define DIF_INSTR_LOAD(op, r1, d)       (DIF_INSTR_FMT(op, r1, 0, d))
```



```
 320 #define DIF_INSTR_STORE(op, r1, d)      (DIF_INSTR_FMT(op, r1, 0, d))
 321 #define DIF_INSTR_SETX(i, d)            ((DIF_OP_SETX << 24) | ((i) << 8) | (d))
 322 #define DIF_INSTR_SETS(s, d)            ((DIF_OP_SETS << 24) | ((s) << 8) | (d))
 323 #define DIF_INSTR_RET(d)                (DIF_INSTR_FMT(DIF_OP_RET, 0, 0, d))
 324 #define DIF_INSTR_NOP                   (DIF_OP_NOP << 24)
 325 #define DIF_INSTR_LDA(op, v, r, d)      (DIF_INSTR_FMT(op, v, r, d))
 326 #define DIF_INSTR_LDV(op, v, d)         (((op) << 24) | ((v) << 8) | (d))
 327 #define DIF_INSTR_STV(op, v, rs)        (((op) << 24) | ((v) << 8) | (rs))
 328 #define DIF_INSTR_CALL(s, d)            ((DIF_OP_CALL << 24) | ((s) << 8) | (d))
 329 #define DIF_INSTR_PUSHTS(op, t, r2, rs) (DIF_INSTR_FMT(op, t, r2, rs))
 330 #define DIF_INSTR_POPTS                 (DIF_OP_POPTS << 24)
 331 #define DIF_INSTR_FLUSHTS               (DIF_OP_FLUSHTS << 24)
 332 #define DIF_INSTR_ALLOCS(r1, d)         (DIF_INSTR_FMT(DIF_OP_ALLOCS, r1, 0, d))
 333 #define DIF_INSTR_COPYS(r1, r2, d)      (DIF_INSTR_FMT(DIF_OP_COPYS, r1, r2, d))
 334 #define DIF_INSTR_XLATE(op, r, d)       (((op) << 24) | ((r) << 8) | (d))

 336 #define DIF_REG_R0      0               /* %r0 is always set to zero */

 338 /*
 339  * A DTrace Intermediate Format Type (DIF Type) is used to represent the types
 340  * of variables, function and associative array arguments, and the return type
 341  * for each DIF object (shown below).  It contains a description of the type,
 342  * its size in bytes, and a module identifier.
 343  */
 344 typedef struct dtrace_diftype {
 345         uint8_t dtdt_kind;              /* type kind (see below) */
 346         uint8_t dtdt_ckind;             /* type kind in CTF */
 347         uint8_t dtdt_flags;             /* type flags (see below) */
 348         uint8_t dtdt_pad;               /* reserved for future use */
 349         uint32_t dtdt_size;             /* type size in bytes (unless string) */
 350 } dtrace_diftype_t;
_____unchanged_portion_omitted_

1351 #define DTRACEMNR_DTRACE        "dtrace"        /* node for DTrace ops */
1352 #define DTRACEMNR_HELPER        "helper"        /* node for helpers */
1353 #define DTRACEMNRN_DTRACE       0               /* minor for DTrace ops */
1354 #define DTRACEMNRN_HELPER       1               /* minor for helpers */
1355 #define DTRACEMNRN_CLONE        2               /* first clone minor */

1357 #ifdef _KERNEL

1359 /*
1360  * DTrace Provider API
1361  *
1362  * The following functions are implemented by the DTrace framework and are
1363  * used to implement separate in-kernel DTrace providers.  Common functions
1364  * are provided in uts/common/os/dtrace.c.  ISA-dependent subroutines are
1365  * defined in uts/<isa>/dtrace/dtrace_asm.s or uts/<isa>/dtrace/dtrace_isa.c.
1366  *
1367  * The provider API has two halves:  the API that the providers consume from
1368  * DTrace, and the API that providers make available to DTrace.
1369  *
1370  * 1 Framework-to-Provider API
1371  *
1372  * 1.1  Overview
1373  *
1374  * The Framework-to-Provider API is represented by the dtrace_pops structure
1375  * that the provider passes to the framework when registering itself.  This
1376  * structure consists of the following members:
1377  *
1378  *   dtps_provide()         <-- Provide all probes, all modules
1379  *   dtps_provide_module()  <-- Provide all probes in specified module
1380  *   dtps_enable()          <-- Enable specified probe
1381  *   dtps_disable()         <-- Disable specified probe
1382  *   dtps_suspend()         <-- Suspend specified probe
1383  *   dtps_resume()          <-- Resume specified probe
```

```
1384  *   dtps_getargdesc()       <-- Get the argument description for args[X]
1385  *   dtps_getargval()        <-- Get the value for an argX or args[X] variable
1386  *   dtps_mode()             <-- Return the mode of the fired probe
1387  *   dtps_destroy()          <-- Destroy all state associated with this probe
1388  *
1389  * 1.2  void dtps_provide(void *arg, const dtrace_probedesc_t *spec)
1390  *
1391  * 1.2.1  Overview
1392  *
1393  *   Called to indicate that the provider should provide all probes.  If the
1394  *   specified description is non-NULL, dtps_provide() is being called because
1395  *   no probe matched a specified probe -- if the provider has the ability to
1396  *   create custom probes, it may wish to create a probe that matches the
1397  *   specified description.
1398  *
1399  * 1.2.2  Arguments and notes
1400  *
1401  *   The first argument is the cookie as passed to dtrace_register().  The
1402  *   second argument is a pointer to a probe description that the provider may
1403  *   wish to consider when creating custom probes.  The provider is expected to
1404  *   call back into the DTrace framework via dtrace_probe_create() to create
1405  *   any necessary probes.  dtps_provide() may be called even if the provider
1406  *   has made available all probes; the provider should check the return value
1407  *   of dtrace_probe_create() to handle this case.  Note that the provider need
1408  *   not implement both dtps_provide() and dtps_provide_module(); see
1409  *   "Arguments and Notes" for dtrace_register(), below.
1410  *
1411  * 1.2.3  Return value
1412  *
1413  *   None.
1414  *
1415  * 1.2.4  Caller's context
1416  *
1417  *   dtps_provide() is typically called from open() or ioctl() context, but may
1418  *   be called from other contexts as well.  The DTrace framework is locked in
1419  *   such a way that providers may not register or unregister.  This means that
1420  *   the provider may not call any DTrace API that affects its registration with
1421  *   the framework, including dtrace_register(), dtrace_unregister(),
1422  *   dtrace_invalidate(), and dtrace_condense().  However, the context is such
1423  *   that the provider may (and indeed, is expected to) call probe-related
1424  *   DTrace routines, including dtrace_probe_create(), dtrace_probe_lookup(),
1425  *   and dtrace_probe_arg().
1426  *
1427  * 1.3  void dtps_provide_module(void *arg, struct modctl *mp)
1428  *
1429  * 1.3.1  Overview
1430  *
1431  *   Called to indicate that the provider should provide all probes in the
1432  *   specified module.
1433  *
1434  * 1.3.2  Arguments and notes
1435  *
1436  *   The first argument is the cookie as passed to dtrace_register().  The
1437  *   second argument is a pointer to a modctl structure that indicates the
1438  *   module for which probes should be created.
1439  *
1440  * 1.3.3  Return value
1441  *
1442  *   None.
1443  *
1444  * 1.3.4  Caller's context
1445  *
1446  *   dtps_provide_module() may be called from open() or ioctl() context, but
1447  *   may also be called from a module loading context.  mod_lock is held, and
1448  *   the DTrace framework is locked in such a way that providers may not
1449  *   register or unregister.  This means that the provider may not call any
```



```
1450  *   DTrace API that affects its registration with the framework, including
1451  *   dtrace_register(), dtrace_unregister(), dtrace_invalidate(), and
1452  *   dtrace_condense().  However, the context is such that the provider may (and
1453  *   indeed, is expected to) call probe-related DTrace routines, including
1454  *   dtrace_probe_create(), dtrace_probe_lookup(), and dtrace_probe_arg().  Note
1455  *   that the provider need not implement both dtps_provide() and
1456  *   dtps_provide_module(); see "Arguments and Notes" for dtrace_register(),
1457  *   below.
1458  *
1459  * 1.4  int dtps_enable(void *arg, dtrace_id_t id, void *parg)
1460  *
1461  * 1.4.1  Overview
1462  *
1463  *   Called to enable the specified probe.
1464  *
1465  * 1.4.2  Arguments and notes
1466  *
1467  *   The first argument is the cookie as passed to dtrace_register().  The
1468  *   second argument is the identifier of the probe to be enabled.  The third
1469  *   argument is the probe argument as passed to dtrace_probe_create().
1470  *   dtps_enable() will be called when a probe transitions from not being
1471  *   enabled at all to having one or more ECB.  The number of ECBs associated
1472  *   with the probe may change without subsequent calls into the provider.
1473  *   When the number of ECBs drops to zero, the provider will be explicitly
1474  *   told to disable the probe via dtps_disable().  dtrace_probe() should never
1475  *   be called for a probe identifier that hasn't been explicitly enabled via
1476  *   dtps_enable().
1477  *
1478  * 1.4.3  Return value
1479  *
1480  *   On success, dtps_enable() should return 0. On failure, -1 should be
1481  *   returned.
1482  *
1483  * 1.4.4  Caller's context
1484  *
1485  *   The DTrace framework is locked in such a way that it may not be called
1486  *   back into at all.  cpu_lock is held.  mod_lock is not held and may not
1487  *   be acquired.
1488  *
1489  * 1.5  void dtps_disable(void *arg, dtrace_id_t id, void *parg)
1490  *
1491  * 1.5.1  Overview
1492  *
1493  *   Called to disable the specified probe.
1494  *
1495  * 1.5.2  Arguments and notes
1496  *
1497  *   The first argument is the cookie as passed to dtrace_register().  The
1498  *   second argument is the identifier of the probe to be disabled.  The third
1499  *   argument is the probe argument as passed to dtrace_probe_create().
1500  *   dtps_disable() will be called when a probe transitions from being enabled
1501  *   to having zero ECBs.  dtrace_probe() should never be called for a probe
1502  *   identifier that has been explicitly enabled via dtps_disable().
1503  *
1504  * 1.5.3  Return value
1505  *
1506  *   None.
1507  *
1508  * 1.5.4  Caller's context
1509  *
1510  *   The DTrace framework is locked in such a way that it may not be called
1511  *   back into at all.  cpu_lock is held.  mod_lock is not held and may not
1512  *   be acquired.
1513  *
1514  * 1.6  void dtps_suspend(void *arg, dtrace_id_t id, void *parg)
1515  *
```

```
1516  * 1.6.1  Overview
1517  *
1518  *   Called to suspend the specified enabled probe.  This entry point is for
1519  *   providers that may need to suspend some or all of their probes when CPUs
1520  *   are being powered on or when the boot monitor is being entered for a
1521  *   prolonged period of time.
1522  *
1523  * 1.6.2  Arguments and notes
1524  *
1525  *   The first argument is the cookie as passed to dtrace_register().  The
1526  *   second argument is the identifier of the probe to be suspended.  The
1527  *   third argument is the probe argument as passed to dtrace_probe_create().
1528  *   dtps_suspend will only be called on an enabled probe.  Providers that
1529  *   provide a dtps_suspend entry point will want to take roughly the action
1530  *   that it takes for dtps_disable.
1531  *
1532  * 1.6.3  Return value
1533  *
1534  *   None.
1535  *
1536  * 1.6.4  Caller's context
1537  *
1538  *   Interrupts are disabled.  The DTrace framework is in a state such that the
1539  *   specified probe cannot be disabled or destroyed for the duration of
1540  *   dtps_suspend().  As interrupts are disabled, the provider is afforded
1541  *   little latitude; the provider is expected to do no more than a store to
1542  *   memory.
1543  *
1544  * 1.7  void dtps_resume(void *arg, dtrace_id_t id, void *parg)
1545  *
1546  * 1.7.1  Overview
1547  *
1548  *   Called to resume the specified enabled probe.  This entry point is for
1549  *   providers that may need to resume some or all of their probes after the
1550  *   completion of an event that induced a call to dtps_suspend().
1551  *
1552  * 1.7.2  Arguments and notes
1553  *
1554  *   The first argument is the cookie as passed to dtrace_register().  The
1555  *   second argument is the identifier of the probe to be resumed.  The
1556  *   third argument is the probe argument as passed to dtrace_probe_create().
1557  *   dtps_resume will only be called on an enabled probe.  Providers that
1558  *   provide a dtps_resume entry point will want to take roughly the action
1559  *   that it takes for dtps_enable.
1560  *
1561  * 1.7.3  Return value
1562  *
1563  *   None.
1564  *
1565  * 1.7.4  Caller's context
1566  *
1567  *   Interrupts are disabled.  The DTrace framework is in a state such that the
1568  *   specified probe cannot be disabled or destroyed for the duration of
1569  *   dtps_resume().  As interrupts are disabled, the provider is afforded
1570  *   little latitude; the provider is expected to do no more than a store to
1571  *   memory.
1572  *
1573  * 1.8  void dtps_getargdesc(void *arg, dtrace_id_t id, void *parg,
1574  *           dtrace_argdesc_t *desc)
1575  *
1576  * 1.8.1  Overview
1577  *
1578  *   Called to retrieve the argument description for an args[X] variable.
1579  *
1580  * 1.8.2  Arguments and notes
1581  *
```



```
1582  *   The first argument is the cookie as passed to dtrace_register(). The
1583  *   second argument is the identifier of the current probe. The third
1584  *   argument is the probe argument as passed to dtrace_probe_create(). The
1585  *   fourth argument is a pointer to the argument description.  This
1586  *   description is both an input and output parameter:  it contains the
1587  *   index of the desired argument in the dtargd_ndx field, and expects
1588  *   the other fields to be filled in upon return.  If there is no argument
1589  *   corresponding to the specified index, the dtargd_ndx field should be set
1590  *   to DTRACE_ARGNONE.
1591  *
1592  * 1.8.3  Return value
1593  *
1594  *   None.  The dtargd_ndx, dtargd_native, dtargd_xlate and dtargd_mapping
1595  *   members of the dtrace_argdesc_t structure are all output values.
1596  *
1597  * 1.8.4  Caller's context
1598  *
1599  *   dtps_getargdesc() is called from ioctl() context. mod_lock is held, and
1600  *   the DTrace framework is locked in such a way that providers may not
1601  *   register or unregister.  This means that the provider may not call any
1602  *   DTrace API that affects its registration with the framework, including
1603  *   dtrace_register(), dtrace_unregister(), dtrace_invalidate(), and
1604  *   dtrace_condense().
1605  *
1606  * 1.9  uint64_t dtps_getargval(void *arg, dtrace_id_t id, void *parg,
1607  *              int argno, int aframes)
1608  *
1609  * 1.9.1  Overview
1610  *
1611  *   Called to retrieve a value for an argX or args[X] variable.
1612  *
1613  * 1.9.2  Arguments and notes
1614  *
1615  *   The first argument is the cookie as passed to dtrace_register(). The
1616  *   second argument is the identifier of the current probe. The third
1617  *   argument is the probe argument as passed to dtrace_probe_create(). The
1618  *   fourth argument is the number of the argument (the X in the example in
1619  *   1.9.1). The fifth argument is the number of stack frames that were used
1620  *   to get from the actual place in the code that fired the probe to
1621  *   dtrace_probe() itself, the so-called artificial frames. This argument may
1622  *   be used to descend an appropriate number of frames to find the correct
1623  *   values. If this entry point is left NULL, the dtrace_getarg() built-in
1624  *   function is used.
1625  *
1626  * 1.9.3  Return value
1627  *
1628  *   The value of the argument.
1629  *
1630  * 1.9.4  Caller's context
1631  *
1632  *   This is called from within dtrace_probe() meaning that interrupts
1633  *   are disabled. No locks should be taken within this entry point.
1634  *
1635  * 1.10  int dtps_mode(void *arg, dtrace_id_t id, void *parg)
1636  *
1637  * 1.10.1  Overview
1638  *
1639  *   Called to determine the mode of a fired probe.
1640  *
1641  * 1.10.2  Arguments and notes
1642  *
1643  *   The first argument is the cookie as passed to dtrace_register(). The
1644  *   second argument is the identifier of the current probe.  The third
1645  *   argument is the probe argument as passed to dtrace_probe_create().  This
1646  *   entry point must not be left NULL for providers whose probes allow for
1647  *   mixed mode tracing, that is to say those unanchored probes that can fire
```

```
 *   during kernel- or user-mode execution.
 *
 * 1.10.3  Return value
 *
 *   A bitwise OR that encapsulates both the mode (either DTRACE_MODE_KERNEL
 *   or DTRACE_MODE_USER) and the policy when the privilege of the enabling
 *   is insufficient for that mode (a combination of DTRACE_MODE_NOPRIV_DROP,
 *   DTRACE_MODE_NOPRIV_RESTRICT, and DTRACE_MODE_LIMITEDPRIV_RESTRICT).  If
 *   DTRACE_MODE_NOPRIV_DROP bit is set, insufficient privilege will result
 *   in the probe firing being silently ignored for the enabling; if the
 *   DTRACE_NODE_NOPRIV_RESTRICT bit is set, insufficient privilege will not
 *   prevent probe processing for the enabling, but restrictions will be in
 *   place that induce a UPRIV fault upon attempt to examine probe arguments
 *   or current process state.  If the DTRACE_MODE_LIMITEDPRIV_RESTRICT bit
 *   is set, similar restrictions will be placed upon operation if the
 *   privilege is sufficient to process the enabling, but does not otherwise
 *   entitle the enabling to all zones.  The DTRACE_MODE_NOPRIV_DROP and
 *   DTRACE_MODE_NOPRIV_RESTRICT are mutually exclusive (and one of these
 *   two policies must be specified), but either may be combined (or not)
 *   with DTRACE_MODE_LIMITEDPRIV_RESTRICT.
 *   is insufficient for that mode (either DTRACE_MODE_NOPRIV_DROP or
 *   DTRACE_MODE_NOPRIV_RESTRICT).  If the policy is DTRACE_MODE_NOPRIV_DROP,
 *   insufficient privilege will result in the probe firing being silently
 *   ignored for the enabling; if the policy is DTRACE_NODE_NOPRIV_RESTRICT,
 *   insufficient privilege will not prevent probe processing for the
 *   enabling, but restrictions will be in place that induce a UPRIV fault
 *   upon attempt to examine probe arguments or current process state.
 *
 * 1.10.4  Caller's context
 *
 *   This is called from within dtrace_probe() meaning that interrupts
 *   are disabled. No locks should be taken within this entry point.
 *
 * 1.11 void dtps_destroy(void *arg, dtrace_id_t id, void *parg)
 *
 * 1.11.1 Overview
 *
 *   Called to destroy the specified probe.
 *
 * 1.11.2 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 destroyed.  The third
 *   argument is the probe argument as passed to dtrace_probe_create().  The
 *   provider should free all state associated with the probe.  The framework
 *   guarantees that dtps_destroy() is only called for probes that have either
 *   been disabled via dtps_disable() or were never enabled via dtps_enable().
 *   Once dtps_disable() has been called for a probe, no further call will be
 *   made specifying the probe.
 *
 * 1.11.3 Return value
 *
 *   None.
 *
 * 1.11.4 Caller's context
 *
 *   The DTrace framework is locked in such a way that it may not be called
 *   back into at all.  mod_lock is held.  cpu_lock is not held, and may not be
 *   acquired.
 *
 *
 * 2 Provider-to-Framework API
 *
 * 2.1  Overview
 *
 * The Provider-to-Framework API provides the mechanism for the provider to
```



```
 * register itself with the DTrace framework, to create probes, to lookup
 * probes and (most importantly) to fire probes.  The Provider-to-Framework
 * consists of:
 *
 *   dtrace_register()       <-- Register a provider with the DTrace framework
 *   dtrace_unregister()     <-- Remove a provider's DTrace registration
 *   dtrace_invalidate()     <-- Invalidate the specified provider
 *   dtrace_condense()       <-- Remove a provider's unenabled probes
 *   dtrace_attached()       <-- Indicates whether or not DTrace has attached
 *   dtrace_probe_create()   <-- Create a DTrace probe
 *   dtrace_probe_lookup()   <-- Lookup a DTrace probe based on its name
 *   dtrace_probe_arg()      <-- Return the probe argument for a specific probe
 *   dtrace_probe()          <-- Fire the specified probe
 *
 * 2.2  int dtrace_register(const char *name, const dtrace_pattr_t *pap,
 *          uint32_t priv, cred_t *cr, const dtrace_pops_t *pops, void *arg,
 *          dtrace_provider_id_t *idp)
 *
 * 2.2.1  Overview
 *
 *   dtrace_register() registers the calling provider with the DTrace
 *   framework.  It should generally be called by DTrace providers in their
 *   attach(9E) entry point.
 *
 * 2.2.2  Arguments and Notes
 *
 *   The first argument is the name of the provider.  The second argument is a
 *   pointer to the stability attributes for the provider.  The third argument
 *   is the privilege flags for the provider, and must be some combination of:
 *
 *     DTRACE_PRIV_NONE     <= All users may enable probes from this provider
 *
 *     DTRACE_PRIV_PROC     <= Any user with privilege of PRIV_DTRACE_PROC may
 *                             enable probes from this provider
 *
 *     DTRACE_PRIV_USER     <= Any user with privilege of PRIV_DTRACE_USER may
 *                             enable probes from this provider
 *
 *     DTRACE_PRIV_KERNEL   <= Any user with privilege of PRIV_DTRACE_KERNEL
 *                             may enable probes from this provider
 *
 *     DTRACE_PRIV_OWNER    <= This flag places an additional constraint on
 *                             the privilege requirements above. These probes
 *                             require either (a) a user ID matching the user
 *                             ID of the cred passed in the fourth argument
 *                             or (b) the PRIV_PROC_OWNER privilege.
 *
 *     DTRACE_PRIV_ZONEOWNER<= This flag places an additional constraint on
 *                             the privilege requirements above. These probes
 *                             require either (a) a zone ID matching the zone
 *                             ID of the cred passed in the fourth argument
 *                             or (b) the PRIV_PROC_ZONE privilege.
 *
 *   Note that these flags designate the _visibility_ of the probes, not
 *   the conditions under which they may or may not fire.
 *
 *   The fourth argument is the credential that is associated with the
 *   provider.  This argument should be NULL if the privilege flags don't
 *   include DTRACE_PRIV_OWNER or DTRACE_PRIV_ZONEOWNER.  If non-NULL, the
 *   framework stashes the uid and zoneid represented by this credential
 *   for use at probe-time, in implicit predicates.  These limit visibility
 *   of the probes to users and/or zones which have sufficient privilege to
 *   access them.
 *
 *   The fifth argument is a DTrace provider operations vector, which provides
 *   the implementation for the Framework-to-Provider API.  (See Section 1,
```

```
1773  *   above.)  This must be non-NULL, and each member must be non-NULL.  The
1774  *   exceptions to this are (1) the dtps_provide() and dtps_provide_module()
1775  *   members (if the provider so desires, _one_ of these members may be left
1776  *   NULL -- denoting that the provider only implements the other) and (2)
1777  *   the dtps_suspend() and dtps_resume() members, which must either both be
1778  *   NULL or both be non-NULL.
1779  *
1780  *   The sixth argument is a cookie to be specified as the first argument for
1781  *   each function in the Framework-to-Provider API.  This argument may have
1782  *   any value.
1783  *
1784  *   The final argument is a pointer to dtrace_provider_id_t.  If
1785  *   dtrace_register() successfully completes, the provider identifier will be
1786  *   stored in the memory pointed to be this argument.  This argument must be
1787  *   non-NULL.
1788  *
1789  * 2.2.3  Return value
1790  *
1791  *   On success, dtrace_register() returns 0 and stores the new provider's
1792  *   identifier into the memory pointed to by the idp argument.  On failure,
1793  *   dtrace_register() returns an errno:
1794  *
1795  *     EINVAL   The arguments passed to dtrace_register() were somehow invalid.
1796  *              This may because a parameter that must be non-NULL was NULL,
1797  *              because the name was invalid (either empty or an illegal
1798  *              provider name) or because the attributes were invalid.
1799  *
1800  *   No other failure code is returned.
1801  *
1802  * 2.2.4  Caller's context
1803  *
1804  *   dtrace_register() may induce calls to dtrace_provide(); the provider must
1805  *   hold no locks across dtrace_register() that may also be acquired by
1806  *   dtrace_provide().  cpu_lock and mod_lock must not be held.
1807  *
1808  * 2.3  int dtrace_unregister(dtrace_provider_t id)
1809  *
1810  * 2.3.1  Overview
1811  *
1812  *   Unregisters the specified provider from the DTrace framework.  It should
1813  *   generally be called by DTrace providers in their detach(9E) entry point.
1814  *
1815  * 2.3.2  Arguments and Notes
1816  *
1817  *   The only argument is the provider identifier, as returned from a
1818  *   successful call to dtrace_register().  As a result of calling
1819  *   dtrace_unregister(), the DTrace framework will call back into the provider
1820  *   via the dtps_destroy() entry point.  Once dtrace_unregister() successfully
1821  *   completes, however, the DTrace framework will no longer make calls through
1822  *   the Framework-to-Provider API.
1823  *
1824  * 2.3.3  Return value
1825  *
1826  *   On success, dtrace_unregister returns 0.  On failure, dtrace_unregister()
1827  *   returns an errno:
1828  *
1829  *     EBUSY    There are currently processes that have the DTrace pseudodevice
1830  *              open, or there exists an anonymous enabling that hasn't yet
1831  *              been claimed.
1832  *
1833  *   No other failure code is returned.
1834  *
1835  * 2.3.4  Caller's context
1836  *
1837  *   Because a call to dtrace_unregister() may induce calls through the
1838  *   Framework-to-Provider API, the caller may not hold any lock across
```



```
1839  *   dtrace_register() that is also acquired in any of the Framework-to-
1840  *   Provider API functions.  Additionally, mod_lock may not be held.
1841  *
1842  * 2.4  void dtrace_invalidate(dtrace_provider_id_t id)
1843  *
1844  * 2.4.1  Overview
1845  *
1846  *   Invalidates the specified provider.  All subsequent probe lookups for the
1847  *   specified provider will fail, but its probes will not be removed.
1848  *
1849  * 2.4.2  Arguments and note
1850  *
1851  *   The only argument is the provider identifier, as returned from a
1852  *   successful call to dtrace_register().  In general, a provider's probes
1853  *   always remain valid; dtrace_invalidate() is a mechanism for invalidating
1854  *   an entire provider, regardless of whether or not probes are enabled or
1855  *   not.  Note that dtrace_invalidate() will _not_ prevent already enabled
1856  *   probes from firing -- it will merely prevent any new enablings of the
1857  *   provider's probes.
1858  *
1859  * 2.5 int dtrace_condense(dtrace_provider_id_t id)
1860  *
1861  * 2.5.1  Overview
1862  *
1863  *   Removes all the unenabled probes for the given provider. This function is
1864  *   not unlike dtrace_unregister(), except that it doesn't remove the
1865  *   provider just as many of its associated probes as it can.
1866  *
1867  * 2.5.2  Arguments and Notes
1868  *
1869  *   As with dtrace_unregister(), the sole argument is the provider identifier
1870  *   as returned from a successful call to dtrace_register().  As a result of
1871  *   calling dtrace_condense(), the DTrace framework will call back into the
1872  *   given provider's dtps_destroy() entry point for each of the provider's
1873  *   unenabled probes.
1874  *
1875  * 2.5.3  Return value
1876  *
1877  *   Currently, dtrace_condense() always returns 0.  However, consumers of this
1878  *   function should check the return value as appropriate; its behavior may
1879  *   change in the future.
1880  *
1881  * 2.5.4  Caller's context
1882  *
1883  *   As with dtrace_unregister(), the caller may not hold any lock across
1884  *   dtrace_condense() that is also acquired in the provider's entry points.
1885  *   Also, mod_lock may not be held.
1886  *
1887  * 2.6 int dtrace_attached()
1888  *
1889  * 2.6.1  Overview
1890  *
1891  *   Indicates whether or not DTrace has attached.
1892  *
1893  * 2.6.2  Arguments and Notes
1894  *
1895  *   For most providers, DTrace makes initial contact beyond registration.
1896  *   That is, once a provider has registered with DTrace, it waits to hear
1897  *   from DTrace to create probes.  However, some providers may wish to
1898  *   proactively create probes without first being told by DTrace to do so.
1899  *   If providers wish to do this, they must first call dtrace_attached() to
1900  *   determine if DTrace itself has attached.  If dtrace_attached() returns 0,
1901  *   the provider must not make any other Provider-to-Framework API call.
1902  *
1903  * 2.6.3  Return value
1904  *
```

```
1905  *   dtrace_attached() returns 1 if DTrace has attached, 0 otherwise.
1906  *
1907  * 2.7  int dtrace_probe_create(dtrace_provider_t id, const char *mod,
1908  *          const char *func, const char *name, int aframes, void *arg)
1909  *
1910  * 2.7.1  Overview
1911  *
1912  *   Creates a probe with specified module name, function name, and name.
1913  *
1914  * 2.7.2  Arguments and Notes
1915  *
1916  *   The first argument is the provider identifier, as returned from a
1917  *   successful call to dtrace_register().  The second, third, and fourth
1918  *   arguments are the module name, function name, and probe name,
1919  *   respectively.  Of these, module name and function name may both be NULL
1920  *   (in which case the probe is considered to be unanchored), or they may both
1921  *   be non-NULL.  The name must be non-NULL, and must point to a non-empty
1922  *   string.
1923  *
1924  *   The fifth argument is the number of artificial stack frames that will be
1925  *   found on the stack when dtrace_probe() is called for the new probe.  These
1926  *   artificial frames will be automatically be pruned should the stack() or
1927  *   stackdepth() functions be called as part of one of the probe's ECBs.  If
1928  *   the parameter doesn't add an artificial frame, this parameter should be
1929  *   zero.
1930  *
1931  *   The final argument is a probe argument that will be passed back to the
1932  *   provider when a probe-specific operation is called.  (e.g., via
1933  *   dtps_enable(), dtps_disable(), etc.)
1934  *
1935  *   Note that it is up to the provider to be sure that the probe that it
1936  *   creates does not already exist -- if the provider is unsure of the probe's
1937  *   existence, it should assure its absence with dtrace_probe_lookup() before
1938  *   calling dtrace_probe_create().
1939  *
1940  * 2.7.3  Return value
1941  *
1942  *   dtrace_probe_create() always succeeds, and always returns the identifier
1943  *   of the newly-created probe.
1944  *
1945  * 2.7.4  Caller's context
1946  *
1947  *   While dtrace_probe_create() is generally expected to be called from
1948  *   dtps_provide() and/or dtps_provide_module(), it may be called from other
1949  *   non-DTrace contexts.  Neither cpu_lock nor mod_lock may be held.
1950  *
1951  * 2.8  dtrace_id_t dtrace_probe_lookup(dtrace_provider_t id, const char *mod,
1952  *          const char *func, const char *name)
1953  *
1954  * 2.8.1  Overview
1955  *
1956  *   Looks up a probe based on provdider and one or more of module name,
1957  *   function name and probe name.
1958  *
1959  * 2.8.2  Arguments and Notes
1960  *
1961  *   The first argument is the provider identifier, as returned from a
1962  *   successful call to dtrace_register().  The second, third, and fourth
1963  *   arguments are the module name, function name, and probe name,
1964  *   respectively.  Any of these may be NULL; dtrace_probe_lookup() will return
1965  *   the identifier of the first probe that is provided by the specified
1966  *   provider and matches all of the non-NULL matching criteria.
1967  *   dtrace_probe_lookup() is generally used by a provider to be check the
1968  *   existence of a probe before creating it with dtrace_probe_create().
1969  *
1970  * 2.8.3  Return value
```



```
1971  *
1972  *   If the probe exists, returns its identifier.  If the probe does not exist,
1973  *   return DTRACE_IDNONE.
1974  *
1975  * 2.8.4  Caller's context
1976  *
1977  *   While dtrace_probe_lookup() is generally expected to be called from
1978  *   dtps_provide() and/or dtps_provide_module(), it may also be called from
1979  *   other non-DTrace contexts.  Neither cpu_lock nor mod_lock may be held.
1980  *
1981  * 2.9  void *dtrace_probe_arg(dtrace_provider_t id, dtrace_id_t probe)
1982  *
1983  * 2.9.1  Overview
1984  *
1985  *   Returns the probe argument associated with the specified probe.
1986  *
1987  * 2.9.2  Arguments and Notes
1988  *
1989  *   The first argument is the provider identifier, as returned from a
1990  *   successful call to dtrace_register().  The second argument is a probe
1991  *   identifier, as returned from dtrace_probe_lookup() or
1992  *   dtrace_probe_create().  This is useful if a probe has multiple
1993  *   provider-specific components to it:  the provider can create the probe
1994  *   once with provider-specific state, and then add to the state by looking
1995  *   up the probe based on probe identifier.
1996  *
1997  * 2.9.3  Return value
1998  *
1999  *   Returns the argument associated with the specified probe.  If the
2000  *   specified probe does not exist, or if the specified probe is not provided
2001  *   by the specified provider, NULL is returned.
2002  *
2003  * 2.9.4  Caller's context
2004  *
2005  *   While dtrace_probe_arg() is generally expected to be called from
2006  *   dtps_provide() and/or dtps_provide_module(), it may also be called from
2007  *   other non-DTrace contexts.  Neither cpu_lock nor mod_lock may be held.
2008  *
2009  * 2.10  void dtrace_probe(dtrace_id_t probe, uintptr_t arg0, uintptr_t arg1,
2010  *             uintptr_t arg2, uintptr_t arg3, uintptr_t arg4)
2011  *
2012  * 2.10.1  Overview
2013  *
2014  *   The epicenter of DTrace:  fires the specified probes with the specified
2015  *   arguments.
2016  *
2017  * 2.10.2  Arguments and Notes
2018  *
2019  *   The first argument is a probe identifier as returned by
2020  *   dtrace_probe_create() or dtrace_probe_lookup().  The second through sixth
2021  *   arguments are the values to which the D variables "arg0" through "arg4"
2022  *   will be mapped.
2023  *
2024  *   dtrace_probe() should be called whenever the specified probe has fired --
2025  *   however the provider defines it.
2026  *
2027  * 2.10.3  Return value
2028  *
2029  *   None.
2030  *
2031  * 2.10.4  Caller's context
2032  *
2033  *   dtrace_probe() may be called in virtually any context:  kernel, user,
2034  *   interrupt, high-level interrupt, with arbitrary adaptive locks held, with
2035  *   dispatcher locks held, with interrupts disabled, etc.  The only latitude
2036  *   that must be afforded to DTrace is the ability to make calls within
```

```
 *   itself (and to its in-kernel subroutines) and the ability to access
 *   arbitrary (but mapped) memory.  On some platforms, this constrains
 *   context.  For example, on UltraSPARC, dtrace_probe() cannot be called
 *   from any context in which TL is greater than zero.  dtrace_probe() may
 *   also not be called from any routine which may be called by dtrace_probe()
 *   -- which includes functions in the DTrace framework and some in-kernel
 *   DTrace subroutines.  All such functions "dtrace_"; providers that
 *   instrument the kernel arbitrarily should be sure to not instrument these
 *   routines.
 */
typedef struct dtrace_pops {
        void (*dtps_provide)(void *arg, const dtrace_probedesc_t *spec);
        void (*dtps_provide_module)(void *arg, struct modctl *mp);
        int (*dtps_enable)(void *arg, dtrace_id_t id, void *parg);
        void (*dtps_disable)(void *arg, dtrace_id_t id, void *parg);
        void (*dtps_suspend)(void *arg, dtrace_id_t id, void *parg);
        void (*dtps_resume)(void *arg, dtrace_id_t id, void *parg);
        void (*dtps_getargdesc)(void *arg, dtrace_id_t id, void *parg,
            dtrace_argdesc_t *desc);
        uint64_t (*dtps_getargval)(void *arg, dtrace_id_t id, void *parg,
            int argno, int aframes);
        int (*dtps_mode)(void *arg, dtrace_id_t id, void *parg);
        void (*dtps_destroy)(void *arg, dtrace_id_t id, void *parg);
} dtrace_pops_t;

#define DTRACE_MODE_KERNEL                      0x01
#define DTRACE_MODE_USER                        0x02
#define DTRACE_MODE_NOPRIV_DROP                 0x10
#define DTRACE_MODE_NOPRIV_RESTRICT             0x20
#define DTRACE_MODE_LIMITEDPRIV_RESTRICT        0x40
#endif /* ! codereview */

typedef uintptr_t       dtrace_provider_id_t;

extern int dtrace_register(const char *, const dtrace_pattr_t *, uint32_t,
    cred_t *, const dtrace_pops_t *, void *, dtrace_provider_id_t *);
extern int dtrace_unregister(dtrace_provider_id_t);
extern int dtrace_condense(dtrace_provider_id_t);
extern void dtrace_invalidate(dtrace_provider_id_t);
extern dtrace_id_t dtrace_probe_lookup(dtrace_provider_id_t, const char *,
    const char *, const char *);
extern dtrace_id_t dtrace_probe_create(dtrace_provider_id_t, const char *,
    const char *, const char *, int, void *);
extern void *dtrace_probe_arg(dtrace_provider_id_t, dtrace_id_t);
extern void dtrace_probe(dtrace_id_t, uintptr_t arg0, uintptr_t arg1,
    uintptr_t arg2, uintptr_t arg3, uintptr_t arg4);

/*
 * DTrace Meta Provider API
 *
 * The following functions are implemented by the DTrace framework and are
 * used to implement meta providers.  Meta providers plug into the DTrace
 * framework and are used to instantiate new providers on the fly.  At
 * present, there is only one type of meta provider and only one meta
 * provider may be registered with the DTrace framework at a time.  The
 * sole meta provider type provides user-land static tracing facilities
 * by taking meta probe descriptions and adding a corresponding provider
 * into the DTrace framework.
 *
 * 1 Framework-to-Provider
 *
 * 1.1 Overview
 *
 * The Framework-to-Provider API is represented by the dtrace_mops structure
 * that the meta provider passes to the framework when registering itself as
 * a meta provider.  This structure consists of the following members:
```



```
 *
 *   dtms_create_probe()        <-- Add a new probe to a created provider
 *   dtms_provide_pid()         <-- Create a new provider for a given process
 *   dtms_remove_pid()          <-- Remove a previously created provider
 *
 * 1.2  void dtms_create_probe(void *arg, void *parg,
 *           dtrace_helper_probedesc_t *probedesc);
 *
 * 1.2.1  Overview
 *
 *   Called by the DTrace framework to create a new probe in a provider
 *   created by this meta provider.
 *
 * 1.2.2  Arguments and notes
 *
 *   The first argument is the cookie as passed to dtrace_meta_register().
 *   The second argument is the provider cookie for the associated provider;
 *   this is obtained from the return value of dtms_provide_pid(). The third
 *   argument is the helper probe description.
 *
 * 1.2.3  Return value
 *
 *   None
 *
 * 1.2.4  Caller's context
 *
 *   dtms_create_probe() is called from either ioctl() or module load context.
 *   The DTrace framework is locked in such a way that meta providers may not
 *   register or unregister. This means that the meta provider cannot call
 *   dtrace_meta_register() or dtrace_meta_unregister(). However, the context is
 *   such that the provider may (and is expected to) call provider-related
 *   DTrace provider APIs including dtrace_probe_create().
 *
 * 1.3  void *dtms_provide_pid(void *arg, dtrace_meta_provider_t *mprov,
 *            pid_t pid)
 *
 * 1.3.1  Overview
 *
 *   Called by the DTrace framework to instantiate a new provider given the
 *   description of the provider and probes in the mprov argument. The
 *   meta provider should call dtrace_register() to insert the new provider
 *   into the DTrace framework.
 *
 * 1.3.2  Arguments and notes
 *
 *   The first argument is the cookie as passed to dtrace_meta_register().
 *   The second argument is a pointer to a structure describing the new
 *   helper provider. The third argument is the process identifier for
 *   process associated with this new provider. Note that the name of the
 *   provider as passed to dtrace_register() should be the contatenation of
 *   the dtmpb_provname member of the mprov argument and the processs
 *   identifier as a string.
 *
 * 1.3.3  Return value
 *
 *   The cookie for the provider that the meta provider creates. This is
 *   the same value that it passed to dtrace_register().
 *
 * 1.3.4  Caller's context
 *
 *   dtms_provide_pid() is called from either ioctl() or module load context.
 *   The DTrace framework is locked in such a way that meta providers may not
 *   register or unregister. This means that the meta provider cannot call
 *   dtrace_meta_register() or dtrace_meta_unregister(). However, the context
 *   is such that the provider may -- and is expected to --  call
 *   provider-related DTrace provider APIs including dtrace_register().
```

```
 *
 * 1.4  void dtms_remove_pid(void *arg, dtrace_meta_provider_t *mprov,
 *           pid_t pid)
 *
 * 1.4.1  Overview
 *
 *   Called by the DTrace framework to remove a provider that had previously
 *   been instantiated via the dtms_provide_pid() entry point. The meta
 *   provider need not remove the provider immediately, but this entry
 *   point indicates that the provider should be removed as soon as possible
 *   using the dtrace_unregister() API.
 *
 * 1.4.2  Arguments and notes
 *
 *   The first argument is the cookie as passed to dtrace_meta_register().
 *   The second argument is a pointer to a structure describing the helper
 *   provider. The third argument is the process identifier for process
 *   associated with this new provider.
 *
 * 1.4.3  Return value
 *
 *   None
 *
 * 1.4.4  Caller's context
 *
 *   dtms_remove_pid() is called from either ioctl() or exit() context.
 *   The DTrace framework is locked in such a way that meta providers may not
 *   register or unregister. This means that the meta provider cannot call
 *   dtrace_meta_register() or dtrace_meta_unregister(). However, the context
 *   is such that the provider may -- and is expected to -- call
 *   provider-related DTrace provider APIs including dtrace_unregister().
 */
typedef struct dtrace_helper_probedesc {
        char *dthpb_mod;                        /* probe module */
        char *dthpb_func;                       /* probe function */
        char *dthpb_name;                       /* probe name */
        uint64_t dthpb_base;                    /* base address */
        uint32_t *dthpb_offs;                   /* offsets array */
        uint32_t *dthpb_enoffs;                 /* is-enabled offsets array */
        uint32_t dthpb_noffs;                   /* offsets count */
        uint32_t dthpb_nenoffs;                 /* is-enabled offsets count */
        uint8_t *dthpb_args;                    /* argument mapping array */
        uint8_t dthpb_xargc;                    /* translated argument count */
        uint8_t dthpb_nargc;                    /* native argument count */
        char *dthpb_xtypes;                     /* translated types strings */
        char *dthpb_ntypes;                     /* native types strings */
} dtrace_helper_probedesc_t;

typedef struct dtrace_helper_provdesc {
        char *dthpv_provname;                   /* provider name */
        dtrace_pattr_t dthpv_pattr;             /* stability attributes */
} dtrace_helper_provdesc_t;

typedef struct dtrace_mops {
        void (*dtms_create_probe)(void *, void *, dtrace_helper_probedesc_t *);
        void *(*dtms_provide_pid)(void *, dtrace_helper_provdesc_t *, pid_t);
        void (*dtms_remove_pid)(void *, dtrace_helper_provdesc_t *, pid_t);
} dtrace_mops_t;

typedef uintptr_t       dtrace_meta_provider_id_t;

extern int dtrace_meta_register(const char *, const dtrace_mops_t *, void *,
    dtrace_meta_provider_id_t *);
extern int dtrace_meta_unregister(dtrace_meta_provider_id_t);

/*
```



```
 * DTrace Kernel Hooks
 *
 * The following functions are implemented by the base kernel and form a set of
 * hooks used by the DTrace framework.  DTrace hooks are implemented in either
 * uts/common/os/dtrace_subr.c, an ISA-specific assembly file, or in a
 * uts/<platform>/os/dtrace_subr.c corresponding to each hardware platform.
 */

typedef enum dtrace_vtime_state {
        DTRACE_VTIME_INACTIVE = 0,      /* No DTrace, no TNF */
        DTRACE_VTIME_ACTIVE,            /* DTrace virtual time, no TNF */
        DTRACE_VTIME_INACTIVE_TNF,      /* No DTrace, TNF active */
        DTRACE_VTIME_ACTIVE_TNF         /* DTrace virtual time _and_ TNF */
} dtrace_vtime_state_t;

extern dtrace_vtime_state_t dtrace_vtime_active;
extern void dtrace_vtime_switch(kthread_t *next);
extern void dtrace_vtime_enable_tnf(void);
extern void dtrace_vtime_disable_tnf(void);
extern void dtrace_vtime_enable(void);
extern void dtrace_vtime_disable(void);

struct regs;

extern int (*dtrace_pid_probe_ptr)(struct regs *);
extern int (*dtrace_return_probe_ptr)(struct regs *);
extern void (*dtrace_fasttrap_fork_ptr)(proc_t *, proc_t *);
extern void (*dtrace_fasttrap_exec_ptr)(proc_t *);
extern void (*dtrace_fasttrap_exit_ptr)(proc_t *);
extern void dtrace_fasttrap_fork(proc_t *, proc_t *);

typedef uintptr_t dtrace_icookie_t;
typedef void (*dtrace_xcall_t)(void *);

extern dtrace_icookie_t dtrace_interrupt_disable(void);
extern void dtrace_interrupt_enable(dtrace_icookie_t);

extern void dtrace_membar_producer(void);
extern void dtrace_membar_consumer(void);

extern void (*dtrace_cpu_init)(processorid_t);
extern void (*dtrace_modload)(struct modctl *);
extern void (*dtrace_modunload)(struct modctl *);
extern void (*dtrace_helpers_cleanup)();
extern void (*dtrace_helpers_fork)(proc_t *parent, proc_t *child);
extern void (*dtrace_cpustart_init)();
extern void (*dtrace_cpustart_fini)();
extern void (*dtrace_closef)();
#endif /* ! codereview */

extern void (*dtrace_debugger_init)();
extern void (*dtrace_debugger_fini)();
extern dtrace_cacheid_t dtrace_predcache_id;

extern hrtime_t dtrace_gethrtime(void);
extern void dtrace_sync(void);
extern void dtrace_toxic_ranges(void (*)(uintptr_t, uintptr_t));
extern void dtrace_xcall(processorid_t, dtrace_xcall_t, void *);
extern void dtrace_vpanic(const char *, __va_list);
extern void dtrace_panic(const char *, ...);

extern int dtrace_safe_defer_signal(void);
extern void dtrace_safe_synchronous_signal(void);

extern int dtrace_mach_aframes(void);
```

```
2301 #if defined(__i386) || defined(__amd64)
2302 extern int dtrace_instr_size(uchar_t *instr);
2303 extern int dtrace_instr_size_isa(uchar_t *, model_t, int *);
2304 extern void dtrace_invop_add(int (*)(uintptr_t, uintptr_t *, uintptr_t));
2305 extern void dtrace_invop_remove(int (*)(uintptr_t, uintptr_t *, uintptr_t));
2306 extern void dtrace_invop_callsite(void);
2307 #endif

2309 #ifdef __sparc
2310 extern int dtrace_blksuword32(uintptr_t, uint32_t *, int);
2311 extern void dtrace_getfsr(uint64_t *);
2312 #endif

2314 #define DTRACE_CPUFLAG_ISSET(flag) \
2315         (cpu_core[CPU->cpu_id].cpuc_dtrace_flags & (flag))

2317 #define DTRACE_CPUFLAG_SET(flag) \
2318         (cpu_core[CPU->cpu_id].cpuc_dtrace_flags |= (flag))

2320 #define DTRACE_CPUFLAG_CLEAR(flag) \
2321         (cpu_core[CPU->cpu_id].cpuc_dtrace_flags &= ~(flag))

2323 #endif /* _KERNEL */

2325 #endif  /* _ASM */

2327 #if defined(__i386) || defined(__amd64)

2329 #define DTRACE_INVOP_PUSHL_EBP          1
2330 #define DTRACE_INVOP_POPL_EBP           2
2331 #define DTRACE_INVOP_LEAVE              3
2332 #define DTRACE_INVOP_NOP                4
2333 #define DTRACE_INVOP_RET                5

2335 #endif

2337 #ifdef  __cplusplus
2338 }
2339 #endif

2341 #endif  /* _SYS_DTRACE_H */
```

```
**********************************************************
   64642 Tue Jan 14 16:50:05 2014
new/usr/src/uts/common/sys/dtrace_impl.h
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 887 /*
 888  * DTrace Machine State
 889  *
 890  * In the process of processing a fired probe, DTrace needs to track and/or
 891  * cache some per-CPU state associated with that particular firing.  This is
 892  * state that is always discarded after the probe firing has completed, and
 893  * much of it is not specific to any DTrace consumer, remaining valid across
 894  * all ECBs.  This state is tracked in the dtrace_mstate structure.
 895  */
 896 #define DTRACE_MSTATE_ARGS              0x00000001
 897 #define DTRACE_MSTATE_PROBE             0x00000002
 898 #define DTRACE_MSTATE_EPID              0x00000004
 899 #define DTRACE_MSTATE_TIMESTAMP         0x00000008
 900 #define DTRACE_MSTATE_STACKDEPTH        0x00000010
 901 #define DTRACE_MSTATE_CALLER            0x00000020
 902 #define DTRACE_MSTATE_IPL               0x00000040
 903 #define DTRACE_MSTATE_FLTOFFS           0x00000080
 904 #define DTRACE_MSTATE_WALLTIMESTAMP     0x00000100
 905 #define DTRACE_MSTATE_USTACKDEPTH       0x00000200
 906 #define DTRACE_MSTATE_UCALLER           0x00000400

 908 typedef struct dtrace_mstate {
 909         uintptr_t dtms_scratch_base;            /* base of scratch space */
 910         uintptr_t dtms_scratch_ptr;             /* current scratch pointer */
 911         size_t dtms_scratch_size;               /* scratch size */
 912         uint32_t dtms_present;                  /* variables that are present */
 913         uint64_t dtms_arg[5];                   /* cached arguments */
 914         dtrace_epid_t dtms_epid;                /* current EPID */
 915         uint64_t dtms_timestamp;                /* cached timestamp */
 916         hrtime_t dtms_walltimestamp;            /* cached wall timestamp */
 917         int dtms_stackdepth;                    /* cached stackdepth */
 918         int dtms_ustackdepth;                   /* cached ustackdepth */
 919         struct dtrace_probe *dtms_probe;        /* current probe */
 920         uintptr_t dtms_caller;                  /* cached caller */
 921         uint64_t dtms_ucaller;                  /* cached user-level caller */
 922         int dtms_ipl;                           /* cached interrupt pri lev */
 923         int dtms_fltoffs;                       /* faulting DIFO offset */
 924         uintptr_t dtms_strtok;                  /* saved strtok() pointer */
 925         uint32_t dtms_access;                   /* memory access rights */
 926         dtrace_difo_t *dtms_difo;               /* current dif object */
 927         file_t *dtms_getf;                      /* cached rval of getf() */
 928 #endif /* ! codereview */
 929 } dtrace_mstate_t;

 931 #define DTRACE_COND_OWNER       0x1
 932 #define DTRACE_COND_USERMODE    0x2
 933 #define DTRACE_COND_ZONEOWNER   0x4

 935 #define DTRACE_PROBEKEY_MAXDEPTH        8       /* max glob recursion depth */

 937 /*
 938  * Access flag used by dtrace_mstate.dtms_access.
 939  */
 940 #define DTRACE_ACCESS_KERNEL    0x1             /* the priv to read kmem */
 941 #define DTRACE_ACCESS_PROC      0x2             /* the priv for proc state */
```



```
 942 #define DTRACE_ACCESS_ARGS      0x4             /* the priv to examine args */

 944 /*
 945  * DTrace Activity
 946  *
 947  * Each DTrace consumer is in one of several states, which (for purposes of
 948  * avoiding yet-another overloading of the noun "state") we call the current
 949  * _activity_.  The activity transitions on dtrace_go() (from DTRACIOCGO), on
 950  * dtrace_stop() (from DTRACIOCSTOP) and on the exit() action.  Activities may
 951  * only transition in one direction; the activity transition diagram is a
 952  * directed acyclic graph.  The activity transition diagram is as follows:
 953  *
 954  *
 955  * +----------+                    +--------+                    +--------+
 956  * | INACTIVE |------------------->| WARMUP |------------------->| ACTIVE |
 957  * +----------+   dtrace_go(),    +--------+   dtrace_go(),    +--------+
 958  *                before BEGIN                  after BEGIN       |  |  |
 959  *                                              |                 |  |  |
 960  *                      exit() action           |                 |  |  |
 961  *                     from BEGIN ECB           |                 |  |  |
 962  *                                              |                 |  |  |
 963  *                                              v                 |  |  |
 964  *                                +----------+     exit() action  |  |  |
 965  * +------------------------------| DRAINING |<-------------------+  |  |
 966  * |                              +----------+                       |  |
 967  * |                                 |                               |  |
 968  * |                   dtrace_stop(), |                               |  |
 969  * |                     before END   |                               |  |
 970  * |                                 |                               |  |
 971  * |                                 v                               |  |
 972  * |  +---------+                 +----------+                       |  |
 973  * |  | STOPPED |<----------------| COOLDOWN |<----------------------+  |
 974  * |  +---------+  dtrace_stop(), +----------+     dtrace_stop(),       |
 975  * |                 after END                      before END         |
 976  * |                                                                   |
 977  * |                               +--------+                          |
 978  * +------------------------------>| KILLED |<-------------------------+
 979  *       deadman timeout or        +--------+     deadman timeout or
 980  *         killed consumer                         killed consumer
 981  *
 982  * Note that once a DTrace consumer has stopped tracing, there is no way to
 983  * restart it; if a DTrace consumer wishes to restart tracing, it must reopen
 984  * the DTrace pseudodevice.
 985  */
 986 typedef enum dtrace_activity {
 987         DTRACE_ACTIVITY_INACTIVE = 0,           /* not yet running */
 988         DTRACE_ACTIVITY_WARMUP,                 /* while starting */
 989         DTRACE_ACTIVITY_ACTIVE,                 /* running */
 990         DTRACE_ACTIVITY_DRAINING,               /* before stopping */
 991         DTRACE_ACTIVITY_COOLDOWN,               /* while stopping */
 992         DTRACE_ACTIVITY_STOPPED,                /* after stopping */
 993         DTRACE_ACTIVITY_KILLED                  /* killed */
 994 } dtrace_activity_t;

 996 /*
 997  * DTrace Helper Implementation
 998  *
 999  * A description of the helper architecture may be found in <sys/dtrace.h>.
1000  * Each process contains a pointer to its helpers in its p_dtrace_helpers
1001  * member.  This is a pointer to a dtrace_helpers structure, which contains an
1002  * array of pointers to dtrace_helper structures, helper variable state (shared
1003  * among a process's helpers) and a generation count.  (The generation count is
1004  * used to provide an identifier when a helper is added so that it may be
1005  * subsequently removed.)  The dtrace_helper structure is self-explanatory,
1006  * containing pointers to the objects needed to execute the helper.  Note that
1007  * helpers are _duplicated_ across fork(2), and destroyed on exec(2).  No more
```

```
 * than dtrace_helpers_max are allowed per-process.
 */
#define DTRACE_HELPER_ACTION_USTACK     0
#define DTRACE_NHELPER_ACTIONS          1

typedef struct dtrace_helper_action {
        int dtha_generation;                    /* helper action generation */
        int dtha_nactions;                      /* number of actions */
        dtrace_difo_t *dtha_predicate;          /* helper action predicate */
        dtrace_difo_t **dtha_actions;           /* array of actions */
        struct dtrace_helper_action *dtha_next; /* next helper action */
} dtrace_helper_action_t;

typedef struct dtrace_helper_provider {
        int dthp_generation;                    /* helper provider generation */
        uint32_t dthp_ref;                      /* reference count */
        dof_helper_t dthp_prov;                 /* DOF w/ provider and probes */
} dtrace_helper_provider_t;

typedef struct dtrace_helpers {
        dtrace_helper_action_t **dthps_actions; /* array of helper actions */
        dtrace_vstate_t dthps_vstate;           /* helper action var. state */
        dtrace_helper_provider_t **dthps_provs; /* array of providers */
        uint_t dthps_nprovs;                    /* count of providers */
        uint_t dthps_maxprovs;                  /* provider array size */
        int dthps_generation;                   /* current generation */
        pid_t dthps_pid;                        /* pid of associated proc */
        int dthps_deferred;                     /* helper in deferred list */
        struct dtrace_helpers *dthps_next;      /* next pointer */
        struct dtrace_helpers *dthps_prev;      /* prev pointer */
} dtrace_helpers_t;

/*
 * DTrace Helper Action Tracing
 *
 * Debugging helper actions can be arduous.  To ease the development and
 * debugging of helpers, DTrace contains a tracing-framework-within-a-tracing-
 * framework: helper tracing.  If dtrace_helptrace_enabled is non-zero (which
 * it is by default on DEBUG kernels), all helper activity will be traced to a
 * global, in-kernel ring buffer.  Each entry includes a pointer to the specific
 * helper, the location within the helper, and a trace of all local variables.
 * The ring buffer may be displayed in a human-readable format with the
 * ::dtrace_helptrace mdb(1) dcmd.
 */
#define DTRACE_HELPTRACE_NEXT   (-1)
#define DTRACE_HELPTRACE_DONE   (-2)
#define DTRACE_HELPTRACE_ERR    (-3)

typedef struct dtrace_helptrace {
        dtrace_helper_action_t  *dtht_helper;   /* helper action */
        int dtht_where;                         /* where in helper action */
        int dtht_nlocals;                       /* number of locals */
        int dtht_fault;                         /* type of fault (if any) */
        int dtht_fltoffs;                       /* DIF offset */
        uint64_t dtht_illval;                   /* faulting value */
        uint64_t dtht_locals[1];                /* local variables */
} dtrace_helptrace_t;

/*
 * DTrace Credentials
 *
 * In probe context, we have limited flexibility to examine the credentials
 * of the DTrace consumer that created a particular enabling.  We use
 * the Least Privilege interfaces to cache the consumer's cred pointer and
 * some facts about that credential in a dtrace_cred_t structure.  These
 * can limit the consumer's breadth of visibility and what actions the
```



```
 * consumer may take.
 */
#define DTRACE_CRV_ALLPROC              0x01
#define DTRACE_CRV_KERNEL               0x02
#define DTRACE_CRV_ALLZONE              0x04

#define DTRACE_CRV_ALL          (DTRACE_CRV_ALLPROC | DTRACE_CRV_KERNEL | \
        DTRACE_CRV_ALLZONE)

#define DTRACE_CRA_PROC                         0x0001
#define DTRACE_CRA_PROC_CONTROL                 0x0002
#define DTRACE_CRA_PROC_DESTRUCTIVE_ALLUSER     0x0004
#define DTRACE_CRA_PROC_DESTRUCTIVE_ALLZONE     0x0008
#define DTRACE_CRA_PROC_DESTRUCTIVE_CREDCHG     0x0010
#define DTRACE_CRA_KERNEL                       0x0020
#define DTRACE_CRA_KERNEL_DESTRUCTIVE           0x0040

#define DTRACE_CRA_ALL          (DTRACE_CRA_PROC | \
        DTRACE_CRA_PROC_CONTROL | \
        DTRACE_CRA_PROC_DESTRUCTIVE_ALLUSER | \
        DTRACE_CRA_PROC_DESTRUCTIVE_ALLZONE | \
        DTRACE_CRA_PROC_DESTRUCTIVE_CREDCHG | \
        DTRACE_CRA_KERNEL | \
        DTRACE_CRA_KERNEL_DESTRUCTIVE)

typedef struct dtrace_cred {
        cred_t                  *dcr_cred;
        uint8_t                 dcr_destructive;
        uint8_t                 dcr_visible;
        uint16_t                dcr_action;
} dtrace_cred_t;

/*
 * DTrace Consumer State
 *
 * Each DTrace consumer has an associated dtrace_state structure that contains
 * its in-kernel DTrace state -- including options, credentials, statistics and
 * pointers to ECBs, buffers, speculations and formats.  A dtrace_state
 * structure is also allocated for anonymous enablings.  When anonymous state
 * is grabbed, the grabbing consumers dts_anon pointer is set to the grabbed
 * dtrace_state structure.
 */
struct dtrace_state {
        dev_t dts_dev;                          /* device */
        int dts_necbs;                          /* total number of ECBs */
        dtrace_ecb_t **dts_ecbs;                /* array of ECBs */
        dtrace_epid_t dts_epid;                 /* next EPID to allocate */
        size_t dts_needed;                      /* greatest needed space */
        struct dtrace_state *dts_anon;          /* anon. state, if grabbed */
        dtrace_activity_t dts_activity;         /* current activity */
        dtrace_vstate_t dts_vstate;             /* variable state */
        dtrace_buffer_t *dts_buffer;            /* principal buffer */
        dtrace_buffer_t *dts_aggbuffer;         /* aggregation buffer */
        dtrace_speculation_t *dts_speculations; /* speculation array */
        int dts_nspeculations;                  /* number of speculations */
        int dts_naggregations;                  /* number of aggregations */
        dtrace_aggregation_t **dts_aggregations; /* aggregation array */
        vmem_t *dts_aggid_arena;                /* arena for aggregation IDs */
        uint64_t dts_errors;                    /* total number of errors */
        uint32_t dts_speculations_busy;         /* number of spec. busy */
        uint32_t dts_speculations_unavail;      /* number of spec unavail */
        uint32_t dts_stkstroverflows;           /* stack string tab overflows */
        uint32_t dts_dblerrors;                 /* errors in ERROR probes */
        uint32_t dts_reserve;                   /* space reserved for END */
        hrtime_t dts_laststatus;                /* time of last status */
        cyclic_id_t dts_cleaner;                /* cleaning cyclic */
```

```
1140         cyclic_id_t dts_deadman;               /* deadman cyclic */
1141         hrtime_t dts_alive;                    /* time last alive */
1142         char dts_speculates;                   /* boolean: has speculations */
1143         char dts_destructive;                  /* boolean: has dest. actions */
1144         int dts_nformats;                      /* number of formats */
1145         char **dts_formats;                    /* format string array */
1146         dtrace_optval_t dts_options[DTRACEOPT_MAX]; /* options */
1147         dtrace_cred_t dts_cred;                /* credentials */
1148         size_t dts_nretained;                  /* number of retained enabs */
1149         int dts_getf;                          /* number of getf() calls */
1150 #endif /* ! codereview */
1151 };

1153 struct dtrace_provider {
1154         dtrace_pattr_t dtpv_attr;              /* provider attributes */
1155         dtrace_ppriv_t dtpv_priv;              /* provider privileges */
1156         dtrace_pops_t dtpv_pops;               /* provider operations */
1157         char *dtpv_name;                       /* provider name */
1158         void *dtpv_arg;                        /* provider argument */
1159         hrtime_t dtpv_defunct;                 /* when made defunct */
1160         struct dtrace_provider *dtpv_next;     /* next provider */
1161 };

1163 struct dtrace_meta {
1164         dtrace_mops_t dtm_mops;                /* meta provider operations */
1165         char *dtm_name;                        /* meta provider name */
1166         void *dtm_arg;                         /* meta provider user arg */
1167         uint64_t dtm_count;                    /* no. of associated provs. */
1168 };

1170 /*
1171  * DTrace Enablings
1172  *
1173  * A dtrace_enabling structure is used to track a collection of ECB
1174  * descriptions -- before they have been turned into actual ECBs.  This is
1175  * created as a result of DOF processing, and is generally used to generate
1176  * ECBs immediately thereafter.  However, enablings are also generally
1177  * retained should the probes they describe be created at a later time; as
1178  * each new module or provider registers with the framework, the retained
1179  * enablings are reevaluated, with any new match resulting in new ECBs.  To
1180  * prevent probes from being matched more than once, the enabling tracks the
1181  * last probe generation matched, and only matches probes from subsequent
1182  * generations.
1183  */
1184 typedef struct dtrace_enabling {
1185         dtrace_ecbdesc_t **dten_desc;          /* all ECB descriptions */
1186         int dten_ndesc;                        /* number of ECB descriptions */
1187         int dten_maxdesc;                      /* size of ECB array */
1188         dtrace_vstate_t *dten_vstate;          /* associated variable state */
1189         dtrace_genid_t dten_probegen;          /* matched probe generation */
1190         dtrace_ecbdesc_t *dten_current;        /* current ECB description */
1191         int dten_error;                        /* current error value */
1192         int dten_primed;                       /* boolean: set if primed */
1193         struct dtrace_enabling *dten_prev;     /* previous enabling */
1194         struct dtrace_enabling *dten_next;     /* next enabling */
1195 } dtrace_enabling_t;

1197 /*
1198  * DTrace Anonymous Enablings
1199  *
1200  * Anonymous enablings are DTrace enablings that are not associated with a
1201  * controlling process, but rather derive their enabling from DOF stored as
1202  * properties in the dtrace.conf file.  If there is an anonymous enabling, a
1203  * DTrace consumer state and enabling are created on attach.  The state may be
1204  * subsequently grabbed by the first consumer specifying the "grabanon"
1205  * option.  As long as an anonymous DTrace enabling exists, dtrace(7D) will
```



```
1206  * refuse to unload.
1207  */
1208 typedef struct dtrace_anon {
1209         dtrace_state_t *dta_state;             /* DTrace consumer state */
1210         dtrace_enabling_t *dta_enabling;       /* pointer to enabling */
1211         processorid_t dta_beganon;             /* which CPU BEGIN ran on */
1212 } dtrace_anon_t;

1214 /*
1215  * DTrace Error Debugging
1216  */
1217 #ifdef DEBUG
1218 #define DTRACE_ERRDEBUG
1219 #endif

1221 #ifdef DTRACE_ERRDEBUG

1223 typedef struct dtrace_errhash {
1224         const char      *dter_msg;      /* error message */
1225         int             dter_count;     /* number of times seen */
1226 } dtrace_errhash_t;

1228 #define DTRACE_ERRHASHSZ        256     /* must be > number of err msgs */

1230 #endif  /* DTRACE_ERRDEBUG */

1232 /*
1233  * DTrace Toxic Ranges
1234  *
1235  * DTrace supports safe loads from probe context; if the address turns out to
1236  * be invalid, a bit will be set by the kernel indicating that DTrace
1237  * encountered a memory error, and DTrace will propagate the error to the user
1238  * accordingly.  However, there may exist some regions of memory in which an
1239  * arbitrary load can change system state, and from which it is impossible to
1240  * recover from such a load after it has been attempted.  Examples of this may
1241  * include memory in which programmable I/O registers are mapped (for which a
1242  * read may have some implications for the device) or (in the specific case of
1243  * UltraSPARC-I and -II) the virtual address hole.  The platform is required
1244  * to make DTrace aware of these toxic ranges; DTrace will then check that
1245  * target addresses are not in a toxic range before attempting to issue a
1246  * safe load.
1247  */
1248 typedef struct dtrace_toxrange {
1249         uintptr_t       dtt_base;               /* base of toxic range */
1250         uintptr_t       dtt_limit;              /* limit of toxic range */
1251 } dtrace_toxrange_t;

1253 extern uint64_t dtrace_getarg(int, int);
1254 extern greg_t dtrace_getfp(void);
1255 extern int dtrace_getipl(void);
1256 extern uintptr_t dtrace_caller(int);
1257 extern uint32_t dtrace_cas32(uint32_t *, uint32_t, uint32_t);
1258 extern void *dtrace_casptr(void *, void *, void *);
1259 extern void dtrace_copyin(uintptr_t, uintptr_t, size_t, volatile uint16_t *);
1260 extern void dtrace_copyinstr(uintptr_t, uintptr_t, size_t, volatile uint16_t *);
1261 extern void dtrace_copyout(uintptr_t, uintptr_t, size_t, volatile uint16_t *);
1262 extern void dtrace_copyoutstr(uintptr_t, uintptr_t, size_t,
1263     volatile uint16_t *);
1264 extern void dtrace_getpcstack(pc_t *, int, int, uint32_t *);
1265 extern ulong_t dtrace_getreg(struct regs *, uint_t);
1266 extern uint64_t dtrace_getvmreg(uint_t, volatile uint16_t *);
1267 extern int dtrace_getstackdepth(int);
1268 extern void dtrace_getupcstack(uint64_t *, int);
1269 extern void dtrace_getufpstack(uint64_t *, uint64_t *, int);
1270 extern int dtrace_getustackdepth(void);
1271 extern uintptr_t dtrace_fulword(void *);
```

```
1272 extern uint8_t dtrace_fuword8(void *);
1273 extern uint16_t dtrace_fuword16(void *);
1274 extern uint32_t dtrace_fuword32(void *);
1275 extern uint64_t dtrace_fuword64(void *);
1276 extern void dtrace_probe_error(dtrace_state_t *, dtrace_epid_t, int, int,
1277     int, uintptr_t);
1278 extern int dtrace_assfail(const char *, const char *, int);
1279 extern int dtrace_attached(void);
1280 extern hrtime_t dtrace_gethrestime();

1282 #ifdef __sparc
1283 extern void dtrace_flush_windows(void);
1284 extern void dtrace_flush_user_windows(void);
1285 extern uint_t dtrace_getotherwin(void);
1286 extern uint_t dtrace_getfprs(void);
1287 #else
1288 extern void dtrace_copy(uintptr_t, uintptr_t, size_t);
1289 extern void dtrace_copystr(uintptr_t, uintptr_t, size_t, volatile uint16_t *);
1290 #endif

1292 /*
1293  * DTrace Assertions
1294  *
1295  * DTrace calls ASSERT from probe context.  To assure that a failed ASSERT
1296  * does not induce a markedly more catastrophic failure (e.g., one from which
1297  * a dump cannot be gleaned), DTrace must define its own ASSERT to be one that
1298  * may safely be called from probe context.  This header file must thus be
1299  * included by any DTrace component that calls ASSERT from probe context, and
1300  * _only_ by those components.  (The only exception to this is kernel
1301  * debugging infrastructure at user-level that doesn't depend on calling
1302  * ASSERT.)
1303  */
1304 #undef ASSERT
1305 #ifdef DEBUG
1306 #define ASSERT(EX)      ((void)((EX) || \
1307                         dtrace_assfail(#EX, __FILE__, __LINE__)))
1308 #else
1309 #define ASSERT(X)       ((void)0)
1310 #endif

1312 #ifdef  __cplusplus
1313 }
1314 #endif

1316 #endif /* _SYS_DTRACE_IMPL_H */
```

```
**********************************************************
    2764 Tue Jan 14 16:50:05 2014
new/usr/src/uts/common/sys/sdt_impl.h
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

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

#endif /* ! codereview */
#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 */
#endif /* ! codereview */
        dtrace_provider_id_t    sdtp_id;        /* provider ID */
} sdt_provider_t;
```



```
extern sdt_provider_t sdt_providers[];          /* array of providers */

typedef struct sdt_probe {
        sdt_provider_t  *sdp_provider;          /* provider */
        char            *sdp_name;              /* name of probe */
        int             sdp_namelen;            /* length of allocated name */
        dtrace_id_t     sdp_id;                 /* probe ID */
        struct modctl   *sdp_ctl;               /* modctl for module */
        int             sdp_loadcnt;            /* load count for module */
        int             sdp_primary;            /* non-zero if primary mod */
        sdt_instr_t     *sdp_patchpoint;        /* patch point */
        sdt_instr_t     sdp_patchval;           /* instruction to patch */
        sdt_instr_t     sdp_savedval;           /* saved instruction value */
        struct sdt_probe *sdp_next;             /* next probe */
        struct sdt_probe *sdp_hashnext;         /* next on hash */
} sdt_probe_t;

typedef struct sdt_argdesc {
        const char *sda_provider;               /* provider for arg */
        const char *sda_name;                   /* name of probe */
        const int sda_ndx;                      /* argument index */
        const int sda_mapping;                  /* mapping of argument */
        const char *sda_native;                 /* native type of argument */
        const char *sda_xlate;                  /* translated type of arg */
} sdt_argdesc_t;

extern void sdt_getargdesc(void *, dtrace_id_t, void *, dtrace_argdesc_t *);
extern int sdt_mode(void *, dtrace_id_t, void *);
#endif /* ! codereview */

#ifdef  __cplusplus
}
#endif

#endif  /* _SYS_SDT_IMPL_H */
```

```
**********************************************************
   24110 Tue Jan 14 16:50:06 2014
new/usr/src/uts/common/sys/zone.h
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
_____unchanged_portion_omitted_

 378 struct cpucap;

 380 typedef struct zone {
 381         /*
 382          * zone_name is never modified once set.
 383          */
 384         char            *zone_name;     /* zone's configuration name */
 385         /*
 386          * zone_nodename and zone_domain are never freed once allocated.
 387          */
 388         char            *zone_nodename; /* utsname.nodename equivalent */
 389         char            *zone_domain;   /* srpc_domain equivalent */
 390         /*
 391          * zone_hostid is used for per-zone hostid emulation.
 392          * Currently it isn't modified after it's set (so no locks protect
 393          * accesses), but that might have to change when we allow
 394          * administrators to change running zones' properties.
 395          *
 396          * The global zone's zone_hostid must always be HW_INVALID_HOSTID so
 397          * that zone_get_hostid() will function correctly.
 398          */
 399         uint32_t        zone_hostid;    /* zone's hostid, HW_INVALID_HOSTID */
 400                                         /* if not emulated */
 401         /*
 402          * zone_lock protects the following fields of a zone_t:
 403          *      zone_ref
 404          *      zone_cred_ref
 405          *      zone_subsys_ref
 406          *      zone_ref_list
 407          *      zone_ntasks
 408          *      zone_flags
 409          *      zone_zsd
 410          *      zone_pfexecd
 411          */
 412         kmutex_t        zone_lock;
 413         /*
 414          * zone_linkage is the zone's linkage into the active or
 415          * death-row list.  The field is protected by zonehash_lock.
 416          */
 417         list_node_t     zone_linkage;
 418         zoneid_t        zone_id;        /* ID of zone */
 419         uint_t          zone_ref;       /* count of zone_hold()s on zone */
 420         uint_t          zone_cred_ref;  /* count of zone_hold_cred()s on zone */
 421         /*
 422          * Fixed-sized array of subsystem-specific reference counts
 423          * The sum of all of the counts must be less than or equal to zone_ref.
 424          * The array is indexed by the counts' subsystems' zone_ref_subsys_t
 425          * constants.
 426          */
 427         uint_t          zone_subsys_ref[ZONE_REF_NUM_SUBSYS];
 428         list_t          zone_ref_list;  /* list of zone_ref_t structs */
 429         /*
 430          * zone_rootvp and zone_rootpath can never be modified once set.
 431          */
 432         struct vnode    *zone_rootvp;   /* zone's root vnode */
```



```
 433         char            *zone_rootpath; /* Path to zone's root + '/' */
 434         ushort_t        zone_flags;     /* misc flags */
 435         zone_status_t   zone_status;    /* protected by zone_status_lock */
 436         uint_t          zone_ntasks;    /* number of tasks executing in zone */
 437         kmutex_t        zone_nlwps_lock; /* protects zone_nlwps, and *_nlwps */
 438                                         /* counters in projects and tasks */
 439                                         /* that are within the zone */
 440         rctl_qty_t      zone_nlwps;     /* number of lwps in zone */
 441         rctl_qty_t      zone_nlwps_ctl; /* protected by zone_rctls->rcs_lock */
 442         rctl_qty_t      zone_shmmax;    /* System V shared memory usage */
 443         ipc_rqty_t      zone_ipc;       /* System V IPC id resource usage */

 445         uint_t          zone_rootpathlen; /* strlen(zone_rootpath) + 1 */
 446         uint32_t        zone_shares;    /* FSS shares allocated to zone */
 447         rctl_set_t      *zone_rctls;    /* zone-wide (zone.*) rctls */
 448         kmutex_t        zone_mem_lock;  /* protects zone_locked_mem and */
 449                                         /* kpd_locked_mem for all */
 450                                         /* projects in zone. */
 451                                         /* Also protects zone_max_swap */
 452                                         /* grab after p_lock, before rcs_lock */
 453         rctl_qty_t      zone_locked_mem;        /* bytes of locked memory in */
 454                                                 /* zone */
 455         rctl_qty_t      zone_locked_mem_ctl;    /* Current locked memory */
 456                                                 /* limit.  Protected by */
 457                                                 /* zone_rctls->rcs_lock */
 458         rctl_qty_t      zone_max_swap; /* bytes of swap reserved by zone */
 459         rctl_qty_t      zone_max_swap_ctl;      /* current swap limit. */
 460                                                 /* Protected by */
 461                                                 /* zone_rctls->rcs_lock */
 462         kmutex_t        zone_rctl_lock; /* protects zone_max_lofi */
 463         rctl_qty_t      zone_max_lofi; /* lofi devs for zone */
 464         rctl_qty_t      zone_max_lofi_ctl;      /* current lofi limit. */
 465                                                 /* Protected by */
 466                                                 /* zone_rctls->rcs_lock */
 467         list_t          zone_zsd;       /* list of Zone-Specific Data values */
 468         kcondvar_t      zone_cv;        /* used to signal state changes */
 469         struct proc     *zone_zsched;   /* Dummy kernel "zsched" process */
 470         pid_t           zone_proc_initpid; /* pid of "init" for this zone */
 471         char            *zone_initname; /* fs path to 'init' */
 472         int             zone_boot_err;  /* for zone_boot() if boot fails */
 473         char            *zone_bootargs; /* arguments passed via zone_boot() */
 474         uint64_t        zone_phys_mcap; /* physical memory cap */
 475         /*
 476          * zone_kthreads is protected by zone_status_lock.
 477          */
 478         kthread_t       *zone_kthreads; /* kernel threads in zone */
 479         struct priv_set *zone_privset;  /* limit set for zone */
 480         /*
 481          * zone_vfslist is protected by vfs_list_lock().
 482          */
 483         struct vfs      *zone_vfslist;  /* list of FS's mounted in zone */
 484         uint64_t        zone_uniqid;    /* unique zone generation number */
 485         struct cred     *zone_kcred;    /* kcred-like, zone-limited cred */
 486         /*
 487          * zone_pool is protected by pool_lock().
 488          */
 489         struct pool     *zone_pool;     /* pool the zone is bound to */
 490         hrtime_t        zone_pool_mod;  /* last pool bind modification time */
 491         /* zone_psetid is protected by cpu_lock */
 492         psetid_t        zone_psetid;    /* pset the zone is bound to */

 494         time_t          zone_boot_time; /* Similar to boot_time */

 496         /*
 497          * The following two can be read without holding any locks.  They are
 498          * updated under cpu_lock.
```

```
499          */
500         int             zone_ncpus;  /* zone's idea of ncpus */
501         int             zone_ncpus_online; /* zone's idea of ncpus_online */
502         /*
503          * List of ZFS datasets exported to this zone.
504          */
505         list_t          zone_datasets;  /* list of datasets */
507         ts_label_t      *zone_slabel;   /* zone sensitivity label */
508         int             zone_match;     /* require label match for packets */
509         tsol_mlp_list_t zone_mlps;      /* MLPs on zone-private addresses */
511         boolean_t       zone_restart_init;      /* Restart init if it dies? */
512         struct brand    *zone_brand;            /* zone's brand */
513         void            *zone_brand_data;       /* store brand specific data */
514         id_t            zone_defaultcid;        /* dflt scheduling class id */
515         kstat_t         *zone_swapresv_kstat;
516         kstat_t         *zone_lockedmem_kstat;
517         /*
518          * zone_dl_list is protected by zone_lock
519          */
520         list_t          zone_dl_list;
521         netstack_t      *zone_netstack;
522         struct cpucap   *zone_cpucap;   /* CPU caps data */
523         /*
524          * Solaris Auditing per-zone audit context
525          */
526         struct au_kcontext      *zone_audit_kctxt;
527         /*
528          * For private use by mntfs.
529          */
530         struct mntelem  *zone_mntfs_db;
531         krwlock_t       zone_mntfs_db_lock;
533         struct klpd_reg         *zone_pfexecd;
535         char            *zone_fs_allowed;
536         rctl_qty_t      zone_nprocs;    /* number of processes in the zone */
537         rctl_qty_t      zone_nprocs_ctl;        /* current limit protected by */
538                                                 /* zone_rctls->rcs_lock */
539         kstat_t         *zone_nprocs_kstat;
541         /*
542          * DTrace-private per-zone state
543          */
544         int             zone_dtrace_getf;       /* # of unprivileged getf()s */
545 #endif /* ! codereview */
546 } zone_t;
548 /*
549  * Special value of zone_psetid to indicate that pools are disabled.
550  */
551 #define ZONE_PS_INVAL   PS_MYID
554 extern zone_t zone0;
555 extern zone_t *global_zone;
556 extern uint_t maxzones;
557 extern rctl_hndl_t rc_zone_nlwps;
558 extern rctl_hndl_t rc_zone_nprocs;
560 extern long zone(int, void *, void *, void *, void *);
561 extern void zone_zsd_init(void);
562 extern void zone_init(void);
563 extern void zone_hold(zone_t *);
564 extern void zone_rele(zone_t *);
```



```
565 extern void zone_init_ref(zone_ref_t *);
566 extern void zone_hold_ref(zone_t *, zone_ref_t *, zone_ref_subsys_t);
567 extern void zone_rele_ref(zone_ref_t *, zone_ref_subsys_t);
568 extern void zone_cred_hold(zone_t *);
569 extern void zone_cred_rele(zone_t *);
570 extern void zone_task_hold(zone_t *);
571 extern void zone_task_rele(zone_t *);
572 extern zone_t *zone_find_by_id(zoneid_t);
573 extern zone_t *zone_find_by_label(const ts_label_t *);
574 extern zone_t *zone_find_by_name(char *);
575 extern zone_t *zone_find_by_any_path(const char *, boolean_t);
576 extern zone_t *zone_find_by_path(const char *);
577 extern zoneid_t getzoneid(void);
578 extern zone_t *zone_find_by_id_nolock(zoneid_t);
579 extern int zone_datalink_walk(zoneid_t, int (*)(datalink_id_t, void *), void *);
580 extern int zone_check_datalink(zoneid_t *, datalink_id_t);
582 /*
583  * Zone-specific data (ZSD) APIs
584  */
585 /*
586  * The following is what code should be initializing its zone_key_t to if it
587  * calls zone_getspecific() without necessarily knowing that zone_key_create()
588  * has been called on the key.
589  */
590 #define ZONE_KEY_UNINITIALIZED  0
592 typedef uint_t zone_key_t;
594 extern void     zone_key_create(zone_key_t *, void *(*)(zoneid_t),
595     void (*)(zoneid_t, void *), void (*)(zoneid_t, void *));
596 extern int      zone_key_delete(zone_key_t);
597 extern void     *zone_getspecific(zone_key_t, zone_t *);
598 extern int      zone_setspecific(zone_key_t, zone_t *, const void *);
600 /*
601  * The definition of a zsd_entry is truly private to zone.c and is only
602  * placed here so it can be shared with mdb.
603  *
604  * State maintained for each zone times each registered key, which tracks
605  * the state of the create, shutdown and destroy callbacks.
606  *
607  * zsd_flags is used to keep track of pending actions to avoid holding locks
608  * when calling the create/shutdown/destroy callbacks, since doing so
609  * could lead to deadlocks.
610  */
611 struct zsd_entry {
612         zone_key_t              zsd_key;        /* Key used to lookup value */
613         void                    *zsd_data;      /* Caller-managed value */
614         /*
615          * Callbacks to be executed when a zone is created, shutdown, and
616          * destroyed, respectively.
617          */
618         void                    *(*zsd_create)(zoneid_t);
619         void                    (*zsd_shutdown)(zoneid_t, void *);
620         void                    (*zsd_destroy)(zoneid_t, void *);
621         list_node_t             zsd_linkage;
622         uint16_t                zsd_flags;      /* See below */
623         kcondvar_t              zsd_cv;
624 };
626 /*
627  * zsd_flags
628  */
629 #define ZSD_CREATE_NEEDED       0x0001
630 #define ZSD_CREATE_INPROGRESS   0x0002
```

```
631 #define ZSD_CREATE_COMPLETED    0x0004
632 #define ZSD_SHUTDOWN_NEEDED     0x0010
633 #define ZSD_SHUTDOWN_INPROGRESS 0x0020
634 #define ZSD_SHUTDOWN_COMPLETED  0x0040
635 #define ZSD_DESTROY_NEEDED      0x0100
636 #define ZSD_DESTROY_INPROGRESS  0x0200
637 #define ZSD_DESTROY_COMPLETED   0x0400

639 #define ZSD_CREATE_ALL  \
640         (ZSD_CREATE_NEEDED|ZSD_CREATE_INPROGRESS|ZSD_CREATE_COMPLETED)
641 #define ZSD_SHUTDOWN_ALL        \
642         (ZSD_SHUTDOWN_NEEDED|ZSD_SHUTDOWN_INPROGRESS|ZSD_SHUTDOWN_COMPLETED)
643 #define ZSD_DESTROY_ALL \
644         (ZSD_DESTROY_NEEDED|ZSD_DESTROY_INPROGRESS|ZSD_DESTROY_COMPLETED)

646 #define ZSD_ALL_INPROGRESS \
647         (ZSD_CREATE_INPROGRESS|ZSD_SHUTDOWN_INPROGRESS|ZSD_DESTROY_INPROGRESS)

649 /*
650  * Macros to help with zone visibility restrictions.
651  */

653 /*
654  * Is process in the global zone?
655  */
656 #define INGLOBALZONE(p) \
657         ((p)->p_zone == global_zone)

659 /*
660  * Can process view objects in given zone?
661  */
662 #define HASZONEACCESS(p, zoneid) \
663         ((p)->p_zone->zone_id == (zoneid) || INGLOBALZONE(p))

665 /*
666  * Convenience macro to see if a resolved path is visible from within a
667  * given zone.
668  *
669  * The basic idea is that the first (zone_rootpathlen - 1) bytes of the
670  * two strings must be equal.  Since the rootpathlen has a trailing '/',
671  * we want to skip everything in the path up to (but not including) the
672  * trailing '/'.
673  */
674 #define ZONE_PATH_VISIBLE(path, zone) \
675         (strncmp((path), (zone)->zone_rootpath,         \
676             (zone)->zone_rootpathlen - 1) == 0)

678 /*
679  * Convenience macro to go from the global view of a path to that seen
680  * from within said zone.  It is the responsibility of the caller to
681  * ensure that the path is a resolved one (ie, no '..'s or '.'s), and is
682  * in fact visible from within the zone.
683  */
684 #define ZONE_PATH_TRANSLATE(path, zone) \
685         (ASSERT(ZONE_PATH_VISIBLE(path, zone)), \
686         (path) + (zone)->zone_rootpathlen - 2)

688 /*
689  * Special processes visible in all zones.
690  */
691 #define ZONE_SPECIALPID(x)       ((x) == 0 || (x) == 1)

693 /*
694  * Zone-safe version of thread_create() to be used when the caller wants to
695  * create a kernel thread to run within the current zone's context.
696  */
```



```
697 extern kthread_t *zthread_create(caddr_t, size_t, void (*)(), void *, size_t,
698     pri_t);
699 extern void zthread_exit(void);

701 /*
702  * Functions for an external observer to register interest in a zone's status
703  * change.  Observers will be woken up when the zone status equals the status
704  * argument passed in (in the case of zone_status_timedwait, the function may
705  * also return because of a timeout; zone_status_wait_sig may return early due
706  * to a signal being delivered; zone_status_timedwait_sig may return for any of
707  * the above reasons).
708  *
709  * Otherwise these behave identically to cv_timedwait(), cv_wait(), and
710  * cv_wait_sig() respectively.
711  */
712 extern clock_t zone_status_timedwait(zone_t *, clock_t, zone_status_t);
713 extern clock_t zone_status_timedwait_sig(zone_t *, clock_t, zone_status_t);
714 extern void zone_status_wait(zone_t *, zone_status_t);
715 extern int zone_status_wait_sig(zone_t *, zone_status_t);

717 /*
718  * Get the status  of the zone (at the time it was called).  The state may
719  * have progressed by the time it is returned.
720  */
721 extern zone_status_t zone_status_get(zone_t *);

723 /*
724  * Safely get the hostid of the specified zone (defaults to machine's hostid
725  * if the specified zone doesn't emulate a hostid).  Passing NULL retrieves
726  * the global zone's (i.e., physical system's) hostid.
727  */
728 extern uint32_t zone_get_hostid(zone_t *);

730 /*
731  * Get the "kcred" credentials corresponding to the given zone.
732  */
733 extern struct cred *zone_get_kcred(zoneid_t);

735 /*
736  * Get/set the pool the zone is currently bound to.
737  */
738 extern struct pool *zone_pool_get(zone_t *);
739 extern void zone_pool_set(zone_t *, struct pool *);

741 /*
742  * Get/set the pset the zone is currently using.
743  */
744 extern psetid_t zone_pset_get(zone_t *);
745 extern void zone_pset_set(zone_t *, psetid_t);

747 /*
748  * Get the number of cpus/online-cpus visible from the given zone.
749  */
750 extern int zone_ncpus_get(zone_t *);
751 extern int zone_ncpus_online_get(zone_t *);

753 /*
754  * Returns true if the named pool/dataset is visible in the current zone.
755  */
756 extern int zone_dataset_visible(const char *, int *);

758 /*
759  * zone version of kadmin()
760  */
761 extern int zone_kadmin(int, int, const char *, cred_t *);
762 extern void zone_shutdown_global(void);
```

```
764 extern void mount_in_progress(void);
765 extern void mount_completed(void);

767 extern int zone_walk(int (*)(zone_t *, void *), void *);

769 extern rctl_hndl_t rc_zone_locked_mem;
770 extern rctl_hndl_t rc_zone_max_swap;
771 extern rctl_hndl_t rc_zone_max_lofi;

773 #endif  /* _KERNEL */

775 #ifdef  __cplusplus
776 }
777 #endif

779 #endif  /* _SYS_ZONE_H */
```

```
**********************************************************
   13145 Tue Jan 14 16:50:06 2014
new/usr/src/uts/intel/dtrace/sdt.c
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

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

/*
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 */
#endif /* ! codereview */

#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);
}

/*ARGSUSED*/
static void
sdt_provide_module(void *arg, struct modctl *ctl)
{
        struct module *mp = ctl->mod_mp;
        char *modname = ctl->mod_modname;
        sdt_probedesc_t *sdpd;
        sdt_probe_t *sdp, *old;
        sdt_provider_t *prov;
        int len;

        /*
         * One for all, and all for one:  if we haven't yet registered all of
         * our providers, we'll refuse to provide anything.
         */
        for (prov = sdt_providers; prov->sdtp_name != NULL; prov++) {
                if (prov->sdtp_id == DTRACE_PROVNONE)
                        return;
        }

        if (mp->sdt_nprobes != 0 || (sdpd = mp->sdt_probes) == NULL)
                return;

        for (sdpd = mp->sdt_probes; sdpd != NULL; sdpd = sdpd->sdpd_next) {
```

```
124                 char *name = sdpd->sdpd_name, *func, *nname;
125                 int i, j;
126                 sdt_provider_t *prov;
127                 ulong_t offs;
128                 dtrace_id_t id;

130                 for (prov = sdt_providers; prov->sdtp_prefix != NULL; prov++) {
131                         char *prefix = prov->sdtp_prefix;

133                         if (strncmp(name, prefix, strlen(prefix)) == 0) {
134                                 name += strlen(prefix);
135                                 break;
136                         }
137                 }

139                 nname = kmem_alloc(len = strlen(name) + 1, KM_SLEEP);

141                 for (i = 0, j = 0; name[j] != '\0'; i++) {
142                         if (name[j] == '_' && name[j + 1] == '_') {
143                                 nname[i] = '-';
144                                 j += 2;
145                         } else {
146                                 nname[i] = name[j++];
147                         }
148                 }

150                 nname[i] = '\0';

152                 sdp = kmem_zalloc(sizeof (sdt_probe_t), KM_SLEEP);
153                 sdp->sdp_loadcnt = ctl->mod_loadcnt;
154                 sdp->sdp_ctl = ctl;
155                 sdp->sdp_name = nname;
156                 sdp->sdp_namelen = len;
157                 sdp->sdp_provider = prov;

159                 func = kobj_searchsym(mp, sdpd->sdpd_offset, &offs);

161                 if (func == NULL)
162                         func = "<unknown>";

164                 /*
165                  * We have our provider.  Now create the probe.
166                  */
167                 if ((id = dtrace_probe_lookup(prov->sdtp_id, modname,
168                     func, nname)) != DTRACE_IDNONE) {
169                         old = dtrace_probe_arg(prov->sdtp_id, id);
170                         ASSERT(old != NULL);

172                         sdp->sdp_next = old->sdp_next;
173                         sdp->sdp_id = id;
174                         old->sdp_next = sdp;
175                 } else {
176                         sdp->sdp_id = dtrace_probe_create(prov->sdtp_id,
177                             modname, func, nname, 3, sdp);

179                         mp->sdt_nprobes++;
180                 }

182                 sdp->sdp_hashnext =
183                     sdt_probetab[SDT_ADDR2NDX(sdpd->sdpd_offset)];
184                 sdt_probetab[SDT_ADDR2NDX(sdpd->sdpd_offset)] = sdp;

186                 sdp->sdp_patchval = SDT_PATCHVAL;
187                 sdp->sdp_patchpoint = (uint8_t *)sdpd->sdpd_offset;
188                 sdp->sdp_savedval = *sdp->sdp_patchpoint;
189         }
```



```
190 }

192 /*ARGSUSED*/
193 static void
194 sdt_destroy(void *arg, dtrace_id_t id, void *parg)
195 {
196         sdt_probe_t *sdp = parg, *old, *last, *hash;
197         struct modctl *ctl = sdp->sdp_ctl;
198         int ndx;

200         if (ctl != NULL && ctl->mod_loadcnt == sdp->sdp_loadcnt) {
201                 if ((ctl->mod_loadcnt == sdp->sdp_loadcnt &&
202                     ctl->mod_loaded)) {
203                         ((struct module *)(ctl->mod_mp))->sdt_nprobes--;
204                 }
205         }

207         while (sdp != NULL) {
208                 old = sdp;

210                 /*
211                  * Now we need to remove this probe from the sdt_probetab.
212                  */
213                 ndx = SDT_ADDR2NDX(sdp->sdp_patchpoint);
214                 last = NULL;
215                 hash = sdt_probetab[ndx];

217                 while (hash != sdp) {
218                         ASSERT(hash != NULL);
219                         last = hash;
220                         hash = hash->sdp_hashnext;
221                 }

223                 if (last != NULL) {
224                         last->sdp_hashnext = sdp->sdp_hashnext;
225                 } else {
226                         sdt_probetab[ndx] = sdp->sdp_hashnext;
227                 }

229                 kmem_free(sdp->sdp_name, sdp->sdp_namelen);
230                 sdp = sdp->sdp_next;
231                 kmem_free(old, sizeof (sdt_probe_t));
232         }
233 }

235 /*ARGSUSED*/
236 static int
237 sdt_enable(void *arg, dtrace_id_t id, void *parg)
238 {
239         sdt_probe_t *sdp = parg;
240         struct modctl *ctl = sdp->sdp_ctl;

242         ctl->mod_nenabled++;

244         /*
245          * If this module has disappeared since we discovered its probes,
246          * refuse to enable it.
247          */
248         if (!ctl->mod_loaded) {
249                 if (sdt_verbose) {
250                         cmn_err(CE_NOTE, "sdt is failing for probe %s "
251                             "(module %s unloaded)",
252                             sdp->sdp_name, ctl->mod_modname);
253                 }
254                 goto err;
255         }
```

```
257         /*
258          * Now check that our modctl has the expected load count.  If it
259          * doesn't, this module must have been unloaded and reloaded -- and
260          * we're not going to touch it.
261          */
262         if (ctl->mod_loadcnt != sdp->sdp_loadcnt) {
263                 if (sdt_verbose) {
264                         cmn_err(CE_NOTE, "sdt is failing for probe %s "
265                             "(module %s reloaded)",
266                             sdp->sdp_name, ctl->mod_modname);
267                 }
268                 goto err;
269         }
270
271         while (sdp != NULL) {
272                 *sdp->sdp_patchpoint = sdp->sdp_patchval;
273                 sdp = sdp->sdp_next;
274         }
275 err:
276         return (0);
277 }
278
279 /*ARGSUSED*/
280 static void
281 sdt_disable(void *arg, dtrace_id_t id, void *parg)
282 {
283         sdt_probe_t *sdp = parg;
284         struct modctl *ctl = sdp->sdp_ctl;
285
286         ctl->mod_nenabled--;
287
288         if (!ctl->mod_loaded || ctl->mod_loadcnt != sdp->sdp_loadcnt)
289                 goto err;
290
291         while (sdp != NULL) {
292                 *sdp->sdp_patchpoint = sdp->sdp_savedval;
293                 sdp = sdp->sdp_next;
294         }
295
296 err:
297         ;
298 }
299
300 /*ARGSUSED*/
301 uint64_t
302 sdt_getarg(void *arg, dtrace_id_t id, void *parg, int argno, int aframes)
303 {
304         uintptr_t val;
305         struct frame *fp = (struct frame *)dtrace_getfp();
306         uintptr_t *stack;
307         int i;
308 #if defined(__amd64)
309         /*
310          * A total of 6 arguments are passed via registers; any argument with
311          * index of 5 or lower is therefore in a register.
312          */
313         int inreg = 5;
314 #endif
315
316         for (i = 1; i <= aframes; i++) {
317                 fp = (struct frame *)(fp->fr_savfp);
318
319                 if (fp->fr_savpc == (pc_t)dtrace_invop_callsite) {
320 #if !defined(__amd64)
321                         /*
```



```
322                          * If we pass through the invalid op handler, we will
323                          * use the pointer that it passed to the stack as the
324                          * second argument to dtrace_invop() as the pointer to
325                          * the stack.
326                          */
327                         stack = ((uintptr_t **)&fp[1])[1];
328 #else
329                         /*
330                          * In the case of amd64, we will use the pointer to the
331                          * regs structure that was pushed when we took the
332                          * trap.  To get this structure, we must increment
333                          * beyond the frame structure.  If the argument that
334                          * we're seeking is passed on the stack, we'll pull
335                          * the true stack pointer out of the saved registers
336                          * and decrement our argument by the number of
337                          * arguments passed in registers; if the argument
338                          * we're seeking is passed in regsiters, we can just
339                          * load it directly.
340                          */
341                         struct regs *rp = (struct regs *)((uintptr_t)&fp[1] +
342                             sizeof (uintptr_t));
343
344                         if (argno <= inreg) {
345                                 stack = (uintptr_t *)&rp->r_rdi;
346                         } else {
347                                 stack = (uintptr_t *)(rp->r_rsp);
348                                 argno -= (inreg + 1);
349                         }
350 #endif
351                         goto load;
352                 }
353         }
354
355         /*
356          * We know that we did not come through a trap to get into
357          * dtrace_probe() -- the provider simply called dtrace_probe()
358          * directly.  As this is the case, we need to shift the argument
359          * that we're looking for:  the probe ID is the first argument to
360          * dtrace_probe(), so the argument n will actually be found where
361          * one would expect to find argument (n + 1).
362          */
363         argno++;
364
365 #if defined(__amd64)
366         if (argno <= inreg) {
367                 /*
368                  * This shouldn't happen.  If the argument is passed in a
369                  * register then it should have been, well, passed in a
370                  * register...
371                  */
372                 DTRACE_CPUFLAG_SET(CPU_DTRACE_ILLOP);
373                 return (0);
374         }
375
376         argno -= (inreg + 1);
377 #endif
378         stack = (uintptr_t *)&fp[1];
379
380 load:
381         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
382         val = stack[argno];
383         DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
384
385         return (val);
386 }
```

```
388 static dtrace_pops_t sdt_pops = {
389         NULL,
390         sdt_provide_module,
391         sdt_enable,
392         sdt_disable,
393         NULL,
394         NULL,
395         sdt_getargdesc,
396         sdt_getarg,
397         NULL,
398         sdt_destroy
399 };

401 /*ARGSUSED*/
402 static int
403 sdt_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
404 {
405         sdt_provider_t *prov;

407         if (ddi_create_minor_node(devi, "sdt", S_IFCHR,
408             0, DDI_PSEUDO, NULL) == DDI_FAILURE) {
409                 cmn_err(CE_NOTE, "/dev/sdt couldn't create minor node");
410                 ddi_remove_minor_node(devi, NULL);
411                 return (DDI_FAILURE);
412         }

414         ddi_report_dev(devi);
415         sdt_devi = devi;

417         if (sdt_probetab_size == 0)
418                 sdt_probetab_size = SDT_PROBETAB_SIZE;

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

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

428                 if (prov->sdtp_priv == DTRACE_PRIV_NONE) {
429                         priv = DTRACE_PRIV_KERNEL;
430                         sdt_pops.dtps_mode = NULL;
431                 } else {
432                         priv = prov->sdtp_priv;
433                         ASSERT(priv == DTRACE_PRIV_USER);
434                         sdt_pops.dtps_mode = sdt_mode;
435                 }

437 #endif /* ! codereview */
438                 if (dtrace_register(prov->sdtp_name, prov->sdtp_attr,
439                     priv, NULL, &sdt_pops, prov, &prov->sdtp_id) != 0) {
 26                     DTRACE_PRIV_KERNEL, NULL,
 27                     &sdt_pops, prov, &prov->sdtp_id) != 0) {
440                         cmn_err(CE_WARN, "failed to register sdt provider %s",
441                             prov->sdtp_name);
442                 }
443         }

445         return (DDI_SUCCESS);
446 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   11768 Tue Jan 14 16:50:07 2014
new/usr/src/uts/sparc/dtrace/sdt.c
2915 DTrace in a zone should see "cpu", "curpsinfo", et al
2916 DTrace in a zone should be able to access fds[]
2917 DTrace in a zone should have limited provider access
Reviewed by: Joshua M. Clulow <josh@sysmgr.org>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

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

/*
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 */
#endif /* ! codereview */

#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;
```

```
124 }

126 /*ARGSUSED*/
127 static void
128 sdt_provide_module(void *arg, struct modctl *ctl)
129 {
130         struct module *mp = ctl->mod_mp;
131         char *modname = ctl->mod_modname;
132         int primary, nprobes = 0;
133         sdt_probedesc_t *sdpd;
134         sdt_probe_t *sdp, *old;
135         uint32_t *tab;
136         sdt_provider_t *prov;
137         int len;

139         /*
140          * One for all, and all for one:  if we haven't yet registered all of
141          * our providers, we'll refuse to provide anything.
142          */
143         for (prov = sdt_providers; prov->sdtp_name != NULL; prov++) {
144                 if (prov->sdtp_id == DTRACE_PROVNONE)
145                         return;
146         }

148         if (mp->sdt_nprobes != 0 || (sdpd = mp->sdt_probes) == NULL)
149                 return;

151         kobj_textwin_alloc(mp);

153         /*
154          * Hack to identify unix/genunix/krtld.
155          */
156         primary = vmem_contains(heap_arena, (void *)ctl,
157             sizeof (struct modctl)) == 0;

159         /*
160          * If there hasn't been an sdt table allocated, we'll do so now.
161          */
162         if (mp->sdt_tab == NULL) {
163                 for (; sdpd != NULL; sdpd = sdpd->sdpd_next) {
164                         nprobes++;
165                 }

167                 /*
168                  * We could (should?) determine precisely the size of the
169                  * table -- but a reasonable maximum will suffice.
170                  */
171                 mp->sdt_size = nprobes * SDT_ENTRY_SIZE;
172                 mp->sdt_tab = kobj_texthole_alloc(mp->text, mp->sdt_size);

174                 if (mp->sdt_tab == NULL) {
175                         cmn_err(CE_WARN, "couldn't allocate SDT table "
176                             "for module %s", modname);
177                         return;
178                 }
179         }

181         tab = (uint32_t *)mp->sdt_tab;

183         for (sdpd = mp->sdt_probes; sdpd != NULL; sdpd = sdpd->sdpd_next) {
184                 char *name = sdpd->sdpd_name, *func, *nname;
185                 int i, j;
186                 sdt_provider_t *prov;
187                 ulong_t offs;
188                 dtrace_id_t id;
```



```
190                 for (prov = sdt_providers; prov->sdtp_prefix != NULL; prov++) {
191                         char *prefix = prov->sdtp_prefix;

193                         if (strncmp(name, prefix, strlen(prefix)) == 0) {
194                                 name += strlen(prefix);
195                                 break;
196                         }
197                 }

199                 nname = kmem_alloc(len = strlen(name) + 1, KM_SLEEP);

201                 for (i = 0, j = 0; name[j] != '\0'; i++) {
202                         if (name[j] == '_' && name[j + 1] == '_') {
203                                 nname[i] = '-';
204                                 j += 2;
205                         } else {
206                                 nname[i] = name[j++];
207                         }
208                 }

210                 nname[i] = '\0';

212                 sdp = kmem_zalloc(sizeof (sdt_probe_t), KM_SLEEP);
213                 sdp->sdp_loadcnt = ctl->mod_loadcnt;
214                 sdp->sdp_primary = primary;
215                 sdp->sdp_ctl = ctl;
216                 sdp->sdp_name = nname;
217                 sdp->sdp_namelen = len;
218                 sdp->sdp_provider = prov;

220                 func = kobj_searchsym(mp, sdpd->sdpd_offset +
221                     (uintptr_t)mp->text, &offs);

223                 if (func == NULL)
224                         func = "<unknown>";

226                 /*
227                  * We have our provider.  Now create the probe.
228                  */
229                 if ((id = dtrace_probe_lookup(prov->sdtp_id, modname,
230                     func, nname)) != DTRACE_IDNONE) {
231                         old = dtrace_probe_arg(prov->sdtp_id, id);
232                         ASSERT(old != NULL);

234                         sdp->sdp_next = old->sdp_next;
235                         sdp->sdp_id = id;
236                         old->sdp_next = sdp;
237                 } else {
238                         sdp->sdp_id = dtrace_probe_create(prov->sdtp_id,
239                             modname, func, nname, 1, sdp);

241                         mp->sdt_nprobes++;
242                 }

244                 sdp->sdp_patchval = SDT_CALL((uintptr_t)mp->text +
245                     sdpd->sdpd_offset, tab);
246                 sdp->sdp_patchpoint = (uint32_t *)((uintptr_t)mp->textwin +
247                     sdpd->sdpd_offset);
248                 sdp->sdp_savedval = *sdp->sdp_patchpoint;
249                 sdt_initialize(sdp, &tab);
250         }
251 }

253 /*ARGSUSED*/
254 static void
255 sdt_destroy(void *arg, dtrace_id_t id, void *parg)
```

```
256 {
257         sdt_probe_t *sdp = parg, *old;
258         struct modctl *ctl = sdp->sdp_ctl;

260         if (ctl != NULL && ctl->mod_loadcnt == sdp->sdp_loadcnt) {
261                 if ((ctl->mod_loadcnt == sdp->sdp_loadcnt &&
262                     ctl->mod_loaded) || sdp->sdp_primary) {
263                         ((struct module *)(ctl->mod_mp))->sdt_nprobes--;
264                 }
265         }

267         while (sdp != NULL) {
268                 old = sdp;
269                 kmem_free(sdp->sdp_name, sdp->sdp_namelen);
270                 sdp = sdp->sdp_next;
271                 kmem_free(old, sizeof (sdt_probe_t));
272         }
273 }

275 /*ARGSUSED*/
276 static int
277 sdt_enable(void *arg, dtrace_id_t id, void *parg)
278 {
279         sdt_probe_t *sdp = parg;
280         struct modctl *ctl = sdp->sdp_ctl;

282         ctl->mod_nenabled++;

284         /*
285          * If this module has disappeared since we discovered its probes,
286          * refuse to enable it.
287          */
288         if (!sdp->sdp_primary && !ctl->mod_loaded) {
289                 if (sdt_verbose) {
290                         cmn_err(CE_NOTE, "sdt is failing for probe %s "
291                             "(module %s unloaded)",
292                             sdp->sdp_name, ctl->mod_modname);
293                 }
294                 goto err;
295         }

297         /*
298          * Now check that our modctl has the expected load count.  If it
299          * doesn't, this module must have been unloaded and reloaded -- and
300          * we're not going to touch it.
301          */
302         if (ctl->mod_loadcnt != sdp->sdp_loadcnt) {
303                 if (sdt_verbose) {
304                         cmn_err(CE_NOTE, "sdt is failing for probe %s "
305                             "(module %s reloaded)",
306                             sdp->sdp_name, ctl->mod_modname);
307                 }
308                 goto err;
309         }

311         while (sdp != NULL) {
312                 *sdp->sdp_patchpoint = sdp->sdp_patchval;
313                 sdp = sdp->sdp_next;
314         }

316 err:
317         return (0);
318 }

320 /*ARGSUSED*/
321 static void
```



```
322 sdt_disable(void *arg, dtrace_id_t id, void *parg)
323 {
324         sdt_probe_t *sdp = parg;
325         struct modctl *ctl = sdp->sdp_ctl;

327         ASSERT(ctl->mod_nenabled > 0);
328         ctl->mod_nenabled--;

330         if ((!sdp->sdp_primary && !ctl->mod_loaded) ||
331             (ctl->mod_loadcnt != sdp->sdp_loadcnt))
332                 goto err;

334         while (sdp != NULL) {
335                 *sdp->sdp_patchpoint = sdp->sdp_savedval;
336                 sdp = sdp->sdp_next;
337         }

339 err:
340         ;
341 }

343 static dtrace_pops_t sdt_pops = {
344         NULL,
345         sdt_provide_module,
346         sdt_enable,
347         sdt_disable,
348         NULL,
349         NULL,
350         sdt_getargdesc,
351         NULL,
352         NULL,
353         sdt_destroy
354 };

356 static int
357 sdt_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
358 {
359         sdt_provider_t *prov;

361         switch (cmd) {
362         case DDI_ATTACH:
363                 break;
364         case DDI_RESUME:
365                 return (DDI_SUCCESS);
366         default:
367                 return (DDI_FAILURE);
368         }

370         if (ddi_create_minor_node(devi, "sdt", S_IFCHR, 0,
371             DDI_PSEUDO, NULL) == DDI_FAILURE) {
372                 ddi_remove_minor_node(devi, NULL);
373                 return (DDI_FAILURE);
374         }

376         ddi_report_dev(devi);
377         sdt_devi = devi;

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

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

```
388                         sdt_pops.dtps_mode = sdt_mode;
389                 }

391 #endif /* ! codereview */
392                 if (dtrace_register(prov->sdtp_name, prov->sdtp_attr,
393                     priv, NULL, &sdt_pops, prov, &prov->sdtp_id) != 0) {
 26                     DTRACE_PRIV_KERNEL, NULL,
 27                     &sdt_pops, prov, &prov->sdtp_id) != 0) {
394                         cmn_err(CE_WARN, "failed to register sdt provider %s",
395                             prov->sdtp_name);
396                 }
397         }

399         return (DDI_SUCCESS);
400 }
```

*unchanged_portion_omitted_*