```
**********************************************************
    2951 Fri Jun 22 23:48:16 2012
new/usr/src/cmd/dtrace/test/tst/common/aggs/tst.subr.d
2916 DTrace in a zone should be able to access fds[]
**********************************************************
```

```
/*
 * 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))

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

_____*unchanged_portion_omitted_*

```
**********************************************************
    2018 Fri Jun 22 23:48:16 2012
new/usr/src/cmd/dtrace/test/tst/common/privs/tst.fds.ksh
2916 DTrace in a zone should be able to access fds[]
**********************************************************
```

```
#
# 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
```

```
**********************************************************
    2178 Fri Jun 22 23:48:17 2012
new/usr/src/cmd/dtrace/test/tst/common/privs/tst.getf.ksh
2916 DTrace in a zone should be able to access fds[]
**********************************************************
```

```
#
# 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
```

```
**********************************************************
   53759 Fri Jun 22 23:48:17 2012
new/usr/src/lib/libdtrace/common/dt_open.c
2916 DTrace in a zone should be able to access fds[]
**********************************************************
```

```
/*
 * 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) 2011 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 Solaris. 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_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
#define DT_VERS_STRING  "Sun D 1.9"

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 */
        DT_VERS_1_2,    /* D API 1.2.0 Solaris 10 Update 1 */
        DT_VERS_1_2_1,  /* D API 1.2.1 Solaris Express 4/06 */
        DT_VERS_1_2_2,  /* D API 1.2.2 Solaris Express 6/06 */
        DT_VERS_1_3,    /* D API 1.3 Solaris Express 10/06 */
        DT_VERS_1_4,    /* D API 1.4 Solaris Express 2/07 */
```

```
125         DT_VERS_1_4_1,  /* D API 1.4.1 Solaris Express 4/07 */
126         DT_VERS_1_5,    /* D API 1.5 Solaris Express 7/07 */
127         DT_VERS_1_6,    /* D API 1.6 */
128         DT_VERS_1_6_1,  /* D API 1.6.1 */
129         DT_VERS_1_6_2,  /* D API 1.6.2 */
130         DT_VERS_1_6_3,  /* D API 1.6.3 */
131         DT_VERS_1_7,    /* D API 1.7 */
132         DT_VERS_1_7_1,  /* D API 1.7.1 */
133         DT_VERS_1_8,    /* D API 1.8 */
134         DT_VERS_1_8_1,  /* D API 1.8.1 */
135         DT_VERS_1_9,    /* D API 1.9 */
136         DT_VERS_1_10,   /* D API 1.10 */
137         0
138 };

140 /*
141  * Table of global identifiers.  This is used to populate the global identifier
142  * hash when a new dtrace client open occurs.  For more info see dt_ident.h.
143  * The global identifiers that represent functions use the dt_idops_func ops
144  * and specify the private data pointer as a prototype string which is parsed
145  * when the identifier is first encountered.  These prototypes look like ANSI
146  * C function prototypes except that the special symbol "@" can be used as a
147  * wildcard to represent a single parameter of any type (i.e. any dt_node_t).
148  * The standard "..." notation can also be used to represent varargs.  An empty
149  * parameter list is taken to mean void (that is, no arguments are permitted).
150  * A parameter enclosed in square brackets (e.g. "[int]") denotes an optional
151  * argument.
152  */
153 static const dt_ident_t _dtrace_globals[] = {
154 { "alloca", DT_IDENT_FUNC, 0, DIF_SUBR_ALLOCA, DT_ATTR_STABCMN, DT_VERS_1_0,
155         &dt_idops_func, "void *(size_t)" },
156 { "arg0", DT_IDENT_SCALAR, 0, DIF_VAR_ARG0, DT_ATTR_STABCMN, DT_VERS_1_0,
157         &dt_idops_type, "int64_t" },
158 { "arg1", DT_IDENT_SCALAR, 0, DIF_VAR_ARG1, DT_ATTR_STABCMN, DT_VERS_1_0,
159         &dt_idops_type, "int64_t" },
160 { "arg2", DT_IDENT_SCALAR, 0, DIF_VAR_ARG2, DT_ATTR_STABCMN, DT_VERS_1_0,
161         &dt_idops_type, "int64_t" },
162 { "arg3", DT_IDENT_SCALAR, 0, DIF_VAR_ARG3, DT_ATTR_STABCMN, DT_VERS_1_0,
163         &dt_idops_type, "int64_t" },
164 { "arg4", DT_IDENT_SCALAR, 0, DIF_VAR_ARG4, DT_ATTR_STABCMN, DT_VERS_1_0,
165         &dt_idops_type, "int64_t" },
166 { "arg5", DT_IDENT_SCALAR, 0, DIF_VAR_ARG5, DT_ATTR_STABCMN, DT_VERS_1_0,
167         &dt_idops_type, "int64_t" },
168 { "arg6", DT_IDENT_SCALAR, 0, DIF_VAR_ARG6, DT_ATTR_STABCMN, DT_VERS_1_0,
169         &dt_idops_type, "int64_t" },
170 { "arg7", DT_IDENT_SCALAR, 0, DIF_VAR_ARG7, DT_ATTR_STABCMN, DT_VERS_1_0,
171         &dt_idops_type, "int64_t" },
172 { "arg8", DT_IDENT_SCALAR, 0, DIF_VAR_ARG8, DT_ATTR_STABCMN, DT_VERS_1_0,
173         &dt_idops_type, "int64_t" },
174 { "arg9", DT_IDENT_SCALAR, 0, DIF_VAR_ARG9, DT_ATTR_STABCMN, DT_VERS_1_0,
175         &dt_idops_type, "int64_t" },
176 { "args", DT_IDENT_ARRAY, 0, DIF_VAR_ARGS, DT_ATTR_STABCMN, DT_VERS_1_0,
177         &dt_idops_args, NULL },
178 { "avg", DT_IDENT_AGGFUNC, 0, DTRACEAGG_AVG, DT_ATTR_STABCMN, DT_VERS_1_0,
179         &dt_idops_func, "void(@)" },
180 { "basename", DT_IDENT_FUNC, 0, DIF_SUBR_BASENAME, DT_ATTR_STABCMN, DT_VERS_1_0,
181         &dt_idops_func, "string(const char *)" },
182 { "bcopy", DT_IDENT_FUNC, 0, DIF_SUBR_BCOPY, DT_ATTR_STABCMN, DT_VERS_1_0,
183         &dt_idops_func, "void(void *, void *, size_t)" },
184 { "breakpoint", DT_IDENT_ACTFUNC, 0, DT_ACT_BREAKPOINT,
185         DT_ATTR_STABCMN, DT_VERS_1_0,
186         &dt_idops_func, "void()" },
187 { "caller", DT_IDENT_SCALAR, 0, DIF_VAR_CALLER, DT_ATTR_STABCMN, DT_VERS_1_0,
188         &dt_idops_type, "uintptr_t" },
189 { "chill", DT_IDENT_ACTFUNC, 0, DT_ACT_CHILL, DT_ATTR_STABCMN, DT_VERS_1_0,
190         &dt_idops_func, "void(int)" },
```



```
191 { "cleanpath", DT_IDENT_FUNC, 0, DIF_SUBR_CLEANPATH, DT_ATTR_STABCMN,
192         DT_VERS_1_0, &dt_idops_func, "string(const char *)" },
193 { "clear", DT_IDENT_ACTFUNC, 0, DT_ACT_CLEAR, DT_ATTR_STABCMN, DT_VERS_1_0,
194         &dt_idops_func, "void(...)" },
195 { "commit", DT_IDENT_ACTFUNC, 0, DT_ACT_COMMIT, DT_ATTR_STABCMN, DT_VERS_1_0,
196         &dt_idops_func, "void(int)" },
197 { "copyin", DT_IDENT_FUNC, 0, DIF_SUBR_COPYIN, DT_ATTR_STABCMN, DT_VERS_1_0,
198         &dt_idops_func, "void *(uintptr_t, size_t)" },
199 { "copyinstr", DT_IDENT_FUNC, 0, DIF_SUBR_COPYINSTR,
200         DT_ATTR_STABCMN, DT_VERS_1_0,
201         &dt_idops_func, "string(uintptr_t, [size_t])" },
202 { "copyinto", DT_IDENT_FUNC, 0, DIF_SUBR_COPYINTO, DT_ATTR_STABCMN,
203         DT_VERS_1_0, &dt_idops_func, "void(uintptr_t, size_t, void *)" },
204 { "copyout", DT_IDENT_FUNC, 0, DIF_SUBR_COPYOUT, DT_ATTR_STABCMN, DT_VERS_1_0,
205         &dt_idops_func, "void(void *, uintptr_t, size_t)" },
206 { "copyoutstr", DT_IDENT_FUNC, 0, DIF_SUBR_COPYOUTSTR,
207         DT_ATTR_STABCMN, DT_VERS_1_0,
208         &dt_idops_func, "void(char *, uintptr_t, size_t)" },
209 { "count", DT_IDENT_AGGFUNC, 0, DTRACEAGG_COUNT, DT_ATTR_STABCMN, DT_VERS_1_0,
210         &dt_idops_func, "void()" },
211 { "curthread", DT_IDENT_SCALAR, 0, DIF_VAR_CURTHREAD,
212         { DTRACE_STABILITY_STABLE, DTRACE_STABILITY_PRIVATE,
213         DTRACE_CLASS_COMMON }, DT_VERS_1_0,
214         &dt_idops_type, "genunix`kthread_t *" },
215 { "ddi_pathname", DT_IDENT_FUNC, 0, DIF_SUBR_DDI_PATHNAME,
216         DT_ATTR_EVOLCMN, DT_VERS_1_0,
217         &dt_idops_func, "string(void *, int64_t)" },
218 { "denormalize", DT_IDENT_ACTFUNC, 0, DT_ACT_DENORMALIZE, DT_ATTR_STABCMN,
219         DT_VERS_1_0, &dt_idops_func, "void(...)" },
220 { "dirname", DT_IDENT_FUNC, 0, DIF_SUBR_DIRNAME, DT_ATTR_STABCMN, DT_VERS_1_0,
221         &dt_idops_func, "string(const char *)" },
222 { "discard", DT_IDENT_ACTFUNC, 0, DT_ACT_DISCARD, DT_ATTR_STABCMN, DT_VERS_1_0,
223         &dt_idops_func, "void(int)" },
224 { "epid", DT_IDENT_SCALAR, 0, DIF_VAR_EPID, DT_ATTR_STABCMN, DT_VERS_1_0,
225         &dt_idops_type, "uint_t" },
226 { "errno", DT_IDENT_SCALAR, 0, DIF_VAR_ERRNO, DT_ATTR_STABCMN, DT_VERS_1_0,
227         &dt_idops_type, "int" },
228 { "execname", DT_IDENT_SCALAR, 0, DIF_VAR_EXECNAME,
229         DT_ATTR_STABCMN, DT_VERS_1_0, &dt_idops_type, "string" },
230 { "exit", DT_IDENT_ACTFUNC, 0, DT_ACT_EXIT, DT_ATTR_STABCMN, DT_VERS_1_0,
231         &dt_idops_func, "void(int)" },
232 { "freopen", DT_IDENT_ACTFUNC, 0, DT_ACT_FREOPEN, DT_ATTR_STABCMN,
233         DT_VERS_1_1, &dt_idops_func, "void(@, ...)" },
234 { "ftruncate", DT_IDENT_ACTFUNC, 0, DT_ACT_FTRUNCATE, DT_ATTR_STABCMN,
235         DT_VERS_1_0, &dt_idops_func, "void()" },
236 { "func", DT_IDENT_ACTFUNC, 0, DT_ACT_SYM, DT_ATTR_STABCMN,
237         DT_VERS_1_2, &dt_idops_func, "_symaddr(uintptr_t)" },
238 { "getmajor", DT_IDENT_FUNC, 0, DIF_SUBR_GETMAJOR,
239         DT_ATTR_EVOLCMN, DT_VERS_1_0,
240         &dt_idops_func, "genunix`major_t(genunix`dev_t)" },
241 { "getminor", DT_IDENT_FUNC, 0, DIF_SUBR_GETMINOR,
242         DT_ATTR_EVOLCMN, DT_VERS_1_0,
243         &dt_idops_func, "genunix`minor_t(genunix`dev_t)" },
244 { "htonl", DT_IDENT_FUNC, 0, DIF_SUBR_HTONL, DT_ATTR_EVOLCMN, DT_VERS_1_3,
245         &dt_idops_func, "uint32_t(uint32_t)" },
246 { "htonll", DT_IDENT_FUNC, 0, DIF_SUBR_HTONLL, DT_ATTR_EVOLCMN, DT_VERS_1_3,
247         &dt_idops_func, "uint64_t(uint64_t)" },
248 { "htons", DT_IDENT_FUNC, 0, DIF_SUBR_HTONS, DT_ATTR_EVOLCMN, DT_VERS_1_3,
249         &dt_idops_func, "uint16_t(uint16_t)" },
250 { "getf", DT_IDENT_FUNC, 0, DIF_SUBR_GETF, DT_ATTR_STABCMN, DT_VERS_1_10,
251         &dt_idops_func, "file_t *(int)" },
252 { "gid", DT_IDENT_SCALAR, 0, DIF_VAR_GID, DT_ATTR_STABCMN, DT_VERS_1_0,
253         &dt_idops_type, "gid_t" },
254 { "id", DT_IDENT_SCALAR, 0, DIF_VAR_ID, DT_ATTR_STABCMN, DT_VERS_1_0,
255         &dt_idops_type, "uint_t" },
256 { "index", DT_IDENT_FUNC, 0, DIF_SUBR_INDEX, DT_ATTR_STABCMN, DT_VERS_1_1,
```

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



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

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

*unchanged_portion_omitted_*

```
**********************************************************
    7992 Fri Jun 22 23:48:17 2012
new/usr/src/lib/libdtrace/common/io.d.in
2916 DTrace in a zone should be able to access fds[]
**********************************************************
/*
 * 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_
```

```
**********************************************************
  419011 Fri Jun 22 23:48:18 2012
new/usr/src/uts/common/dtrace/dtrace.c
2916 DTrace in a zone should be able to access fds[]
**********************************************************
```

```
/*
 * 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.
 */

/*
 * 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 */
```

```
128 dtrace_optval_t dtrace_cleanrate_max = (uint64_t)60 * NANOSEC;  /* 1/minute */
129 dtrace_optval_t dtrace_aggrate_default = NANOSEC;              /* 1 hz */
130 dtrace_optval_t dtrace_statusrate_default = NANOSEC;           /* 1 hz */
131 dtrace_optval_t dtrace_statusrate_max = (hrtime_t)10 * NANOSEC;  /* 6/minute */
132 dtrace_optval_t dtrace_switchrate_default = NANOSEC;           /* 1 hz */
133 dtrace_optval_t dtrace_nspec_default = 1;
134 dtrace_optval_t dtrace_specsize_default = 32 * 1024;
135 dtrace_optval_t dtrace_stackframes_default = 20;
136 dtrace_optval_t dtrace_ustackframes_default = 20;
137 dtrace_optval_t dtrace_jstackframes_default = 50;
138 dtrace_optval_t dtrace_jstackstrsize_default = 512;
139 int             dtrace_msgdsize_max = 128;
140 hrtime_t        dtrace_chill_max = 500 * (NANOSEC / MILLISEC);  /* 500 ms */
141 hrtime_t        dtrace_chill_interval = NANOSEC;               /* 1000 ms */
142 int             dtrace_devdepth_max = 32;
143 int             dtrace_err_verbose;
144 hrtime_t        dtrace_deadman_interval = NANOSEC;
145 hrtime_t        dtrace_deadman_timeout = (hrtime_t)10 * NANOSEC;
146 hrtime_t        dtrace_deadman_user = (hrtime_t)30 * NANOSEC;
147 hrtime_t        dtrace_unregister_defunct_reap = (hrtime_t)60 * NANOSEC;

149 /*
150  * DTrace External Variables
151  *
152  * As dtrace(7D) is a kernel module, any DTrace variables are obviously
153  * available to DTrace consumers via the backtick (`) syntax.  One of these,
154  * dtrace_zero, is made deliberately so:  it is provided as a source of
155  * well-known, zero-filled memory.  While this variable is not documented,
156  * it is used by some translators as an implementation detail.
157  */
158 const char      dtrace_zero[256] = { 0 };       /* zero-filled memory */

160 /*
161  * DTrace Internal Variables
162  */
163 static dev_info_t       *dtrace_devi;           /* device info */
164 static vmem_t           *dtrace_arena;          /* probe ID arena */
165 static vmem_t           *dtrace_minor;          /* minor number arena */
166 static taskq_t          *dtrace_taskq;          /* task queue */
167 static dtrace_probe_t   **dtrace_probes;        /* array of all probes */
168 static int              dtrace_nprobes;         /* number of probes */
169 static dtrace_provider_t *dtrace_provider;      /* provider list */
170 static dtrace_meta_t    *dtrace_meta_pid;       /* user-land meta provider */
171 static int              dtrace_opens;           /* number of opens */
172 static int              dtrace_helpers;         /* number of helpers */
173 static int              dtrace_getf;            /* number of unpriv getf()s */
174 static void             *dtrace_softstate;      /* softstate pointer */
175 static dtrace_hash_t    *dtrace_bymod;          /* probes hashed by module */
176 static dtrace_hash_t    *dtrace_byfunc;         /* probes hashed by function */
177 static dtrace_hash_t    *dtrace_byname;         /* probes hashed by name */
178 static dtrace_toxrange_t *dtrace_toxrange;      /* toxic range array */
179 static int              dtrace_toxranges;       /* number of toxic ranges */
180 static int              dtrace_toxranges_max;   /* size of toxic range array */
181 static dtrace_anon_t    dtrace_anon;            /* anonymous enabling */
182 static kmem_cache_t     *dtrace_state_cache;    /* cache for dynamic state */
183 static uint64_t         dtrace_vtime_references; /* number of vtimestamp refs */
184 static kthread_t        *dtrace_panicked;       /* panicking thread */
185 static dtrace_ecb_t     *dtrace_ecb_create_cache; /* cached created ECB */
186 static dtrace_genid_t   dtrace_probegen;        /* current probe generation */
187 static dtrace_helpers_t *dtrace_deferred_pid;   /* deferred helper list */
188 static dtrace_enabling_t *dtrace_retained;      /* list of retained enablings */
189 static dtrace_genid_t   dtrace_retained_gen;    /* current retained enab gen */
190 static dtrace_dynvar_t  dtrace_dynhash_sink;    /* end of dynamic hash chains */
191 static int              dtrace_dynvar_failclean; /* dynvars failed to clean */

193 /*
```



```
194  * DTrace Locking
195  * DTrace is protected by three (relatively coarse-grained) locks:
196  *
197  * (1) dtrace_lock is required to manipulate essentially any DTrace state,
198  *     including enabling state, probes, ECBs, consumer state, helper state,
199  *     etc.  Importantly, dtrace_lock is _not_ required when in probe context;
200  *     probe context is lock-free -- synchronization is handled via the
201  *     dtrace_sync() cross call mechanism.
202  *
203  * (2) dtrace_provider_lock is required when manipulating provider state, or
204  *     when provider state must be held constant.
205  *
206  * (3) dtrace_meta_lock is required when manipulating meta provider state, or
207  *     when meta provider state must be held constant.
208  *
209  * The lock ordering between these three locks is dtrace_meta_lock before
210  * dtrace_provider_lock before dtrace_lock.  (In particular, there are
211  * several places where dtrace_provider_lock is held by the framework as it
212  * calls into the providers -- which then call back into the framework,
213  * grabbing dtrace_lock.)
214  *
215  * There are two other locks in the mix:  mod_lock and cpu_lock.  With respect
216  * to dtrace_provider_lock and dtrace_lock, cpu_lock continues its historical
217  * role as a coarse-grained lock; it is acquired before both of these locks.
218  * With respect to dtrace_meta_lock, its behavior is stranger:  cpu_lock must
219  * be acquired _between_ dtrace_meta_lock and any other DTrace locks.
220  * mod_lock is similar with respect to dtrace_provider_lock in that it must be
221  * acquired _between_ dtrace_provider_lock and dtrace_lock.
222  */
223 static kmutex_t         dtrace_lock;            /* probe state lock */
224 static kmutex_t         dtrace_provider_lock;   /* provider state lock */
225 static kmutex_t         dtrace_meta_lock;       /* meta-provider state lock */

227 /*
228  * DTrace Provider Variables
229  *
230  * These are the variables relating to DTrace as a provider (that is, the
231  * provider of the BEGIN, END, and ERROR probes).
232  */
233 static dtrace_pattr_t   dtrace_provider_attr = {
234 { DTRACE_STABILITY_STABLE, DTRACE_STABILITY_STABLE, DTRACE_CLASS_COMMON },
235 { DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
236 { DTRACE_STABILITY_PRIVATE, DTRACE_STABILITY_PRIVATE, DTRACE_CLASS_UNKNOWN },
237 { DTRACE_STABILITY_STABLE, DTRACE_STABILITY_STABLE, DTRACE_CLASS_COMMON },
238 { DTRACE_STABILITY_STABLE, DTRACE_STABILITY_STABLE, DTRACE_CLASS_COMMON },
239 };
_____unchanged_portion_omitted_

428 #ifdef _LP64
429 #define dtrace_loadptr  dtrace_load64
430 #else
431 #define dtrace_loadptr  dtrace_load32
432 #endif

434 #define DTRACE_DYNHASH_FREE     0
435 #define DTRACE_DYNHASH_SINK     1
436 #define DTRACE_DYNHASH_VALID    2

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

444 #define DTRACE_FLAGS2FLT(flags)                                         \
445         (((flags) & CPU_DTRACE_BADADDR) ? DTRACEFLT_BADADDR :           \
```

```
446         ((flags) & CPU_DTRACE_ILLOP) ? DTRACEFLT_ILLOP :                \
447         ((flags) & CPU_DTRACE_DIVZERO) ? DTRACEFLT_DIVZERO :            \
448         ((flags) & CPU_DTRACE_KPRIV) ? DTRACEFLT_KPRIV :                \
449         ((flags) & CPU_DTRACE_UPRIV) ? DTRACEFLT_UPRIV :                \
450         ((flags) & CPU_DTRACE_TUPOFLOW) ?  DTRACEFLT_TUPOFLOW :         \
451         ((flags) & CPU_DTRACE_BADALIGN) ?  DTRACEFLT_BADALIGN :         \
452         ((flags) & CPU_DTRACE_NOSCRATCH) ?  DTRACEFLT_NOSCRATCH :       \
453         ((flags) & CPU_DTRACE_BADSTACK) ?  DTRACEFLT_BADSTACK :         \
454         DTRACEFLT_UNKNOWN)

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

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

481 /*
482  * DTrace Probe Context Functions
483  *
484  * These functions are called from probe context.  Because probe context is
485  * any context in which C may be called, arbitrarily locks may be held,
486  * interrupts may be disabled, we may be in arbitrary dispatched state, etc.
487  * As a result, functions called from probe context may only call other DTrace
488  * support functions -- they may not interact at all with the system at large.
489  * (Note that the ASSERT macro is made probe-context safe by redefining it in
490  * terms of dtrace_assfail(), a probe-context safe function.) If arbitrary
491  * loads are to be performed from probe context, they _must_ be in terms of
492  * the safe dtrace_load*() variants.
493  *
494  * Some functions in this block are not actually called from probe context;
495  * for these functions, there will be a comment above the function reading
496  * "Note:  not called from probe context."
497  */
498 void
499 dtrace_panic(const char *format, ...)
500 {
501         va_list alist;

503         va_start(alist, format);
504         dtrace_vpanic(format, alist);
505         va_end(alist);
506 }
```

unchanged_portion_omitted_

```
677 /*
678  * Convenience routine to check to see if the address is within a memory
```



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

693         /*
694          * If we hold the privilege to read from kernel memory, then
695          * everything is readable.
696          */
697         if ((mstate->dtms_access & DTRACE_ACCESS_KERNEL) != 0)
698                 return (1);

700         /*
701          * You can obviously read that which you can store.
702          */
703         if (dtrace_canstore(addr, sz, mstate, vstate))
704                 return (1);

706         /*
707          * We're allowed to read from our own string table.
708          */
709         if (DTRACE_INRANGE(addr, sz, mstate->dtms_difo->dtdo_strtab,
710             mstate->dtms_difo->dtdo_strlen))
711                 return (1);

713         if (vstate->dtvs_state != NULL &&
714             dtrace_priv_proc(vstate->dtvs_state, mstate)) {
715                 proc_t *p;

717                 /*
718                  * When we have privileges to the current process, there are
719                  * several context-related kernel structures that are safe to
720                  * read, even absent the privilege to read from kernel memory.
721                  * These reads are safe because these structures contain only
722                  * state that (1) we're permitted to read, (2) is harmless or
723                  * (3) contains pointers to additional kernel state that we're
724                  * not permitted to read (and as such, do not present an
725                  * opportunity for privilege escalation).  Finally (and
726                  * critically), because of the nature of their relation with
727                  * the current thread context, the memory associated with these
728                  * structures cannot change over the duration of probe context,
729                  * and it is therefore impossible for this memory to be
730                  * deallocated and reallocated as something else while it's
731                  * being operated upon.
732                  */
733                 if (DTRACE_INRANGE(addr, sz, curthread, sizeof (kthread_t)))
734                         return (1);

736                 if ((p = curthread->t_procp) != NULL && DTRACE_INRANGE(addr,
737                     sz, curthread->t_procp, sizeof (proc_t))) {
738                         return (1);
739                 }

741                 if (curthread->t_cred != NULL && DTRACE_INRANGE(addr, sz,
742                     curthread->t_cred, sizeof (cred_t))) {
743                         return (1);
744                 }
```

```
746                 if (p != NULL && p->p_pidp != NULL && DTRACE_INRANGE(addr, sz,
747                     &(p->p_pidp->pid_id), sizeof (pid_t))) {
748                         return (1);
749                 }

751                 if (curthread->t_cpu != NULL && DTRACE_INRANGE(addr, sz,
752                     curthread->t_cpu, offsetof(cpu_t, cpu_pause_thread))) {
753                         return (1);
754                 }
755         }

757         if ((fp = mstate->dtms_getf) != NULL) {
758                 uintptr_t psz = sizeof (void *);
759                 vnode_t *vp;
760                 vnodeops_t *op;

762                 /*
763                  * When getf() returns a file_t, the enabling is implicitly
764                  * granted the (transient) right to read the returned file_t
765                  * as well as the v_path and v_op->vnop_name of the underlying
766                  * vnode.  These accesses are allowed after a successful
767                  * getf() because the members that they refer to cannot change
768                  * once set -- and the barrier logic in the kernel's closef()
769                  * path assures that the file_t and its referenced vode_t
770                  * cannot themselves be stale (that is, it impossible for
771                  * either dtms_getf itself or its f_vnode member to reference
772                  * freed memory).
773                  */
774                 if (DTRACE_INRANGE(addr, sz, fp, sizeof (file_t)))
775                         return (1);

777                 if ((vp = fp->f_vnode) != NULL) {
778                         if (DTRACE_INRANGE(addr, sz, &vp->v_path, psz))
779                                 return (1);

781                         if (vp->v_path != NULL && DTRACE_INRANGE(addr, sz,
782                             vp->v_path, strlen(vp->v_path) + 1)) {
783                                 return (1);
784                         }

786                         if (DTRACE_INRANGE(addr, sz, &vp->v_op, psz))
787                                 return (1);

789                         if ((op = vp->v_op) != NULL &&
790                             DTRACE_INRANGE(addr, sz, &op->vnop_name, psz)) {
791                                 return (1);
792                         }

794                         if (op != NULL && op->vnop_name != NULL &&
795                             DTRACE_INRANGE(addr, sz, op->vnop_name,
796                             strlen(op->vnop_name) + 1)) {
797                                 return (1);
798                         }
799                 }
800         }

802         DTRACE_CPUFLAG_SET(CPU_DTRACE_KPRIV);
803         *illval = addr;
804         return (0);
805 }
```

_____*unchanged_portion_omitted_*

```
1164 /*
1165  * This privilege check should be used by actions and subroutines to
1166  * verify that the zone of the process that enabled the invoking ECB
```



```
1167  * matches the target credentials
1168  */
1169 static int
1170 dtrace_priv_proc_common_zone(dtrace_state_t *state)
1171 {
1172         cred_t *cr, *s_cr = state->dts_cred.dcr_cred;

1174         /*
1175          * We should always have a non-NULL state cred here, since if cred
1176          * is null (anonymous tracing), we fast-path bypass this routine.
1177          */
1178         ASSERT(s_cr != NULL);

1180         if ((cr = CRED()) != NULL && s_cr->cr_zone == cr->cr_zone)
1132         if ((cr = CRED()) != NULL &&
1133             s_cr->cr_zone == cr->cr_zone)
1181                 return (1);

1183         return (0);
1184 }
```

_____*unchanged_portion_omitted_*

```
3299 /*
3300  * Emulate the execution of DTrace ID subroutines invoked by the call opcode.
3301  * Notice that we don't bother validating the proper number of arguments or
3302  * their types in the tuple stack.  This isn't needed because all argument
3303  * interpretation is safe because of our load safety -- the worst that can
3304  * happen is that a bogus program can obtain bogus results.
3305  */
3306 static void
3307 dtrace_dif_subr(uint_t subr, uint_t rd, uint64_t *regs,
3308     dtrace_key_t *tupregs, int nargs,
3309     dtrace_mstate_t *mstate, dtrace_state_t *state)
3310 {
3311         volatile uint16_t *flags = &cpu_core[CPU->cpu_id].cpuc_dtrace_flags;
3312         volatile uintptr_t *illval = &cpu_core[CPU->cpu_id].cpuc_dtrace_illval;
3313         dtrace_vstate_t *vstate = &state->dts_vstate;

3315         union {
3316                 mutex_impl_t mi;
3317                 uint64_t mx;
3318         } m;

3320         union {
3321                 krwlock_t ri;
3322                 uintptr_t rw;
3323         } r;

3325         switch (subr) {
3326         case DIF_SUBR_RAND:
3327                 regs[rd] = (dtrace_gethrtime() * 2416 + 374441) % 1771875;
3328                 break;

3330         case DIF_SUBR_MUTEX_OWNED:
3331                 if (!dtrace_canload(tupregs[0].dttk_value, sizeof (kmutex_t),
3332                     mstate, vstate)) {
3333                         regs[rd] = NULL;
3334                         break;
3335                 }

3337                 m.mx = dtrace_load64(tupregs[0].dttk_value);
3338                 if (MUTEX_TYPE_ADAPTIVE(&m.mi))
3339                         regs[rd] = MUTEX_OWNER(&m.mi) != MUTEX_NO_OWNER;
3340                 else
3341                         regs[rd] = LOCK_HELD(&m.mi.m_spin.m_spinlock);
3342                 break;
```

```
3344         case DIF_SUBR_MUTEX_OWNER:
3345                 if (!dtrace_canload(tupregs[0].dttk_value, sizeof (kmutex_t),
3346                     mstate, vstate)) {
3347                         regs[rd] = NULL;
3348                         break;
3349                 }

3351                 m.mx = dtrace_load64(tupregs[0].dttk_value);
3352                 if (MUTEX_TYPE_ADAPTIVE(&m.mi) &&
3353                     MUTEX_OWNER(&m.mi) != MUTEX_NO_OWNER)
3354                         regs[rd] = (uintptr_t)MUTEX_OWNER(&m.mi);
3355                 else
3356                         regs[rd] = 0;
3357                 break;

3359         case DIF_SUBR_MUTEX_TYPE_ADAPTIVE:
3360                 if (!dtrace_canload(tupregs[0].dttk_value, sizeof (kmutex_t),
3361                     mstate, vstate)) {
3362                         regs[rd] = NULL;
3363                         break;
3364                 }

3366                 m.mx = dtrace_load64(tupregs[0].dttk_value);
3367                 regs[rd] = MUTEX_TYPE_ADAPTIVE(&m.mi);
3368                 break;

3370         case DIF_SUBR_MUTEX_TYPE_SPIN:
3371                 if (!dtrace_canload(tupregs[0].dttk_value, sizeof (kmutex_t),
3372                     mstate, vstate)) {
3373                         regs[rd] = NULL;
3374                         break;
3375                 }

3377                 m.mx = dtrace_load64(tupregs[0].dttk_value);
3378                 regs[rd] = MUTEX_TYPE_SPIN(&m.mi);
3379                 break;

3381         case DIF_SUBR_RW_READ_HELD: {
3382                 uintptr_t tmp;

3384                 if (!dtrace_canload(tupregs[0].dttk_value, sizeof (uintptr_t),
3385                     mstate, vstate)) {
3386                         regs[rd] = NULL;
3387                         break;
3388                 }

3390                 r.rw = dtrace_loadptr(tupregs[0].dttk_value);
3391                 regs[rd] = _RW_READ_HELD(&r.ri, tmp);
3392                 break;
3393         }

3395         case DIF_SUBR_RW_WRITE_HELD:
3396                 if (!dtrace_canload(tupregs[0].dttk_value, sizeof (krwlock_t),
3397                     mstate, vstate)) {
3398                         regs[rd] = NULL;
3399                         break;
3400                 }

3402                 r.rw = dtrace_loadptr(tupregs[0].dttk_value);
3403                 regs[rd] = _RW_WRITE_HELD(&r.ri);
3404                 break;

3406         case DIF_SUBR_RW_ISWRITER:
3407                 if (!dtrace_canload(tupregs[0].dttk_value, sizeof (krwlock_t),
3408                     mstate, vstate)) {
```



```
3409                         regs[rd] = NULL;
3410                         break;
3411                 }

3413                 r.rw = dtrace_loadptr(tupregs[0].dttk_value);
3414                 regs[rd] = _RW_ISWRITER(&r.ri);
3415                 break;

3417         case DIF_SUBR_BCOPY: {
3418                 /*
3419                  * We need to be sure that the destination is in the scratch
3420                  * region -- no other region is allowed.
3421                  */
3422                 uintptr_t src = tupregs[0].dttk_value;
3423                 uintptr_t dest = tupregs[1].dttk_value;
3424                 size_t size = tupregs[2].dttk_value;

3426                 if (!dtrace_inscratch(dest, size, mstate)) {
3427                         *flags |= CPU_DTRACE_BADADDR;
3428                         *illval = regs[rd];
3429                         break;
3430                 }

3432                 if (!dtrace_canload(src, size, mstate, vstate)) {
3433                         regs[rd] = NULL;
3434                         break;
3435                 }

3437                 dtrace_bcopy((void *)src, (void *)dest, size);
3438                 break;
3439         }

3441         case DIF_SUBR_ALLOCA:
3442         case DIF_SUBR_COPYIN: {
3443                 uintptr_t dest = P2ROUNDUP(mstate->dtms_scratch_ptr, 8);
3444                 uint64_t size =
3445                     tupregs[subr == DIF_SUBR_ALLOCA ? 0 : 1].dttk_value;
3446                 size_t scratch_size = (dest - mstate->dtms_scratch_ptr) + size;

3448                 /*
3449                  * This action doesn't require any credential checks since
3450                  * probes will not activate in user contexts to which the
3451                  * enabling user does not have permissions.
3452                  */

3454                 /*
3455                  * Rounding up the user allocation size could have overflowed
3456                  * a large, bogus allocation (like -1ULL) to 0.
3457                  */
3458                 if (scratch_size < size ||
3459                     !DTRACE_INSCRATCH(mstate, scratch_size)) {
3460                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
3461                         regs[rd] = NULL;
3462                         break;
3463                 }

3465                 if (subr == DIF_SUBR_COPYIN) {
3466                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
3467                         dtrace_copyin(tupregs[0].dttk_value, dest, size, flags);
3468                         DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
3469                 }

3471                 mstate->dtms_scratch_ptr += scratch_size;
3472                 regs[rd] = dest;
3473                 break;
3474         }
```

```
3476         case DIF_SUBR_COPYINTO: {
3477                 uint64_t size = tupregs[1].dttk_value;
3478                 uintptr_t dest = tupregs[2].dttk_value;
3479
3480                 /*
3481                  * This action doesn't require any credential checks since
3482                  * probes will not activate in user contexts to which the
3483                  * enabling user does not have permissions.
3484                  */
3485                 if (!dtrace_inscratch(dest, size, mstate)) {
3486                         *flags |= CPU_DTRACE_BADADDR;
3487                         *illval = regs[rd];
3488                         break;
3489                 }
3490
3491                 DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
3492                 dtrace_copyin(tupregs[0].dttk_value, dest, size, flags);
3493                 DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
3494                 break;
3495         }
3496
3497         case DIF_SUBR_COPYINSTR: {
3498                 uintptr_t dest = mstate->dtms_scratch_ptr;
3499                 uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
3500
3501                 if (nargs > 1 && tupregs[1].dttk_value < size)
3502                         size = tupregs[1].dttk_value + 1;
3503
3504                 /*
3505                  * This action doesn't require any credential checks since
3506                  * probes will not activate in user contexts to which the
3507                  * enabling user does not have permissions.
3508                  */
3509                 if (!DTRACE_INSCRATCH(mstate, size)) {
3510                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
3511                         regs[rd] = NULL;
3512                         break;
3513                 }
3514
3515                 DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
3516                 dtrace_copyinstr(tupregs[0].dttk_value, dest, size, flags);
3517                 DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
3518
3519                 ((char *)dest)[size - 1] = '\0';
3520                 mstate->dtms_scratch_ptr += size;
3521                 regs[rd] = dest;
3522                 break;
3523         }
3524
3525         case DIF_SUBR_MSGSIZE:
3526         case DIF_SUBR_MSGDSIZE: {
3527                 uintptr_t baddr = tupregs[0].dttk_value, daddr;
3528                 uintptr_t wptr, rptr;
3529                 size_t count = 0;
3530                 int cont = 0;
3531
3532                 while (baddr != NULL && !(*flags & CPU_DTRACE_FAULT)) {
3533
3534                         if (!dtrace_canload(baddr, sizeof (mblk_t), mstate,
3535                             vstate)) {
3536                                 regs[rd] = NULL;
3537                                 break;
3538                         }
3539
3540                         wptr = dtrace_loadptr(baddr +
```



```
3541                             offsetof(mblk_t, b_wptr));
3542
3543                         rptr = dtrace_loadptr(baddr +
3544                             offsetof(mblk_t, b_rptr));
3545
3546                         if (wptr < rptr) {
3547                                 *flags |= CPU_DTRACE_BADADDR;
3548                                 *illval = tupregs[0].dttk_value;
3549                                 break;
3550                         }
3551
3552                         daddr = dtrace_loadptr(baddr +
3553                             offsetof(mblk_t, b_datap));
3554
3555                         baddr = dtrace_loadptr(baddr +
3556                             offsetof(mblk_t, b_cont));
3557
3558                         /*
3559                          * We want to prevent against denial-of-service here,
3560                          * so we're only going to search the list for
3561                          * dtrace_msgdsize_max mblks.
3562                          */
3563                         if (cont++ > dtrace_msgdsize_max) {
3564                                 *flags |= CPU_DTRACE_ILLOP;
3565                                 break;
3566                         }
3567
3568                         if (subr == DIF_SUBR_MSGDSIZE) {
3569                                 if (dtrace_load8(daddr +
3570                                     offsetof(dblk_t, db_type)) != M_DATA)
3571                                         continue;
3572                         }
3573
3574                         count += wptr - rptr;
3575                 }
3576
3577                 if (!(*flags & CPU_DTRACE_FAULT))
3578                         regs[rd] = count;
3579
3580                 break;
3581         }
3582
3583         case DIF_SUBR_PROGENYOF: {
3584                 pid_t pid = tupregs[0].dttk_value;
3585                 proc_t *p;
3586                 int rval = 0;
3587
3588                 DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
3589
3590                 for (p = curthread->t_procp; p != NULL; p = p->p_parent) {
3591                         if (p->p_pidp->pid_id == pid) {
3592                                 rval = 1;
3593                                 break;
3594                         }
3595                 }
3596
3597                 DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
3598
3599                 regs[rd] = rval;
3600                 break;
3601         }
3602
3603         case DIF_SUBR_SPECULATION:
3604                 regs[rd] = dtrace_speculation(state);
3605                 break;
```

```
3607         case DIF_SUBR_COPYOUT: {
3608                 uintptr_t kaddr = tupregs[0].dttk_value;
3609                 uintptr_t uaddr = tupregs[1].dttk_value;
3610                 uint64_t size = tupregs[2].dttk_value;

3612                 if (!dtrace_destructive_disallow &&
3613                     dtrace_priv_proc_control(state, mstate) &&
3614                     !dtrace_istoxic(kaddr, size)) {
3615                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
3616                         dtrace_copyout(kaddr, uaddr, size, flags);
3617                         DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
3618                 }
3619                 break;
3620         }

3622         case DIF_SUBR_COPYOUTSTR: {
3623                 uintptr_t kaddr = tupregs[0].dttk_value;
3624                 uintptr_t uaddr = tupregs[1].dttk_value;
3625                 uint64_t size = tupregs[2].dttk_value;

3627                 if (!dtrace_destructive_disallow &&
3628                     dtrace_priv_proc_control(state, mstate) &&
3629                     !dtrace_istoxic(kaddr, size)) {
3630                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
3631                         dtrace_copyoutstr(kaddr, uaddr, size, flags);
3632                         DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
3633                 }
3634                 break;
3635         }

3637         case DIF_SUBR_STRLEN: {
3638                 size_t sz;
3639                 uintptr_t addr = (uintptr_t)tupregs[0].dttk_value;
3640                 sz = dtrace_strlen((char *)addr,
3641                     state->dts_options[DTRACEOPT_STRSIZE]);

3643                 if (!dtrace_canload(addr, sz + 1, mstate, vstate)) {
3644                         regs[rd] = NULL;
3645                         break;
3646                 }

3648                 regs[rd] = sz;

3650                 break;
3651         }

3653         case DIF_SUBR_STRCHR:
3654         case DIF_SUBR_STRRCHR: {
3655                 /*
3656                  * We're going to iterate over the string looking for the
3657                  * specified character.  We will iterate until we have reached
3658                  * the string length or we have found the character.  If this
3659                  * is DIF_SUBR_STRRCHR, we will look for the last occurrence
3660                  * of the specified character instead of the first.
3661                  */
3662                 uintptr_t saddr = tupregs[0].dttk_value;
3663                 uintptr_t addr = tupregs[0].dttk_value;
3664                 uintptr_t limit = addr + state->dts_options[DTRACEOPT_STRSIZE];
3665                 char c, target = (char)tupregs[1].dttk_value;

3667                 for (regs[rd] = NULL; addr < limit; addr++) {
3668                         if ((c = dtrace_load8(addr)) == target) {
3669                                 regs[rd] = addr;

3671                                 if (subr == DIF_SUBR_STRCHR)
3672                                         break;
```



```
3673                         }

3675                         if (c == '\0')
3676                                 break;
3677                 }

3679                 if (!dtrace_canload(saddr, addr - saddr, mstate, vstate)) {
3680                         regs[rd] = NULL;
3681                         break;
3682                 }

3684                 break;
3685         }

3687         case DIF_SUBR_STRSTR:
3688         case DIF_SUBR_INDEX:
3689         case DIF_SUBR_RINDEX: {
3690                 /*
3691                  * We're going to iterate over the string looking for the
3692                  * specified string.  We will iterate until we have reached
3693                  * the string length or we have found the string.  (Yes, this
3694                  * is done in the most naive way possible -- but considering
3695                  * that the string we're searching for is likely to be
3696                  * relatively short, the complexity of Rabin-Karp or similar
3697                  * hardly seems merited.)
3698                  */
3699                 char *addr = (char *)(uintptr_t)tupregs[0].dttk_value;
3700                 char *substr = (char *)(uintptr_t)tupregs[1].dttk_value;
3701                 uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
3702                 size_t len = dtrace_strlen(addr, size);
3703                 size_t sublen = dtrace_strlen(substr, size);
3704                 char *limit = addr + len, *orig = addr;
3705                 int notfound = subr == DIF_SUBR_STRSTR ? 0 : -1;
3706                 int inc = 1;

3708                 regs[rd] = notfound;

3710                 if (!dtrace_canload((uintptr_t)addr, len + 1, mstate, vstate)) {
3711                         regs[rd] = NULL;
3712                         break;
3713                 }

3715                 if (!dtrace_canload((uintptr_t)substr, sublen + 1, mstate,
3716                     vstate)) {
3717                         regs[rd] = NULL;
3718                         break;
3719                 }

3721                 /*
3722                  * strstr() and index()/rindex() have similar semantics if
3723                  * both strings are the empty string: strstr() returns a
3724                  * pointer to the (empty) string, and index() and rindex()
3725                  * both return index 0 (regardless of any position argument).
3726                  */
3727                 if (sublen == 0 && len == 0) {
3728                         if (subr == DIF_SUBR_STRSTR)
3729                                 regs[rd] = (uintptr_t)addr;
3730                         else
3731                                 regs[rd] = 0;
3732                         break;
3733                 }

3735                 if (subr != DIF_SUBR_STRSTR) {
3736                         if (subr == DIF_SUBR_RINDEX) {
3737                                 limit = orig - 1;
3738                                 addr += len;
```

```
3739                            inc = -1;
3740                    }
3741
3742                    /*
3743                     * Both index() and rindex() take an optional position
3744                     * argument that denotes the starting position.
3745                     */
3746                    if (nargs == 3) {
3747                            int64_t pos = (int64_t)tupregs[2].dttk_value;
3748
3749                            /*
3750                             * If the position argument to index() is
3751                             * negative, Perl implicitly clamps it at
3752                             * zero.  This semantic is a little surprising
3753                             * given the special meaning of negative
3754                             * positions to similar Perl functions like
3755                             * substr(), but it appears to reflect a
3756                             * notion that index() can start from a
3757                             * negative index and increment its way up to
3758                             * the string.  Given this notion, Perl's
3759                             * rindex() is at least self-consistent in
3760                             * that it implicitly clamps positions greater
3761                             * than the string length to be the string
3762                             * length.  Where Perl completely loses
3763                             * coherence, however, is when the specified
3764                             * substring is the empty string ("").  In
3765                             * this case, even if the position is
3766                             * negative, rindex() returns 0 -- and even if
3767                             * the position is greater than the length,
3768                             * index() returns the string length.  These
3769                             * semantics violate the notion that index()
3770                             * should never return a value less than the
3771                             * specified position and that rindex() should
3772                             * never return a value greater than the
3773                             * specified position.  (One assumes that
3774                             * these semantics are artifacts of Perl's
3775                             * implementation and not the results of
3776                             * deliberate design -- it beggars belief that
3777                             * even Larry Wall could desire such oddness.)
3778                             * While in the abstract one would wish for
3779                             * consistent position semantics across
3780                             * substr(), index() and rindex() -- or at the
3781                             * very least self-consistent position
3782                             * semantics for index() and rindex() -- we
3783                             * instead opt to keep with the extant Perl
3784                             * semantics, in all their broken glory.  (Do
3785                             * we have more desire to maintain Perl's
3786                             * semantics than Perl does?  Probably.)
3787                             */
3788                            if (subr == DIF_SUBR_RINDEX) {
3789                                    if (pos < 0) {
3790                                            if (sublen == 0)
3791                                                    regs[rd] = 0;
3792                                            break;
3793                                    }
3794
3795                                    if (pos > len)
3796                                            pos = len;
3797                            } else {
3798                                    if (pos < 0)
3799                                            pos = 0;
3800
3801                                    if (pos >= len) {
3802                                            if (sublen == 0)
3803                                                    regs[rd] = len;
3804                                            break;
```



```
3805                                    }
3806                            }
3807
3808                            addr = orig + pos;
3809                    }
3810            }
3811
3812            for (regs[rd] = notfound; addr != limit; addr += inc) {
3813                    if (dtrace_strncmp(addr, substr, sublen) == 0) {
3814                            if (subr != DIF_SUBR_STRSTR) {
3815                                    /*
3816                                     * As D index() and rindex() are
3817                                     * modeled on Perl (and not on awk),
3818                                     * we return a zero-based (and not a
3819                                     * one-based) index.  (For you Perl
3820                                     * weenies: no, we're not going to add
3821                                     * $[ -- and shouldn't you be at a con
3822                                     * or something?)
3823                                     */
3824                                    regs[rd] = (uintptr_t)(addr - orig);
3825                                    break;
3826                            }
3827
3828                            ASSERT(subr == DIF_SUBR_STRSTR);
3829                            regs[rd] = (uintptr_t)addr;
3830                            break;
3831                    }
3832            }
3833
3834            break;
3835        }
3836
3837        case DIF_SUBR_STRTOK: {
3838                uintptr_t addr = tupregs[0].dttk_value;
3839                uintptr_t tokaddr = tupregs[1].dttk_value;
3840                uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
3841                uintptr_t limit, toklimit = tokaddr + size;
3842                uint8_t c, tokmap[32];   /* 256 / 8 */
3843                char *dest = (char *)mstate->dtms_scratch_ptr;
3844                int i;
3845
3846                /*
3847                 * Check both the token buffer and (later) the input buffer,
3848                 * since both could be non-scratch addresses.
3849                 */
3850                if (!dtrace_strcanload(tokaddr, size, mstate, vstate)) {
3851                        regs[rd] = NULL;
3852                        break;
3853                }
3854
3855                if (!DTRACE_INSCRATCH(mstate, size)) {
3856                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
3857                        regs[rd] = NULL;
3858                        break;
3859                }
3860
3861                if (addr == NULL) {
3862                        /*
3863                         * If the address specified is NULL, we use our saved
3864                         * strtok pointer from the mstate.  Note that this
3865                         * means that the saved strtok pointer is _only_
3866                         * valid within multiple enablings of the same probe --
3867                         * it behaves like an implicit clause-local variable.
3868                         */
3869                        addr = mstate->dtms_strtok;
3870                } else {
```

```
3871                        /*
3872                         * If the user-specified address is non-NULL we must
3873                         * access check it.  This is the only time we have
3874                         * a chance to do so, since this address may reside
3875                         * in the string table of this clause-- future calls
3876                         * (when we fetch addr from mstate->dtms_strtok)
3877                         * would fail this access check.
3878                         */
3879                        if (!dtrace_strcanload(addr, size, mstate, vstate)) {
3880                                regs[rd] = NULL;
3881                                break;
3882                        }
3883                }

3885                /*
3886                 * First, zero the token map, and then process the token
3887                 * string -- setting a bit in the map for every character
3888                 * found in the token string.
3889                 */
3890                for (i = 0; i < sizeof (tokmap); i++)
3891                        tokmap[i] = 0;

3893                for (; tokaddr < toklimit; tokaddr++) {
3894                        if ((c = dtrace_load8(tokaddr)) == '\0')
3895                                break;

3897                        ASSERT((c >> 3) < sizeof (tokmap));
3898                        tokmap[c >> 3] |= (1 << (c & 0x7));
3899                }

3901                for (limit = addr + size; addr < limit; addr++) {
3902                        /*
3903                         * We're looking for a character that is _not_ contained
3904                         * in the token string.
3905                         */
3906                        if ((c = dtrace_load8(addr)) == '\0')
3907                                break;

3909                        if (!(tokmap[c >> 3] & (1 << (c & 0x7))))
3910                                break;
3911                }

3913                if (c == '\0') {
3914                        /*
3915                         * We reached the end of the string without finding
3916                         * any character that was not in the token string.
3917                         * We return NULL in this case, and we set the saved
3918                         * address to NULL as well.
3919                         */
3920                        regs[rd] = NULL;
3921                        mstate->dtms_strtok = NULL;
3922                        break;
3923                }

3925                /*
3926                 * From here on, we're copying into the destination string.
3927                 */
3928                for (i = 0; addr < limit && i < size - 1; addr++) {
3929                        if ((c = dtrace_load8(addr)) == '\0')
3930                                break;

3932                        if (tokmap[c >> 3] & (1 << (c & 0x7)))
3933                                break;

3935                        ASSERT(i < size);
3936                        dest[i++] = c;
```



```
3937                }

3939                ASSERT(i < size);
3940                dest[i] = '\0';
3941                regs[rd] = (uintptr_t)dest;
3942                mstate->dtms_scratch_ptr += size;
3943                mstate->dtms_strtok = addr;
3944                break;
3945        }

3947        case DIF_SUBR_SUBSTR: {
3948                uintptr_t s = tupregs[0].dttk_value;
3949                uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
3950                char *d = (char *)mstate->dtms_scratch_ptr;
3951                int64_t index = (int64_t)tupregs[1].dttk_value;
3952                int64_t remaining = (int64_t)tupregs[2].dttk_value;
3953                size_t len = dtrace_strlen((char *)s, size);
3954                int64_t i;

3956                if (!dtrace_canload(s, len + 1, mstate, vstate)) {
3957                        regs[rd] = NULL;
3958                        break;
3959                }

3961                if (!DTRACE_INSCRATCH(mstate, size)) {
3962                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
3963                        regs[rd] = NULL;
3964                        break;
3965                }

3967                if (nargs <= 2)
3968                        remaining = (int64_t)size;

3970                if (index < 0) {
3971                        index += len;

3973                        if (index < 0 && index + remaining > 0) {
3974                                remaining += index;
3975                                index = 0;
3976                        }
3977                }

3979                if (index >= len || index < 0) {
3980                        remaining = 0;
3981                } else if (remaining < 0) {
3982                        remaining += len - index;
3983                } else if (index + remaining > size) {
3984                        remaining = size - index;
3985                }

3987                for (i = 0; i < remaining; i++) {
3988                        if ((d[i] = dtrace_load8(s + index + i)) == '\0')
3989                                break;
3990                }

3992                d[i] = '\0';

3994                mstate->dtms_scratch_ptr += size;
3995                regs[rd] = (uintptr_t)d;
3996                break;
3997        }

3999        case DIF_SUBR_TOUPPER:
4000        case DIF_SUBR_TOLOWER: {
4001                uintptr_t s = tupregs[0].dttk_value;
4002                uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
```

```
4003                 char *dest = (char *)mstate->dtms_scratch_ptr, c;
4004                 size_t len = dtrace_strlen((char *)s, size);
4005                 char lower, upper, convert;
4006                 int64_t i;
4007
4008                 if (subr == DIF_SUBR_TOUPPER) {
4009                         lower = 'a';
4010                         upper = 'z';
4011                         convert = 'A';
4012                 } else {
4013                         lower = 'A';
4014                         upper = 'Z';
4015                         convert = 'a';
4016                 }
4017
4018                 if (!dtrace_canload(s, len + 1, mstate, vstate)) {
4019                         regs[rd] = NULL;
4020                         break;
4021                 }
4022
4023                 if (!DTRACE_INSCRATCH(mstate, size)) {
4024                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4025                         regs[rd] = NULL;
4026                         break;
4027                 }
4028
4029                 for (i = 0; i < size - 1; i++) {
4030                         if ((c = dtrace_load8(s + i)) == '\0')
4031                                 break;
4032
4033                         if (c >= lower && c <= upper)
4034                                 c = convert + (c - lower);
4035
4036                         dest[i] = c;
4037                 }
4038
4039                 ASSERT(i < size);
4040                 dest[i] = '\0';
4041                 regs[rd] = (uintptr_t)dest;
4042                 mstate->dtms_scratch_ptr += size;
4043                 break;
4044         }
4045
4046 case DIF_SUBR_GETMAJOR:
4047 #ifdef _LP64
4048                 regs[rd] = (tupregs[0].dttk_value >> NBITSMINOR64) & MAXMAJ64;
4049 #else
4050                 regs[rd] = (tupregs[0].dttk_value >> NBITSMINOR) & MAXMAJ;
4051 #endif
4052                 break;
4053
4054         case DIF_SUBR_GETMINOR:
4055 #ifdef _LP64
4056                 regs[rd] = tupregs[0].dttk_value & MAXMIN64;
4057 #else
4058                 regs[rd] = tupregs[0].dttk_value & MAXMIN;
4059 #endif
4060                 break;
4061
4062         case DIF_SUBR_DDI_PATHNAME: {
4063                 /*
4064                  * This one is a galactic mess.  We are going to roughly
4065                  * emulate ddi_pathname(), but it's made more complicated
4066                  * by the fact that we (a) want to include the minor name and
4067                  * (b) must proceed iteratively instead of recursively.
4068                  */
```



```
4069                 uintptr_t dest = mstate->dtms_scratch_ptr;
4070                 uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
4071                 char *start = (char *)dest, *end = start + size - 1;
4072                 uintptr_t daddr = tupregs[0].dttk_value;
4073                 int64_t minor = (int64_t)tupregs[1].dttk_value;
4074                 char *s;
4075                 int i, len, depth = 0;
4076
4077                 /*
4078                  * Due to all the pointer jumping we do and context we must
4079                  * rely upon, we just mandate that the user must have kernel
4080                  * read privileges to use this routine.
4081                  */
4082                 if ((mstate->dtms_access & DTRACE_ACCESS_KERNEL) == 0) {
4083                         *flags |= CPU_DTRACE_KPRIV;
4084                         *illval = daddr;
4085                         regs[rd] = NULL;
4086                 }
4087
4088                 if (!DTRACE_INSCRATCH(mstate, size)) {
4089                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4090                         regs[rd] = NULL;
4091                         break;
4092                 }
4093
4094                 *end = '\0';
4095
4096                 /*
4097                  * We want to have a name for the minor.  In order to do this,
4098                  * we need to walk the minor list from the devinfo.  We want
4099                  * to be sure that we don't infinitely walk a circular list,
4100                  * so we check for circularity by sending a scout pointer
4101                  * ahead two elements for every element that we iterate over;
4102                  * if the list is circular, these will ultimately point to the
4103                  * same element.  You may recognize this little trick as the
4104                  * answer to a stupid interview question -- one that always
4105                  * seems to be asked by those who had to have it laboriously
4106                  * explained to them, and who can't even concisely describe
4107                  * the conditions under which one would be forced to resort to
4108                  * this technique.  Needless to say, those conditions are
4109                  * found here -- and probably only here.  Is this the only use
4110                  * of this infamous trick in shipping, production code?  If it
4111                  * isn't, it probably should be...
4112                  */
4113                 if (minor != -1) {
4114                         uintptr_t maddr = dtrace_loadptr(daddr +
4115                             offsetof(struct dev_info, devi_minor));
4116
4117                         uintptr_t next = offsetof(struct ddi_minor_data, next);
4118                         uintptr_t name = offsetof(struct ddi_minor_data,
4119                             d_minor) + offsetof(struct ddi_minor, name);
4120                         uintptr_t dev = offsetof(struct ddi_minor_data,
4121                             d_minor) + offsetof(struct ddi_minor, dev);
4122                         uintptr_t scout;
4123
4124                         if (maddr != NULL)
4125                                 scout = dtrace_loadptr(maddr + next);
4126
4127                         while (maddr != NULL && !(*flags & CPU_DTRACE_FAULT)) {
4128                                 uint64_t m;
4129 #ifdef _LP64
4130                                 m = dtrace_load64(maddr + dev) & MAXMIN64;
4131 #else
4132                                 m = dtrace_load32(maddr + dev) & MAXMIN;
4133 #endif
4134                                 if (m != minor) {
```

```
4135                                    maddr = dtrace_loadptr(maddr + next);

4137                                    if (scout == NULL)
4138                                            continue;

4140                                    scout = dtrace_loadptr(scout + next);

4142                                    if (scout == NULL)
4143                                            continue;

4145                                    scout = dtrace_loadptr(scout + next);

4147                                    if (scout == NULL)
4148                                            continue;

4150                                    if (scout == maddr) {
4151                                            *flags |= CPU_DTRACE_ILLOP;
4152                                            break;
4153                                    }

4155                                    continue;
4156                            }

4158                            /*
4159                             * We have the minor data.  Now we need to
4160                             * copy the minor's name into the end of the
4161                             * pathname.
4162                             */
4163                            s = (char *)dtrace_loadptr(maddr + name);
4164                            len = dtrace_strlen(s, size);

4166                            if (*flags & CPU_DTRACE_FAULT)
4167                                    break;

4169                            if (len != 0) {
4170                                    if ((end -= (len + 1)) < start)
4171                                            break;

4173                                    *end = ':';
4174                            }

4176                            for (i = 1; i <= len; i++)
4177                                    end[i] = dtrace_load8((uintptr_t)s++);
4178                            break;
4179                    }
4180            }

4182            while (daddr != NULL && !(*flags & CPU_DTRACE_FAULT)) {
4183                    ddi_node_state_t devi_state;

4185                    devi_state = dtrace_load32(daddr +
4186                        offsetof(struct dev_info, devi_node_state));

4188                    if (*flags & CPU_DTRACE_FAULT)
4189                            break;

4191                    if (devi_state >= DS_INITIALIZED) {
4192                            s = (char *)dtrace_loadptr(daddr +
4193                                offsetof(struct dev_info, devi_addr));
4194                            len = dtrace_strlen(s, size);

4196                            if (*flags & CPU_DTRACE_FAULT)
4197                                    break;

4199                            if (len != 0) {
4200                                    if ((end -= (len + 1)) < start)
```



```
4201                                            break;

4203                                    *end = '@';
4204                            }

4206                            for (i = 1; i <= len; i++)
4207                                    end[i] = dtrace_load8((uintptr_t)s++);
4208                    }

4210                    /*
4211                     * Now for the node name...
4212                     */
4213                    s = (char *)dtrace_loadptr(daddr +
4214                        offsetof(struct dev_info, devi_node_name));

4216                    daddr = dtrace_loadptr(daddr +
4217                        offsetof(struct dev_info, devi_parent));

4219                    /*
4220                     * If our parent is NULL (that is, if we're the root
4221                     * node), we're going to use the special path
4222                     * "devices".
4223                     */
4224                    if (daddr == NULL)
4225                            s = "devices";

4227                    len = dtrace_strlen(s, size);
4228                    if (*flags & CPU_DTRACE_FAULT)
4229                            break;

4231                    if ((end -= (len + 1)) < start)
4232                            break;

4234                    for (i = 1; i <= len; i++)
4235                            end[i] = dtrace_load8((uintptr_t)s++);
4236                    *end = '/';

4238                    if (depth++ > dtrace_devdepth_max) {
4239                            *flags |= CPU_DTRACE_ILLOP;
4240                            break;
4241                    }
4242            }

4244            if (end < start)
4245                    DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);

4247            if (daddr == NULL) {
4248                    regs[rd] = (uintptr_t)end;
4249                    mstate->dtms_scratch_ptr += size;
4250            }

4252            break;
4253    }

4255    case DIF_SUBR_STRJOIN: {
4256            char *d = (char *)mstate->dtms_scratch_ptr;
4257            uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
4258            uintptr_t s1 = tupregs[0].dttk_value;
4259            uintptr_t s2 = tupregs[1].dttk_value;
4260            int i = 0;

4262            if (!dtrace_strcanload(s1, size, mstate, vstate) ||
4263                !dtrace_strcanload(s2, size, mstate, vstate)) {
4264                    regs[rd] = NULL;
4265                    break;
4266            }
```

```
4268                if (!DTRACE_INSCRATCH(mstate, size)) {
4269                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4270                        regs[rd] = NULL;
4271                        break;
4272                }

4274                for (;;) {
4275                        if (i >= size) {
4276                                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4277                                regs[rd] = NULL;
4278                                break;
4279                        }

4281                        if ((d[i++] = dtrace_load8(s1++)) == '\0') {
4282                                i--;
4283                                break;
4284                        }
4285                }

4287                for (;;) {
4288                        if (i >= size) {
4289                                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4290                                regs[rd] = NULL;
4291                                break;
4292                        }

4294                        if ((d[i++] = dtrace_load8(s2++)) == '\0')
4295                                break;
4296                }

4298                if (i < size) {
4299                        mstate->dtms_scratch_ptr += i;
4300                        regs[rd] = (uintptr_t)d;
4301                }

4303                break;
4304        }

4306        case DIF_SUBR_LLTOSTR: {
4307                int64_t i = (int64_t)tupregs[0].dttk_value;
4308                uint64_t val, digit;
4309                uint64_t size = 65;     /* enough room for 2^64 in binary */
4310                char *end = (char *)mstate->dtms_scratch_ptr + size - 1;
4311                int base = 10;

4313                if (nargs > 1) {
4314                        if ((base = tupregs[1].dttk_value) <= 1 ||
4315                            base > ('z' - 'a' + 1) + ('9' - '0' + 1)) {
4316                                *flags |= CPU_DTRACE_ILLOP;
4317                                break;
4318                        }
4319                }

4321                val = (base == 10 && i < 0) ? i * -1 : i;

4323                if (!DTRACE_INSCRATCH(mstate, size)) {
4324                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4325                        regs[rd] = NULL;
4326                        break;
4327                }

4329                for (*end-- = '\0'; val; val /= base) {
4330                        if ((digit = val % base) <= '9' - '0') {
4331                                *end-- = '0' + digit;
4332                        } else {
```



```
4333                                *end-- = 'a' + (digit - ('9' - '0') - 1);
4334                        }
4335                }

4337                if (i == 0 && base == 16)
4338                        *end-- = '0';

4340                if (base == 16)
4341                        *end-- = 'x';

4343                if (i == 0 || base == 8 || base == 16)
4344                        *end-- = '0';

4346                if (i < 0 && base == 10)
4347                        *end-- = '-';

4349                regs[rd] = (uintptr_t)end + 1;
4350                mstate->dtms_scratch_ptr += size;
4351                break;
4352        }

4354        case DIF_SUBR_HTONS:
4355        case DIF_SUBR_NTOHS:
4356 #ifdef _BIG_ENDIAN
4357                regs[rd] = (uint16_t)tupregs[0].dttk_value;
4358 #else
4359                regs[rd] = DT_BSWAP_16((uint16_t)tupregs[0].dttk_value);
4360 #endif
4361                break;


4364        case DIF_SUBR_HTONL:
4365        case DIF_SUBR_NTOHL:
4366 #ifdef _BIG_ENDIAN
4367                regs[rd] = (uint32_t)tupregs[0].dttk_value;
4368 #else
4369                regs[rd] = DT_BSWAP_32((uint32_t)tupregs[0].dttk_value);
4370 #endif
4371                break;


4374        case DIF_SUBR_HTONLL:
4375        case DIF_SUBR_NTOHLL:
4376 #ifdef _BIG_ENDIAN
4377                regs[rd] = (uint64_t)tupregs[0].dttk_value;
4378 #else
4379                regs[rd] = DT_BSWAP_64((uint64_t)tupregs[0].dttk_value);
4380 #endif
4381                break;


4384        case DIF_SUBR_DIRNAME:
4385        case DIF_SUBR_BASENAME: {
4386                char *dest = (char *)mstate->dtms_scratch_ptr;
4387                uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
4388                uintptr_t src = tupregs[0].dttk_value;
4389                int i, j, len = dtrace_strlen((char *)src, size);
4390                int lastbase = -1, firstbase = -1, lastdir = -1;
4391                int start, end;

4393                if (!dtrace_canload(src, len + 1, mstate, vstate)) {
4394                        regs[rd] = NULL;
4395                        break;
4396                }

4398                if (!DTRACE_INSCRATCH(mstate, size)) {
```

```
4399                        DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4400                        regs[rd] = NULL;
4401                        break;
4402                }

4404                /*
4405                 * The basename and dirname for a zero-length string is
4406                 * defined to be "."
4407                 */
4408                if (len == 0) {
4409                        len = 1;
4410                        src = (uintptr_t)".";
4411                }

4413                /*
4414                 * Start from the back of the string, moving back toward the
4415                 * front until we see a character that isn't a slash.  That
4416                 * character is the last character in the basename.
4417                 */
4418                for (i = len - 1; i >= 0; i--) {
4419                        if (dtrace_load8(src + i) != '/')
4420                                break;
4421                }

4423                if (i >= 0)
4424                        lastbase = i;

4426                /*
4427                 * Starting from the last character in the basename, move
4428                 * towards the front until we find a slash.  The character
4429                 * that we processed immediately before that is the first
4430                 * character in the basename.
4431                 */
4432                for (; i >= 0; i--) {
4433                        if (dtrace_load8(src + i) == '/')
4434                                break;
4435                }

4437                if (i >= 0)
4438                        firstbase = i + 1;

4440                /*
4441                 * Now keep going until we find a non-slash character.  That
4442                 * character is the last character in the dirname.
4443                 */
4444                for (; i >= 0; i--) {
4445                        if (dtrace_load8(src + i) != '/')
4446                                break;
4447                }

4449                if (i >= 0)
4450                        lastdir = i;

4452                ASSERT(!(lastbase == -1 && firstbase != -1));
4453                ASSERT(!(firstbase == -1 && lastdir != -1));

4455                if (lastbase == -1) {
4456                        /*
4457                         * We didn't find a non-slash character.  We know that
4458                         * the length is non-zero, so the whole string must be
4459                         * slashes.  In either the dirname or the basename
4460                         * case, we return '/'.
4461                         */
4462                        ASSERT(firstbase == -1);
4463                        firstbase = lastbase = lastdir = 0;
4464                }
```



```
4466                if (firstbase == -1) {
4467                        /*
4468                         * The entire string consists only of a basename
4469                         * component.  If we're looking for dirname, we need
4470                         * to change our string to be just "."; if we're
4471                         * looking for a basename, we'll just set the first
4472                         * character of the basename to be 0.
4473                         */
4474                        if (subr == DIF_SUBR_DIRNAME) {
4475                                ASSERT(lastdir == -1);
4476                                src = (uintptr_t)".";
4477                                lastdir = 0;
4478                        } else {
4479                                firstbase = 0;
4480                        }
4481                }

4483                if (subr == DIF_SUBR_DIRNAME) {
4484                        if (lastdir == -1) {
4485                                /*
4486                                 * We know that we have a slash in the name --
4487                                 * or lastdir would be set to 0, above.  And
4488                                 * because lastdir is -1, we know that this
4489                                 * slash must be the first character.  (That
4490                                 * is, the full string must be of the form
4491                                 * "/basename".)  In this case, the last
4492                                 * character of the directory name is 0.
4493                                 */
4494                                lastdir = 0;
4495                        }

4497                        start = 0;
4498                        end = lastdir;
4499                } else {
4500                        ASSERT(subr == DIF_SUBR_BASENAME);
4501                        ASSERT(firstbase != -1 && lastbase != -1);
4502                        start = firstbase;
4503                        end = lastbase;
4504                }

4506                for (i = start, j = 0; i <= end && j < size - 1; i++, j++)
4507                        dest[j] = dtrace_load8(src + i);

4509                dest[j] = '\0';
4510                regs[rd] = (uintptr_t)dest;
4511                mstate->dtms_scratch_ptr += size;
4512                break;
4513        }

4515        case DIF_SUBR_GETF: {
4516                uintptr_t fd = tupregs[0].dttk_value;
4517                uf_info_t *finfo = &curthread->t_procp->p_user.u_finfo;
4518                file_t *fp;

4520                if (!dtrace_priv_proc(state, mstate)) {
4521                        regs[rd] = NULL;
4522                        break;
4523                }

4525                /*
4526                 * This is safe because fi_nfiles only increases, and the
4527                 * fi_list array is not freed when the array size doubles.
4528                 * (See the comment in flist_grow() for details on the
4529                 * management of the u_finfo structure.)
4530                 */
```

```
4531                 fp = fd < finfo->fi_nfiles ? finfo->fi_list[fd].uf_file : NULL;

4533                 mstate->dtms_getf = fp;
4534                 regs[rd] = (uintptr_t)fp;
4535                 break;
4536         }

4538         case DIF_SUBR_CLEANPATH: {
4539                 char *dest = (char *)mstate->dtms_scratch_ptr, c;
4540                 uint64_t size = state->dts_options[DTRACEOPT_STRSIZE];
4541                 uintptr_t src = tupregs[0].dttk_value;
4542                 int i = 0, j = 0;
4543                 zone_t *z;

4545                 if (!dtrace_strcanload(src, size, mstate, vstate)) {
4546                         regs[rd] = NULL;
4547                         break;
4548                 }

4550                 if (!DTRACE_INSCRATCH(mstate, size)) {
4551                         DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4552                         regs[rd] = NULL;
4553                         break;
4554                 }

4556                 /*
4557                  * Move forward, loading each character.
4558                  */
4559                 do {
4560                         c = dtrace_load8(src + i++);
4561 next:
4562                         if (j + 5 >= size)      /* 5 = strlen("/..c\0") */
4563                                 break;

4565                         if (c != '/') {
4566                                 dest[j++] = c;
4567                                 continue;
4568                         }

4570                         c = dtrace_load8(src + i++);

4572                         if (c == '/') {
4573                                 /*
4574                                  * We have two slashes -- we can just advance
4575                                  * to the next character.
4576                                  */
4577                                 goto next;
4578                         }

4580                         if (c != '.') {
4581                                 /*
4582                                  * This is not "." and it's not ".." -- we can
4583                                  * just store the "/" and this character and
4584                                  * drive on.
4585                                  */
4586                                 dest[j++] = '/';
4587                                 dest[j++] = c;
4588                                 continue;
4589                         }

4591                         c = dtrace_load8(src + i++);

4593                         if (c == '/') {
4594                                 /*
4595                                  * This is a "/./" component.  We're not going
4596                                  * to store anything in the destination buffer;
```



```
4597                                  * we're just going to go to the next component.
4598                                  */
4599                                 goto next;
4600                         }

4602                         if (c != '.') {
4603                                 /*
4604                                  * This is not ".." -- we can just store the
4605                                  * "/." and this character and continue
4606                                  * processing.
4607                                  */
4608                                 dest[j++] = '/';
4609                                 dest[j++] = '.';
4610                                 dest[j++] = c;
4611                                 continue;
4612                         }

4614                         c = dtrace_load8(src + i++);

4616                         if (c != '/' && c != '\0') {
4617                                 /*
4618                                  * This is not ".." -- it's "..[mumble]".
4619                                  * We'll store the "/.." and this character
4620                                  * and continue processing.
4621                                  */
4622                                 dest[j++] = '/';
4623                                 dest[j++] = '.';
4624                                 dest[j++] = '.';
4625                                 dest[j++] = c;
4626                                 continue;
4627                         }

4629                         /*
4630                          * This is "/../" or "/..\0".  We need to back up
4631                          * our destination pointer until we find a "/".
4632                          */
4633                         i--;
4634                         while (j != 0 && dest[--j] != '/')
4635                                 continue;

4637                         if (c == '\0')
4638                                 dest[++j] = '/';
4639                 } while (c != '\0');

4641                 dest[j] = '\0';

4643                 if (mstate->dtms_getf != NULL &&
4644                     !(mstate->dtms_access & DTRACE_ACCESS_KERNEL) &&
4645                     (z = state->dts_cred.dcr_cred->cr_zone) != kcred->cr_zone) {
4646                         /*
4647                          * If we've done a getf() as a part of this ECB and we
4648                          * don't have kernel access (and we're not in the global
4649                          * zone), check if the path we cleaned up begins with
4650                          * the zone's root path, and trim it off if so.  Note
4651                          * that this is an output cleanliness issue, not a
4652                          * security issue: knowing one's zone root path does
4653                          * not enable privilege escalation.
4654                          */
4655                         if (strstr(dest, z->zone_rootpath) == dest)
4656                                 dest += strlen(z->zone_rootpath) - 1;
4657                 }

4659                 regs[rd] = (uintptr_t)dest;
4660                 mstate->dtms_scratch_ptr += size;
4661                 break;
4662         }
```

```
4664         case DIF_SUBR_INET_NTOA:
4665         case DIF_SUBR_INET_NTOA6:
4666         case DIF_SUBR_INET_NTOP: {
4667                 size_t size;
4668                 int af, argi, i;
4669                 char *base, *end;

4671                 if (subr == DIF_SUBR_INET_NTOP) {
4672                         af = (int)tupregs[0].dttk_value;
4673                         argi = 1;
4674                 } else {
4675                         af = subr == DIF_SUBR_INET_NTOA ? AF_INET: AF_INET6;
4676                         argi = 0;
4677                 }

4679                 if (af == AF_INET) {
4680                         ipaddr_t ip4;
4681                         uint8_t *ptr8, val;

4683                         /*
4684                          * Safely load the IPv4 address.
4685                          */
4686                         ip4 = dtrace_load32(tupregs[argi].dttk_value);

4688                         /*
4689                          * Check an IPv4 string will fit in scratch.
4690                          */
4691                         size = INET_ADDRSTRLEN;
4692                         if (!DTRACE_INSCRATCH(mstate, size)) {
4693                                 DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4694                                 regs[rd] = NULL;
4695                                 break;
4696                         }
4697                         base = (char *)mstate->dtms_scratch_ptr;
4698                         end = (char *)mstate->dtms_scratch_ptr + size - 1;

4700                         /*
4701                          * Stringify as a dotted decimal quad.
4702                          */
4703                         *end-- = '\0';
4704                         ptr8 = (uint8_t *)&ip4;
4705                         for (i = 3; i >= 0; i--) {
4706                                 val = ptr8[i];

4708                                 if (val == 0) {
4709                                         *end-- = '0';
4710                                 } else {
4711                                         for (; val; val /= 10) {
4712                                                 *end-- = '0' + (val % 10);
4713                                         }
4714                                 }

4716                                 if (i > 0)
4717                                         *end-- = '.';
4718                         }
4719                         ASSERT(end + 1 >= base);

4721                 } else if (af == AF_INET6) {
4722                         struct in6_addr ip6;
4723                         int firstzero, tryzero, numzero, v6end;
4724                         uint16_t val;
4725                         const char digits[] = "0123456789abcdef";

4727                         /*
4728                          * Stringify using RFC 1884 convention 2 - 16 bit
```



```
4729                          * hexadecimal values with a zero-run compression.
4730                          * Lower case hexadecimal digits are used.
4731                          *      eg, fe80::214:4fff:fe0b:76c8.
4732                          * The IPv4 embedded form is returned for inet_ntop,
4733                          * just the IPv4 string is returned for inet_ntoa6.
4734                          */

4736                         /*
4737                          * Safely load the IPv6 address.
4738                          */
4739                         dtrace_bcopy(
4740                             (void *)(uintptr_t)tupregs[argi].dttk_value,
4741                             (void *)(uintptr_t)&ip6, sizeof (struct in6_addr));

4743                         /*
4744                          * Check an IPv6 string will fit in scratch.
4745                          */
4746                         size = INET6_ADDRSTRLEN;
4747                         if (!DTRACE_INSCRATCH(mstate, size)) {
4748                                 DTRACE_CPUFLAG_SET(CPU_DTRACE_NOSCRATCH);
4749                                 regs[rd] = NULL;
4750                                 break;
4751                         }
4752                         base = (char *)mstate->dtms_scratch_ptr;
4753                         end = (char *)mstate->dtms_scratch_ptr + size - 1;
4754                         *end-- = '\0';

4756                         /*
4757                          * Find the longest run of 16 bit zero values
4758                          * for the single allowed zero compression - "::".
4759                          */
4760                         firstzero = -1;
4761                         tryzero = -1;
4762                         numzero = 1;
4763                         for (i = 0; i < sizeof (struct in6_addr); i++) {
4764                                 if (ip6._S6_un._S6_u8[i] == 0 &&
4765                                     tryzero == -1 && i % 2 == 0) {
4766                                         tryzero = i;
4767                                         continue;
4768                                 }

4770                                 if (tryzero != -1 &&
4771                                     (ip6._S6_un._S6_u8[i] != 0 ||
4772                                     i == sizeof (struct in6_addr) - 1)) {

4774                                         if (i - tryzero <= numzero) {
4775                                                 tryzero = -1;
4776                                                 continue;
4777                                         }

4779                                         firstzero = tryzero;
4780                                         numzero = i - i % 2 - tryzero;
4781                                         tryzero = -1;

4783                                         if (ip6._S6_un._S6_u8[i] == 0 &&
4784                                             i == sizeof (struct in6_addr) - 1)
4785                                                 numzero += 2;
4786                                 }
4787                         }
4788                         ASSERT(firstzero + numzero <= sizeof (struct in6_addr));

4790                         /*
4791                          * Check for an IPv4 embedded address.
4792                          */
4793                         v6end = sizeof (struct in6_addr) - 2;
4794                         if (IN6_IS_ADDR_V4MAPPED(&ip6) ||
```

```
4795                            IN6_IS_ADDR_V4COMPAT(&ip6)) {
4796                                for (i = sizeof (struct in6_addr) - 1;
4797                                    i >= DTRACE_V4MAPPED_OFFSET; i--) {
4798                                        ASSERT(end >= base);

4800                                        val = ip6._S6_un._S6_u8[i];

4802                                        if (val == 0) {
4803                                                *end-- = '0';
4804                                        } else {
4805                                                for (; val; val /= 10) {
4806                                                        *end-- = '0' + val % 10;
4807                                                }
4808                                        }

4810                                        if (i > DTRACE_V4MAPPED_OFFSET)
4811                                                *end-- = '.';
4812                                }

4814                                if (subr == DIF_SUBR_INET_NTOA6)
4815                                        goto inetout;

4817                                /*
4818                                 * Set v6end to skip the IPv4 address that
4819                                 * we have already stringified.
4820                                 */
4821                                v6end = 10;
4822                        }

4824                        /*
4825                         * Build the IPv6 string by working through the
4826                         * address in reverse.
4827                         */
4828                        for (i = v6end; i >= 0; i -= 2) {
4829                                ASSERT(end >= base);

4831                                if (i == firstzero + numzero - 2) {
4832                                        *end-- = ':';
4833                                        *end-- = ':';
4834                                        i -= numzero - 2;
4835                                        continue;
4836                                }

4838                                if (i < 14 && i != firstzero - 2)
4839                                        *end-- = ':';

4841                                val = (ip6._S6_un._S6_u8[i] << 8) +
4842                                    ip6._S6_un._S6_u8[i + 1];

4844                                if (val == 0) {
4845                                        *end-- = '0';
4846                                } else {
4847                                        for (; val; val /= 16) {
4848                                                *end-- = digits[val % 16];
4849                                        }
4850                                }
4851                        }
4852                        ASSERT(end + 1 >= base);

4854                } else {
4855                        /*
4856                         * The user didn't use AH_INET or AH_INET6.
4857                         */
4858                        DTRACE_CPUFLAG_SET(CPU_DTRACE_ILLOP);
4859                        regs[rd] = NULL;
4860                        break;
```



```
4861                }

4863 inetout:       regs[rd] = (uintptr_t)end + 1;
4864                mstate->dtms_scratch_ptr += size;
4865                break;
4866        }

4868        }
4869 }
```

*unchanged_portion_omitted_*

```
5925 /*
5926  * If you're looking for the epicenter of DTrace, you just found it.  This
5927  * is the function called by the provider to fire a probe -- from which all
5928  * subsequent probe-context DTrace activity emanates.
5929  */
5930 void
5931 dtrace_probe(dtrace_id_t id, uintptr_t arg0, uintptr_t arg1,
5932     uintptr_t arg2, uintptr_t arg3, uintptr_t arg4)
5933 {
5934         processorid_t cpuid;
5935         dtrace_icookie_t cookie;
5936         dtrace_probe_t *probe;
5937         dtrace_mstate_t mstate;
5938         dtrace_ecb_t *ecb;
5939         dtrace_action_t *act;
5940         intptr_t offs;
5941         size_t size;
5942         int vtime, onintr;
5943         volatile uint16_t *flags;
5944         hrtime_t now;

5946         /*
5947          * Kick out immediately if this CPU is still being born (in which case
5948          * curthread will be set to -1) or the current thread can't allow
5949          * probes in its current context.
5950          */
5951         if (((uintptr_t)curthread & 1) || (curthread->t_flag & T_DONTDTRACE))
5952                 return;

5954         cookie = dtrace_interrupt_disable();
5955         probe = dtrace_probes[id - 1];
5956         cpuid = CPU->cpu_id;
5957         onintr = CPU_ON_INTR(CPU);

5959         if (!onintr && probe->dtpr_predcache != DTRACE_CACHEIDNONE &&
5960             probe->dtpr_predcache == curthread->t_predcache) {
5961                 /*
5962                  * We have hit in the predicate cache; we know that
5963                  * this predicate would evaluate to be false.
5964                  */
5965                 dtrace_interrupt_enable(cookie);
5966                 return;
5967         }

5969         if (panic_quiesce) {
5970                 /*
5971                  * We don't trace anything if we're panicking.
5972                  */
5973                 dtrace_interrupt_enable(cookie);
5974                 return;
5975         }

5977         now = dtrace_gethrtime();
5978         vtime = dtrace_vtime_references != 0;
```

```
5980         if (vtime && curthread->t_dtrace_start)
5981                 curthread->t_dtrace_vtime += now - curthread->t_dtrace_start;

5983         mstate.dtms_difo = NULL;
5984         mstate.dtms_probe = probe;
5985         mstate.dtms_strtok = NULL;
5986         mstate.dtms_arg[0] = arg0;
5987         mstate.dtms_arg[1] = arg1;
5988         mstate.dtms_arg[2] = arg2;
5989         mstate.dtms_arg[3] = arg3;
5990         mstate.dtms_arg[4] = arg4;

5992         flags = (volatile uint16_t *)&cpu_core[cpuid].cpuc_dtrace_flags;

5994         for (ecb = probe->dtpr_ecb; ecb != NULL; ecb = ecb->dte_next) {
5995                 dtrace_predicate_t *pred = ecb->dte_predicate;
5996                 dtrace_state_t *state = ecb->dte_state;
5997                 dtrace_buffer_t *buf = &state->dts_buffer[cpuid];
5998                 dtrace_buffer_t *aggbuf = &state->dts_aggbuffer[cpuid];
5999                 dtrace_vstate_t *vstate = &state->dts_vstate;
6000                 dtrace_provider_t *prov = probe->dtpr_provider;
6001                 uint64_t tracememsize = 0;
6002                 int committed = 0;
6003                 caddr_t tomax;

6005                 /*
6006                  * A little subtlety with the following (seemingly innocuous)
6007                  * declaration of the automatic 'val':  by looking at the
6008                  * code, you might think that it could be declared in the
6009                  * action processing loop, below.  (That is, it's only used in
6010                  * the action processing loop.)  However, it must be declared
6011                  * out of that scope because in the case of DIF expression
6012                  * arguments to aggregating actions, one iteration of the
6013                  * action loop will use the last iteration's value.
6014                  */
6015 #ifdef lint
6016                 uint64_t val = 0;
6017 #else
6018                 uint64_t val;
6019 #endif

6021                 mstate.dtms_present = DTRACE_MSTATE_ARGS | DTRACE_MSTATE_PROBE;
6022                 mstate.dtms_access = DTRACE_ACCESS_ARGS | DTRACE_ACCESS_PROC;
6023                 mstate.dtms_getf = NULL;

6025                 *flags &= ~CPU_DTRACE_ERROR;

6027                 if (prov == dtrace_provider) {
6028                         /*
6029                          * If dtrace itself is the provider of this probe,
6030                          * we're only going to continue processing the ECB if
6031                          * arg0 (the dtrace_state_t) is equal to the ECB's
6032                          * creating state.  (This prevents disjoint consumers
6033                          * from seeing one another's metaprobes.)
6034                          */
6035                         if (arg0 != (uint64_t)(uintptr_t)state)
6036                                 continue;
6037                 }

6039                 if (state->dts_activity != DTRACE_ACTIVITY_ACTIVE) {
6040                         /*
6041                          * We're not currently active.  If our provider isn't
6042                          * the dtrace pseudo provider, we're not interested.
6043                          */
6044                         if (prov != dtrace_provider)
6045                                 continue;
```



```
6047                         /*
6048                          * Now we must further check if we are in the BEGIN
6049                          * probe.  If we are, we will only continue processing
6050                          * if we're still in WARMUP -- if one BEGIN enabling
6051                          * has invoked the exit() action, we don't want to
6052                          * evaluate subsequent BEGIN enablings.
6053                          */
6054                         if (probe->dtpr_id == dtrace_probeid_begin &&
6055                             state->dts_activity != DTRACE_ACTIVITY_WARMUP) {
6056                                 ASSERT(state->dts_activity ==
6057                                     DTRACE_ACTIVITY_DRAINING);
6058                                 continue;
6059                         }
6060                 }

6062                 if (ecb->dte_cond && !dtrace_priv_probe(state, &mstate, ecb))
6063                         continue;

6065                 if (now - state->dts_alive > dtrace_deadman_timeout) {
6066                         /*
6067                          * We seem to be dead.  Unless we (a) have kernel
6068                          * destructive permissions (b) have expicitly enabled
6069                          * destructive actions and (c) destructive actions have
6070                          * not been disabled, we're going to transition into
6071                          * the KILLED state, from which no further processing
6072                          * on this state will be performed.
6073                          */
6074                         if (!dtrace_priv_kernel_destructive(state) ||
6075                             !state->dts_cred.dcr_destructive ||
6076                             dtrace_destructive_disallow) {
6077                                 void *activity = &state->dts_activity;
6078                                 dtrace_activity_t current;

6080                                 do {
6081                                         current = state->dts_activity;
6082                                 } while (dtrace_cas32(activity, current,
6083                                     DTRACE_ACTIVITY_KILLED) != current);

6085                                 continue;
6086                         }
6087                 }

6089                 if ((offs = dtrace_buffer_reserve(buf, ecb->dte_needed,
6090                     ecb->dte_alignment, state, &mstate)) < 0)
6091                         continue;

6093                 tomax = buf->dtb_tomax;
6094                 ASSERT(tomax != NULL);

6096                 if (ecb->dte_size != 0)
6097                         DTRACE_STORE(uint32_t, tomax, offs, ecb->dte_epid);

6099                 mstate.dtms_epid = ecb->dte_epid;
6100                 mstate.dtms_present |= DTRACE_MSTATE_EPID;

6102                 if (state->dts_cred.dcr_visible & DTRACE_CRV_KERNEL)
6103                         mstate.dtms_access |= DTRACE_ACCESS_KERNEL;

6105                 if (pred != NULL) {
6106                         dtrace_difo_t *dp = pred->dtp_difo;
6107                         int rval;

6109                         rval = dtrace_dif_emulate(dp, &mstate, vstate, state);

6111                         if (!(*flags & CPU_DTRACE_ERROR) && !rval) {
```

```
6112                                dtrace_cacheid_t cid = probe->dtpr_predcache;

6114                                if (cid != DTRACE_CACHEIDNONE && !onintr) {
6115                                        /*
6116                                         * Update the predicate cache...
6117                                         */
6118                                        ASSERT(cid == pred->dtp_cacheid);
6119                                        curthread->t_predcache = cid;
6120                                }

6122                                continue;
6123                        }
6124                }

6126                for (act = ecb->dte_action; !(*flags & CPU_DTRACE_ERROR) &&
6127                    act != NULL; act = act->dta_next) {
6128                        size_t valoffs;
6129                        dtrace_difo_t *dp;
6130                        dtrace_recdesc_t *rec = &act->dta_rec;

6132                        size = rec->dtrd_size;
6133                        valoffs = offs + rec->dtrd_offset;

6135                        if (DTRACEACT_ISAGG(act->dta_kind)) {
6136                                uint64_t v = 0xbad;
6137                                dtrace_aggregation_t *agg;

6139                                agg = (dtrace_aggregation_t *)act;

6141                                if ((dp = act->dta_difo) != NULL)
6142                                        v = dtrace_dif_emulate(dp,
6143                                            &mstate, vstate, state);

6145                                if (*flags & CPU_DTRACE_ERROR)
6146                                        continue;

6148                                /*
6149                                 * Note that we always pass the expression
6150                                 * value from the previous iteration of the
6151                                 * action loop.  This value will only be used
6152                                 * if there is an expression argument to the
6153                                 * aggregating action, denoted by the
6154                                 * dtag_hasarg field.
6155                                 */
6156                                dtrace_aggregate(agg, buf,
6157                                    offs, aggbuf, v, val);
6158                                continue;
6159                        }

6161                        switch (act->dta_kind) {
6162                        case DTRACEACT_STOP:
6163                                if (dtrace_priv_proc_destructive(state,
6164                                    &mstate))
6165                                        dtrace_action_stop();
6166                                continue;

6168                        case DTRACEACT_BREAKPOINT:
6169                                if (dtrace_priv_kernel_destructive(state))
6170                                        dtrace_action_breakpoint(ecb);
6171                                continue;

6173                        case DTRACEACT_PANIC:
6174                                if (dtrace_priv_kernel_destructive(state))
6175                                        dtrace_action_panic(ecb);
6176                                continue;
```



```
6178                        case DTRACEACT_STACK:
6179                                if (!dtrace_priv_kernel(state))
6180                                        continue;

6182                                dtrace_getpcstack((pc_t *)(tomax + valoffs),
6183                                    size / sizeof (pc_t), probe->dtpr_aframes,
6184                                    DTRACE_ANCHORED(probe) ? NULL :
6185                                    (uint32_t *)arg0);

6187                                continue;

6189                        case DTRACEACT_JSTACK:
6190                        case DTRACEACT_USTACK:
6191                                if (!dtrace_priv_proc(state, &mstate))
6192                                        continue;

6194                                /*
6195                                 * See comment in DIF_VAR_PID.
6196                                 */
6197                                if (DTRACE_ANCHORED(mstate.dtms_probe) &&
6198                                    CPU_ON_INTR(CPU)) {
6199                                        int depth = DTRACE_USTACK_NFRAMES(
6200                                            rec->dtrd_arg) + 1;

6202                                        dtrace_bzero((void *)(tomax + valoffs),
6203                                            DTRACE_USTACK_STRSIZE(rec->dtrd_arg)
6204                                            + depth * sizeof (uint64_t));

6206                                        continue;
6207                                }

6209                                if (DTRACE_USTACK_STRSIZE(rec->dtrd_arg) != 0 &&
6210                                    curproc->p_dtrace_helpers != NULL) {
6211                                        /*
6212                                         * This is the slow path -- we have
6213                                         * allocated string space, and we're
6214                                         * getting the stack of a process that
6215                                         * has helpers.  Call into a separate
6216                                         * routine to perform this processing.
6217                                         */
6218                                        dtrace_action_ustack(&mstate, state,
6219                                            (uint64_t *)(tomax + valoffs),
6220                                            rec->dtrd_arg);
6221                                        continue;
6222                                }

6224                                /*
6225                                 * Clear the string space, since there's no
6226                                 * helper to do it for us.
6227                                 */
6228                                if (DTRACE_USTACK_STRSIZE(rec->dtrd_arg) != 0) {
6229                                        int depth = DTRACE_USTACK_NFRAMES(
6230                                            rec->dtrd_arg);
6231                                        size_t strsize = DTRACE_USTACK_STRSIZE(
6232                                            rec->dtrd_arg);
6233                                        uint64_t *buf = (uint64_t *)(tomax +
6234                                            valoffs);
6235                                        void *strspace = &buf[depth + 1];

6237                                        dtrace_bzero(strspace,
6238                                            MIN(depth, strsize));
6239                                }

6241                                DTRACE_CPUFLAG_SET(CPU_DTRACE_NOFAULT);
6242                                dtrace_getupcstack((uint64_t *)
6243                                    (tomax + valoffs),
```

```
6244                                    DTRACE_USTACK_NFRAMES(rec->dtrd_arg) + 1);
6245                            DTRACE_CPUFLAG_CLEAR(CPU_DTRACE_NOFAULT);
6246                            continue;

6248                    default:
6249                            break;
6250                    }

6252                    dp = act->dta_difo;
6253                    ASSERT(dp != NULL);

6255                    val = dtrace_dif_emulate(dp, &mstate, vstate, state);

6257                    if (*flags & CPU_DTRACE_ERROR)
6258                            continue;

6260                    switch (act->dta_kind) {
6261                    case DTRACEACT_SPECULATE:
6262                            ASSERT(buf == &state->dts_buffer[cpuid]);
6263                            buf = dtrace_speculation_buffer(state,
6264                                cpuid, val);

6266                            if (buf == NULL) {
6267                                    *flags |= CPU_DTRACE_DROP;
6268                                    continue;
6269                            }

6271                            offs = dtrace_buffer_reserve(buf,
6272                                ecb->dte_needed, ecb->dte_alignment,
6273                                state, NULL);

6275                            if (offs < 0) {
6276                                    *flags |= CPU_DTRACE_DROP;
6277                                    continue;
6278                            }

6280                            tomax = buf->dtb_tomax;
6281                            ASSERT(tomax != NULL);

6283                            if (ecb->dte_size != 0)
6284                                    DTRACE_STORE(uint32_t, tomax, offs,
6285                                        ecb->dte_epid);
6286                            continue;

6288                    case DTRACEACT_CHILL:
6289                            if (dtrace_priv_kernel_destructive(state))
6290                                    dtrace_action_chill(&mstate, val);
6291                            continue;

6293                    case DTRACEACT_RAISE:
6294                            if (dtrace_priv_proc_destructive(state,
6295                                &mstate))
6296                                    dtrace_action_raise(val);
6297                            continue;

6299                    case DTRACEACT_COMMIT:
6300                            ASSERT(!committed);

6302                            /*
6303                             * We need to commit our buffer state.
6304                             */
6305                            if (ecb->dte_size)
6306                                    buf->dtb_offset = offs + ecb->dte_size;
6307                            buf = &state->dts_buffer[cpuid];
6308                            dtrace_speculation_commit(state, cpuid, val);
6309                            committed = 1;
```



```
6310                            continue;

6312                    case DTRACEACT_DISCARD:
6313                            dtrace_speculation_discard(state, cpuid, val);
6314                            continue;

6316                    case DTRACEACT_DIFEXPR:
6317                    case DTRACEACT_LIBACT:
6318                    case DTRACEACT_PRINTF:
6319                    case DTRACEACT_PRINTA:
6320                    case DTRACEACT_SYSTEM:
6321                    case DTRACEACT_FREOPEN:
6322                    case DTRACEACT_TRACEMEM:
6323                            break;

6325                    case DTRACEACT_TRACEMEM_DYNSIZE:
6326                            tracememsize = val;
6327                            break;

6329                    case DTRACEACT_SYM:
6330                    case DTRACEACT_MOD:
6331                            if (!dtrace_priv_kernel(state))
6332                                    continue;
6333                            break;

6335                    case DTRACEACT_USYM:
6336                    case DTRACEACT_UMOD:
6337                    case DTRACEACT_UADDR: {
6338                            struct pid *pid = curthread->t_procp->p_pidp;

6340                            if (!dtrace_priv_proc(state, &mstate))
6341                                    continue;

6343                            DTRACE_STORE(uint64_t, tomax,
6344                                valoffs, (uint64_t)pid->pid_id);
6345                            DTRACE_STORE(uint64_t, tomax,
6346                                valoffs + sizeof (uint64_t), val);

6348                            continue;
6349                    }

6351                    case DTRACEACT_EXIT: {
6352                            /*
6353                             * For the exit action, we are going to attempt
6354                             * to atomically set our activity to be
6355                             * draining.  If this fails (either because
6356                             * another CPU has beat us to the exit action,
6357                             * or because our current activity is something
6358                             * other than ACTIVE or WARMUP), we will
6359                             * continue.  This assures that the exit action
6360                             * can be successfully recorded at most once
6361                             * when we're in the ACTIVE state.  If we're
6362                             * encountering the exit() action while in
6363                             * COOLDOWN, however, we want to honor the new
6364                             * status code.  (We know that we're the only
6365                             * thread in COOLDOWN, so there is no race.)
6366                             */
6367                            void *activity = &state->dts_activity;
6368                            dtrace_activity_t current = state->dts_activity;

6370                            if (current == DTRACE_ACTIVITY_COOLDOWN)
6371                                    break;

6373                            if (current != DTRACE_ACTIVITY_WARMUP)
6374                                    current = DTRACE_ACTIVITY_ACTIVE;
```

```
6376                                    if (dtrace_cas32(activity, current,
6377                                        DTRACE_ACTIVITY_DRAINING) != current) {
6378                                            *flags |= CPU_DTRACE_DROP;
6379                                            continue;
6380                                    }

6382                                    break;
6383                            }

6385                            default:
6386                                    ASSERT(0);
6387                            }

6389                            if (dp->dtdo_rtype.dtdt_flags & DIF_TF_BYREF) {
6390                                    uintptr_t end = valoffs + size;

6392                                    if (tracememsize != 0 &&
6393                                        valoffs + tracememsize < end) {
6394                                            end = valoffs + tracememsize;
6395                                            tracememsize = 0;
6396                                    }

6398                                    if (!dtrace_vcanload((void *)(uintptr_t)val,
6399                                        &dp->dtdo_rtype, &mstate, vstate))
6400                                            continue;

6402                                    /*
6403                                     * If this is a string, we're going to only
6404                                     * load until we find the zero byte -- after
6405                                     * which we'll store zero bytes.
6406                                     */
6407                                    if (dp->dtdo_rtype.dtdt_kind ==
6408                                        DIF_TYPE_STRING) {
6409                                            char c = '\0' + 1;
6410                                            int intuple = act->dta_intuple;
6411                                            size_t s;

6413                                            for (s = 0; s < size; s++) {
6414                                                    if (c != '\0')
6415                                                            c = dtrace_load8(val++);

6417                                                    DTRACE_STORE(uint8_t, tomax,
6418                                                        valoffs++, c);

6420                                                    if (c == '\0' && intuple)
6421                                                            break;
6422                                            }

6424                                            continue;
6425                                    }

6427                                    while (valoffs < end) {
6428                                            DTRACE_STORE(uint8_t, tomax, valoffs++,
6429                                                dtrace_load8(val++));
6430                                    }

6432                                    continue;
6433                            }

6435                            switch (size) {
6436                            case 0:
6437                                    break;

6439                            case sizeof (uint8_t):
6440                                    DTRACE_STORE(uint8_t, tomax, valoffs, val);
6441                                    break;
```



```
6442                            case sizeof (uint16_t):
6443                                    DTRACE_STORE(uint16_t, tomax, valoffs, val);
6444                                    break;
6445                            case sizeof (uint32_t):
6446                                    DTRACE_STORE(uint32_t, tomax, valoffs, val);
6447                                    break;
6448                            case sizeof (uint64_t):
6449                                    DTRACE_STORE(uint64_t, tomax, valoffs, val);
6450                                    break;
6451                            default:
6452                                    /*
6453                                     * Any other size should have been returned by
6454                                     * reference, not by value.
6455                                     */
6456                                    ASSERT(0);
6457                                    break;
6458                            }
6459                    }

6461                    if (*flags & CPU_DTRACE_DROP)
6462                            continue;

6464                    if (*flags & CPU_DTRACE_FAULT) {
6465                            int ndx;
6466                            dtrace_action_t *err;

6468                            buf->dtb_errors++;

6470                            if (probe->dtpr_id == dtrace_probeid_error) {
6471                                    /*
6472                                     * There's nothing we can do -- we had an
6473                                     * error on the error probe.  We bump an
6474                                     * error counter to at least indicate that
6475                                     * this condition happened.
6476                                     */
6477                                    dtrace_error(&state->dts_dblerrors);
6478                                    continue;
6479                            }

6481                            if (vtime) {
6482                                    /*
6483                                     * Before recursing on dtrace_probe(), we
6484                                     * need to explicitly clear out our start
6485                                     * time to prevent it from being accumulated
6486                                     * into t_dtrace_vtime.
6487                                     */
6488                                    curthread->t_dtrace_start = 0;
6489                            }

6491                            /*
6492                             * Iterate over the actions to figure out which action
6493                             * we were processing when we experienced the error.
6494                             * Note that act points _past_ the faulting action; if
6495                             * act is ecb->dte_action, the fault was in the
6496                             * predicate, if it's ecb->dte_action->dta_next it's
6497                             * in action #1, and so on.
6498                             */
6499                            for (err = ecb->dte_action, ndx = 0;
6500                                err != act; err = err->dta_next, ndx++)
6501                                    continue;

6503                            dtrace_probe_error(state, ecb->dte_epid, ndx,
6504                                (mstate.dtms_present & DTRACE_MSTATE_FLTOFFS) ?
6505                                mstate.dtms_fltoffs : -1, DTRACE_FLAGS2FLT(*flags),
6506                                cpu_core[cpuid].cpuc_dtrace_illval);
```

```
6508                        continue;
6509                }

6511                if (!committed)
6512                        buf->dtb_offset = offs + ecb->dte_size;
6513        }

6515        if (vtime)
6516                curthread->t_dtrace_start = dtrace_gethrtime();

6518        dtrace_interrupt_enable(cookie);
6519 }
```

*unchanged_portion_omitted_*

```
8253 /*
8254  * Validate a DTrace DIF object by checking the IR instructions.  The following
8255  * rules are currently enforced by dtrace_difo_validate():
8256  *
8257  * 1. Each instruction must have a valid opcode
8258  * 2. Each register, string, variable, or subroutine reference must be valid
8259  * 3. No instruction can modify register %r0 (must be zero)
8260  * 4. All instruction reserved bits must be set to zero
8261  * 5. The last instruction must be a "ret" instruction
8262  * 6. All branch targets must reference a valid instruction _after_ the branch
8263  */
8264 static int
8265 dtrace_difo_validate(dtrace_difo_t *dp, dtrace_vstate_t *vstate, uint_t nregs,
8266     cred_t *cr)
8267 {
8268        int err = 0, i;
8269        int (*efunc)(uint_t pc, const char *, ...) = dtrace_difo_err;
8270        int kcheckload;
8271        uint_t pc;

8273        kcheckload = cr == NULL ||
8274            (vstate->dtvs_state->dts_cred.dcr_visible & DTRACE_CRV_KERNEL) == 0;

8276        dp->dtdo_destructive = 0;

8278        for (pc = 0; pc < dp->dtdo_len && err == 0; pc++) {
8279                dif_instr_t instr = dp->dtdo_buf[pc];

8281                uint_t r1 = DIF_INSTR_R1(instr);
8282                uint_t r2 = DIF_INSTR_R2(instr);
8283                uint_t rd = DIF_INSTR_RD(instr);
8284                uint_t rs = DIF_INSTR_RS(instr);
8285                uint_t label = DIF_INSTR_LABEL(instr);
8286                uint_t v = DIF_INSTR_VAR(instr);
8287                uint_t subr = DIF_INSTR_SUBR(instr);
8288                uint_t type = DIF_INSTR_TYPE(instr);
8289                uint_t op = DIF_INSTR_OP(instr);

8291                switch (op) {
8292                case DIF_OP_OR:
8293                case DIF_OP_XOR:
8294                case DIF_OP_AND:
8295                case DIF_OP_SLL:
8296                case DIF_OP_SRL:
8297                case DIF_OP_SRA:
8298                case DIF_OP_SUB:
8299                case DIF_OP_ADD:
8300                case DIF_OP_MUL:
8301                case DIF_OP_SDIV:
8302                case DIF_OP_UDIV:
8303                case DIF_OP_SREM:
8304                case DIF_OP_UREM:
```



```
8305                case DIF_OP_COPYS:
8306                        if (r1 >= nregs)
8307                                err += efunc(pc, "invalid register %u\n", r1);
8308                        if (r2 >= nregs)
8309                                err += efunc(pc, "invalid register %u\n", r2);
8310                        if (rd >= nregs)
8311                                err += efunc(pc, "invalid register %u\n", rd);
8312                        if (rd == 0)
8313                                err += efunc(pc, "cannot write to %r0\n");
8314                        break;
8315                case DIF_OP_NOT:
8316                case DIF_OP_MOV:
8317                case DIF_OP_ALLOCS:
8318                        if (r1 >= nregs)
8319                                err += efunc(pc, "invalid register %u\n", r1);
8320                        if (r2 != 0)
8321                                err += efunc(pc, "non-zero reserved bits\n");
8322                        if (rd >= nregs)
8323                                err += efunc(pc, "invalid register %u\n", rd);
8324                        if (rd == 0)
8325                                err += efunc(pc, "cannot write to %r0\n");
8326                        break;
8327                case DIF_OP_LDSB:
8328                case DIF_OP_LDSH:
8329                case DIF_OP_LDSW:
8330                case DIF_OP_LDUB:
8331                case DIF_OP_LDUH:
8332                case DIF_OP_LDUW:
8333                case DIF_OP_LDX:
8334                        if (r1 >= nregs)
8335                                err += efunc(pc, "invalid register %u\n", r1);
8336                        if (r2 != 0)
8337                                err += efunc(pc, "non-zero reserved bits\n");
8338                        if (rd >= nregs)
8339                                err += efunc(pc, "invalid register %u\n", rd);
8340                        if (rd == 0)
8341                                err += efunc(pc, "cannot write to %r0\n");
8342                        if (kcheckload)
8343                                dp->dtdo_buf[pc] = DIF_INSTR_LOAD(op +
8344                                    DIF_OP_RLDSB - DIF_OP_LDSB, r1, rd);
8345                        break;
8346                case DIF_OP_RLDSB:
8347                case DIF_OP_RLDSH:
8348                case DIF_OP_RLDSW:
8349                case DIF_OP_RLDUB:
8350                case DIF_OP_RLDUH:
8351                case DIF_OP_RLDUW:
8352                case DIF_OP_RLDX:
8353                        if (r1 >= nregs)
8354                                err += efunc(pc, "invalid register %u\n", r1);
8355                        if (r2 != 0)
8356                                err += efunc(pc, "non-zero reserved bits\n");
8357                        if (rd >= nregs)
8358                                err += efunc(pc, "invalid register %u\n", rd);
8359                        if (rd == 0)
8360                                err += efunc(pc, "cannot write to %r0\n");
8361                        break;
8362                case DIF_OP_ULDSB:
8363                case DIF_OP_ULDSH:
8364                case DIF_OP_ULDSW:
8365                case DIF_OP_ULDUB:
8366                case DIF_OP_ULDUH:
8367                case DIF_OP_ULDUW:
8368                case DIF_OP_ULDX:
8369                        if (r1 >= nregs)
8370                                err += efunc(pc, "invalid register %u\n", r1);
```

```
8371                        if (r2 != 0)
8372                                err += efunc(pc, "non-zero reserved bits\n");
8373                        if (rd >= nregs)
8374                                err += efunc(pc, "invalid register %u\n", rd);
8375                        if (rd == 0)
8376                                err += efunc(pc, "cannot write to %r0\n");
8377                        break;
8378                case DIF_OP_STB:
8379                case DIF_OP_STH:
8380                case DIF_OP_STW:
8381                case DIF_OP_STX:
8382                        if (r1 >= nregs)
8383                                err += efunc(pc, "invalid register %u\n", r1);
8384                        if (r2 != 0)
8385                                err += efunc(pc, "non-zero reserved bits\n");
8386                        if (rd >= nregs)
8387                                err += efunc(pc, "invalid register %u\n", rd);
8388                        if (rd == 0)
8389                                err += efunc(pc, "cannot write to 0 address\n");
8390                        break;
8391                case DIF_OP_CMP:
8392                case DIF_OP_SCMP:
8393                        if (r1 >= nregs)
8394                                err += efunc(pc, "invalid register %u\n", r1);
8395                        if (r2 >= nregs)
8396                                err += efunc(pc, "invalid register %u\n", r2);
8397                        if (rd != 0)
8398                                err += efunc(pc, "non-zero reserved bits\n");
8399                        break;
8400                case DIF_OP_TST:
8401                        if (r1 >= nregs)
8402                                err += efunc(pc, "invalid register %u\n", r1);
8403                        if (r2 != 0 || rd != 0)
8404                                err += efunc(pc, "non-zero reserved bits\n");
8405                        break;
8406                case DIF_OP_BA:
8407                case DIF_OP_BE:
8408                case DIF_OP_BNE:
8409                case DIF_OP_BG:
8410                case DIF_OP_BGU:
8411                case DIF_OP_BGE:
8412                case DIF_OP_BGEU:
8413                case DIF_OP_BL:
8414                case DIF_OP_BLU:
8415                case DIF_OP_BLE:
8416                case DIF_OP_BLEU:
8417                        if (label >= dp->dtdo_len) {
8418                                err += efunc(pc, "invalid branch target %u\n",
8419                                    label);
8420                        }
8421                        if (label <= pc) {
8422                                err += efunc(pc, "backward branch to %u\n",
8423                                    label);
8424                        }
8425                        break;
8426                case DIF_OP_RET:
8427                        if (r1 != 0 || r2 != 0)
8428                                err += efunc(pc, "non-zero reserved bits\n");
8429                        if (rd >= nregs)
8430                                err += efunc(pc, "invalid register %u\n", rd);
8431                        break;
8432                case DIF_OP_NOP:
8433                case DIF_OP_POPTS:
8434                case DIF_OP_FLUSHTS:
8435                        if (r1 != 0 || r2 != 0 || rd != 0)
8436                                err += efunc(pc, "non-zero reserved bits\n");
```



```
8437                        break;
8438                case DIF_OP_SETX:
8439                        if (DIF_INSTR_INTEGER(instr) >= dp->dtdo_intlen) {
8440                                err += efunc(pc, "invalid integer ref %u\n",
8441                                    DIF_INSTR_INTEGER(instr));
8442                        }
8443                        if (rd >= nregs)
8444                                err += efunc(pc, "invalid register %u\n", rd);
8445                        if (rd == 0)
8446                                err += efunc(pc, "cannot write to %r0\n");
8447                        break;
8448                case DIF_OP_SETS:
8449                        if (DIF_INSTR_STRING(instr) >= dp->dtdo_strlen) {
8450                                err += efunc(pc, "invalid string ref %u\n",
8451                                    DIF_INSTR_STRING(instr));
8452                        }
8453                        if (rd >= nregs)
8454                                err += efunc(pc, "invalid register %u\n", rd);
8455                        if (rd == 0)
8456                                err += efunc(pc, "cannot write to %r0\n");
8457                        break;
8458                case DIF_OP_LDGA:
8459                case DIF_OP_LDTA:
8460                        if (r1 > DIF_VAR_ARRAY_MAX)
8461                                err += efunc(pc, "invalid array %u\n", r1);
8462                        if (r2 >= nregs)
8463                                err += efunc(pc, "invalid register %u\n", r2);
8464                        if (rd >= nregs)
8465                                err += efunc(pc, "invalid register %u\n", rd);
8466                        if (rd == 0)
8467                                err += efunc(pc, "cannot write to %r0\n");
8468                        break;
8469                case DIF_OP_LDGS:
8470                case DIF_OP_LDTS:
8471                case DIF_OP_LDLS:
8472                case DIF_OP_LDGAA:
8473                case DIF_OP_LDTAA:
8474                        if (v < DIF_VAR_OTHER_MIN || v > DIF_VAR_OTHER_MAX)
8475                                err += efunc(pc, "invalid variable %u\n", v);
8476                        if (rd >= nregs)
8477                                err += efunc(pc, "invalid register %u\n", rd);
8478                        if (rd == 0)
8479                                err += efunc(pc, "cannot write to %r0\n");
8480                        break;
8481                case DIF_OP_STGS:
8482                case DIF_OP_STTS:
8483                case DIF_OP_STLS:
8484                case DIF_OP_STGAA:
8485                case DIF_OP_STTAA:
8486                        if (v < DIF_VAR_OTHER_UBASE || v > DIF_VAR_OTHER_MAX)
8487                                err += efunc(pc, "invalid variable %u\n", v);
8488                        if (rs >= nregs)
8489                                err += efunc(pc, "invalid register %u\n", rd);
8490                        break;
8491                case DIF_OP_CALL:
8492                        if (subr > DIF_SUBR_MAX)
8493                                err += efunc(pc, "invalid subr %u\n", subr);
8494                        if (rd >= nregs)
8495                                err += efunc(pc, "invalid register %u\n", rd);
8496                        if (rd == 0)
8497                                err += efunc(pc, "cannot write to %r0\n");

8499                        if (subr == DIF_SUBR_COPYOUT ||
8500                            subr == DIF_SUBR_COPYOUTSTR) {
8501                                dp->dtdo_destructive = 1;
8502                        }
```

```
8504                        if (subr == DIF_SUBR_GETF) {
8505                                /*
8506                                 * If we have a getf() we need to record that
8507                                 * in our state.  Note that our state can be
8508                                 * NULL if this is a helper -- but in that
8509                                 * case, the call to getf() is itself illegal,
8510                                 * and will be caught (slightly later) when
8511                                 * the helper is validated.
8512                                 */
8513                                if (vstate->dtvs_state != NULL)
8514                                        vstate->dtvs_state->dts_getf++;
8515                        }

8517                        break;
8518                case DIF_OP_PUSHTR:
8519                        if (type != DIF_TYPE_STRING && type != DIF_TYPE_CTF)
8520                                err += efunc(pc, "invalid ref type %u\n", type);
8521                        if (r2 >= nregs)
8522                                err += efunc(pc, "invalid register %u\n", r2);
8523                        if (rs >= nregs)
8524                                err += efunc(pc, "invalid register %u\n", rs);
8525                        break;
8526                case DIF_OP_PUSHTV:
8527                        if (type != DIF_TYPE_CTF)
8528                                err += efunc(pc, "invalid val type %u\n", type);
8529                        if (r2 >= nregs)
8530                                err += efunc(pc, "invalid register %u\n", r2);
8531                        if (rs >= nregs)
8532                                err += efunc(pc, "invalid register %u\n", rs);
8533                        break;
8534                default:
8535                        err += efunc(pc, "invalid opcode %u\n",
8536                            DIF_INSTR_OP(instr));
8537                }
8538        }

8540        if (dp->dtdo_len != 0 &&
8541            DIF_INSTR_OP(dp->dtdo_buf[dp->dtdo_len - 1]) != DIF_OP_RET) {
8542                err += efunc(dp->dtdo_len - 1,
8543                    "expected 'ret' as last DIF instruction\n");
8544        }

8546        if (!(dp->dtdo_rtype.dtdt_flags & DIF_TF_BYREF)) {
8547                /*
8548                 * If we're not returning by reference, the size must be either
8549                 * 0 or the size of one of the base types.
8550                 */
8551                switch (dp->dtdo_rtype.dtdt_size) {
8552                case 0:
8553                case sizeof (uint8_t):
8554                case sizeof (uint16_t):
8555                case sizeof (uint32_t):
8556                case sizeof (uint64_t):
8557                        break;

8559                default:
8560                        err += efunc(dp->dtdo_len - 1, "bad return size\n");
8561                }
8562        }

8564        for (i = 0; i < dp->dtdo_varlen && err == 0; i++) {
8565                dtrace_difv_t *v = &dp->dtdo_vartab[i], *existing = NULL;
8566                dtrace_diftype_t *vt, *et;
8567                uint_t id, ndx;
```



```
8569                if (v->dtdv_scope != DIFV_SCOPE_GLOBAL &&
8570                    v->dtdv_scope != DIFV_SCOPE_THREAD &&
8571                    v->dtdv_scope != DIFV_SCOPE_LOCAL) {
8572                        err += efunc(i, "unrecognized variable scope %d\n",
8573                            v->dtdv_scope);
8574                        break;
8575                }

8577                if (v->dtdv_kind != DIFV_KIND_ARRAY &&
8578                    v->dtdv_kind != DIFV_KIND_SCALAR) {
8579                        err += efunc(i, "unrecognized variable type %d\n",
8580                            v->dtdv_kind);
8581                        break;
8582                }

8584                if ((id = v->dtdv_id) > DIF_VARIABLE_MAX) {
8585                        err += efunc(i, "%d exceeds variable id limit\n", id);
8586                        break;
8587                }

8589                if (id < DIF_VAR_OTHER_UBASE)
8590                        continue;

8592                /*
8593                 * For user-defined variables, we need to check that this
8594                 * definition is identical to any previous definition that we
8595                 * encountered.
8596                 */
8597                ndx = id - DIF_VAR_OTHER_UBASE;

8599                switch (v->dtdv_scope) {
8600                case DIFV_SCOPE_GLOBAL:
8601                        if (ndx < vstate->dtvs_nglobals) {
8602                                dtrace_statvar_t *svar;

8604                                if ((svar = vstate->dtvs_globals[ndx]) != NULL)
8605                                        existing = &svar->dtsv_var;
8606                        }

8608                        break;

8610                case DIFV_SCOPE_THREAD:
8611                        if (ndx < vstate->dtvs_ntlocals)
8612                                existing = &vstate->dtvs_tlocals[ndx];
8613                        break;

8615                case DIFV_SCOPE_LOCAL:
8616                        if (ndx < vstate->dtvs_nlocals) {
8617                                dtrace_statvar_t *svar;

8619                                if ((svar = vstate->dtvs_locals[ndx]) != NULL)
8620                                        existing = &svar->dtsv_var;
8621                        }

8623                        break;
8624                }

8626                vt = &v->dtdv_type;

8628                if (vt->dtdt_flags & DIF_TF_BYREF) {
8629                        if (vt->dtdt_size == 0) {
8630                                err += efunc(i, "zero-sized variable\n");
8631                                break;
8632                        }

8634                        if (v->dtdv_scope == DIFV_SCOPE_GLOBAL &&
```

```
8635                            vt->dtdt_size > dtrace_global_maxsize) {
8636                                err += efunc(i, "oversized by-ref global\n");
8637                                break;
8638                        }
8639                }

8641                if (existing == NULL || existing->dtdv_id == 0)
8642                        continue;

8644                ASSERT(existing->dtdv_id == v->dtdv_id);
8645                ASSERT(existing->dtdv_scope == v->dtdv_scope);

8647                if (existing->dtdv_kind != v->dtdv_kind)
8648                        err += efunc(i, "%d changed variable kind\n", id);

8650                et = &existing->dtdv_type;

8652                if (vt->dtdt_flags != et->dtdt_flags) {
8653                        err += efunc(i, "%d changed variable type flags\n", id);
8654                        break;
8655                }

8657                if (vt->dtdt_size != 0 && vt->dtdt_size != et->dtdt_size) {
8658                        err += efunc(i, "%d changed variable type size\n", id);
8659                        break;
8660                }
8661        }

8663        return (err);
8664 }
```

*unchanged_portion_omitted_*

```
12983 static int
12984 dtrace_state_go(dtrace_state_t *state, processorid_t *cpu)
12985 {
12986         dtrace_optval_t *opt = state->dts_options, sz, nspec;
12987         dtrace_speculation_t *spec;
12988         dtrace_buffer_t *buf;
12989         cyc_handler_t hdlr;
12990         cyc_time_t when;
12991         int rval = 0, i, bufsize = NCPU * sizeof (dtrace_buffer_t);
12992         dtrace_icookie_t cookie;

12994         mutex_enter(&cpu_lock);
12995         mutex_enter(&dtrace_lock);

12997         if (state->dts_activity != DTRACE_ACTIVITY_INACTIVE) {
12998                 rval = EBUSY;
12999                 goto out;
13000         }

13002         /*
13003          * Before we can perform any checks, we must prime all of the
13004          * retained enablings that correspond to this state.
13005          */
13006         dtrace_enabling_prime(state);

13008         if (state->dts_destructive && !state->dts_cred.dcr_destructive) {
13009                 rval = EACCES;
13010                 goto out;
13011         }

13013         dtrace_state_prereserve(state);

13015         /*
13016          * Now we want to do is try to allocate our speculations.
```



```
13017          * We do not automatically resize the number of speculations; if
13018          * this fails, we will fail the operation.
13019          */
13020         nspec = opt[DTRACEOPT_NSPEC];
13021         ASSERT(nspec != DTRACEOPT_UNSET);

13023         if (nspec > INT_MAX) {
13024                 rval = ENOMEM;
13025                 goto out;
13026         }

13028         spec = kmem_zalloc(nspec * sizeof (dtrace_speculation_t),
13029             KM_NOSLEEP | KM_NORMALPRI);

13031         if (spec == NULL) {
13032                 rval = ENOMEM;
13033                 goto out;
13034         }

13036         state->dts_speculations = spec;
13037         state->dts_nspeculations = (int)nspec;

13039         for (i = 0; i < nspec; i++) {
13040                 if ((buf = kmem_zalloc(bufsize,
13041                     KM_NOSLEEP | KM_NORMALPRI)) == NULL) {
13042                         rval = ENOMEM;
13043                         goto err;
13044                 }

13046                 spec[i].dtsp_buffer = buf;
13047         }

13049         if (opt[DTRACEOPT_GRABANON] != DTRACEOPT_UNSET) {
13050                 if (dtrace_anon.dta_state == NULL) {
13051                         rval = ENOENT;
13052                         goto out;
13053                 }

13055                 if (state->dts_necbs != 0) {
13056                         rval = EALREADY;
13057                         goto out;
13058                 }

13060                 state->dts_anon = dtrace_anon_grab();
13061                 ASSERT(state->dts_anon != NULL);
13062                 state = state->dts_anon;

13064                 /*
13065                  * We want "grabanon" to be set in the grabbed state, so we'll
13066                  * copy that option value from the grabbing state into the
13067                  * grabbed state.
13068                  */
13069                 state->dts_options[DTRACEOPT_GRABANON] =
13070                     opt[DTRACEOPT_GRABANON];

13072                 *cpu = dtrace_anon.dta_beganon;

13074                 /*
13075                  * If the anonymous state is active (as it almost certainly
13076                  * is if the anonymous enabling ultimately matched anything),
13077                  * we don't allow any further option processing -- but we
13078                  * don't return failure.
13079                  */
13080                 if (state->dts_activity != DTRACE_ACTIVITY_INACTIVE)
13081                         goto out;
13082         }
```

```
13084         if (opt[DTRACEOPT_AGGSIZE] != DTRACEOPT_UNSET &&
13085             opt[DTRACEOPT_AGGSIZE] != 0) {
13086                 if (state->dts_aggregations == NULL) {
13087                         /*
13088                          * We're not going to create an aggregation buffer
13089                          * because we don't have any ECBs that contain
13090                          * aggregations -- set this option to 0.
13091                          */
13092                         opt[DTRACEOPT_AGGSIZE] = 0;
13093                 } else {
13094                         /*
13095                          * If we have an aggregation buffer, we must also have
13096                          * a buffer to use as scratch.
13097                          */
13098                         if (opt[DTRACEOPT_BUFSIZE] == DTRACEOPT_UNSET ||
13099                             opt[DTRACEOPT_BUFSIZE] < state->dts_needed) {
13100                                 opt[DTRACEOPT_BUFSIZE] = state->dts_needed;
13101                         }
13102                 }
13103         }

13105         if (opt[DTRACEOPT_SPECSIZE] != DTRACEOPT_UNSET &&
13106             opt[DTRACEOPT_SPECSIZE] != 0) {
13107                 if (!state->dts_speculates) {
13108                         /*
13109                          * We're not going to create speculation buffers
13110                          * because we don't have any ECBs that actually
13111                          * speculate -- set the speculation size to 0.
13112                          */
13113                         opt[DTRACEOPT_SPECSIZE] = 0;
13114                 }
13115         }

13117         /*
13118          * The bare minimum size for any buffer that we're actually going to
13119          * do anything to is sizeof (uint64_t).
13120          */
13121         sz = sizeof (uint64_t);

13123         if ((state->dts_needed != 0 && opt[DTRACEOPT_BUFSIZE] < sz) ||
13124             (state->dts_speculates && opt[DTRACEOPT_SPECSIZE] < sz) ||
13125             (state->dts_aggregations != NULL && opt[DTRACEOPT_AGGSIZE] < sz)) {
13126                 /*
13127                  * A buffer size has been explicitly set to 0 (or to a size
13128                  * that will be adjusted to 0) and we need the space -- we
13129                  * need to return failure.  We return ENOSPC to differentiate
13130                  * it from failing to allocate a buffer due to failure to meet
13131                  * the reserve (for which we return E2BIG).
13132                  */
13133                 rval = ENOSPC;
13134                 goto out;
13135         }

13137         if ((rval = dtrace_state_buffers(state)) != 0)
13138                 goto err;

13140         if ((sz = opt[DTRACEOPT_DYNVARSIZE]) == DTRACEOPT_UNSET)
13141                 sz = dtrace_dstate_defsize;

13143         do {
13144                 rval = dtrace_dstate_init(&state->dts_vstate.dtvs_dynvars, sz);

13146                 if (rval == 0)
13147                         break;
```



```
13149                 if (opt[DTRACEOPT_BUFRESIZE] == DTRACEOPT_BUFRESIZE_MANUAL)
13150                         goto err;
13151         } while (sz >>= 1);

13153         opt[DTRACEOPT_DYNVARSIZE] = sz;

13155         if (rval != 0)
13156                 goto err;

13158         if (opt[DTRACEOPT_STATUSRATE] > dtrace_statusrate_max)
13159                 opt[DTRACEOPT_STATUSRATE] = dtrace_statusrate_max;

13161         if (opt[DTRACEOPT_CLEANRATE] == 0)
13162                 opt[DTRACEOPT_CLEANRATE] = dtrace_cleanrate_max;

13164         if (opt[DTRACEOPT_CLEANRATE] < dtrace_cleanrate_min)
13165                 opt[DTRACEOPT_CLEANRATE] = dtrace_cleanrate_min;

13167         if (opt[DTRACEOPT_CLEANRATE] > dtrace_cleanrate_max)
13168                 opt[DTRACEOPT_CLEANRATE] = dtrace_cleanrate_max;

13170         hdlr.cyh_func = (cyc_func_t)dtrace_state_clean;
13171         hdlr.cyh_arg = state;
13172         hdlr.cyh_level = CY_LOW_LEVEL;

13174         when.cyt_when = 0;
13175         when.cyt_interval = opt[DTRACEOPT_CLEANRATE];

13177         state->dts_cleaner = cyclic_add(&hdlr, &when);

13179         hdlr.cyh_func = (cyc_func_t)dtrace_state_deadman;
13180         hdlr.cyh_arg = state;
13181         hdlr.cyh_level = CY_LOW_LEVEL;

13183         when.cyt_when = 0;
13184         when.cyt_interval = dtrace_deadman_interval;

13186         state->dts_alive = state->dts_laststatus = dtrace_gethrtime();
13187         state->dts_deadman = cyclic_add(&hdlr, &when);

13189         state->dts_activity = DTRACE_ACTIVITY_WARMUP;

13191         if (state->dts_getf != 0 &&
13192             !(state->dts_cred.dcr_visible & DTRACE_CRV_KERNEL)) {
13193                 /*
13194                  * We don't have kernel privs but we have at least one call
13195                  * to getf(); we need to bump our zone's count, and (if
13196                  * this is the first enabling to have an unprivileged call
13197                  * to getf()) we need to hook into closef().
13198                  */
13199                 state->dts_cred.dcr_cred->cr_zone->zone_dtrace_getf++;

13201                 if (dtrace_getf++ == 0) {
13202                         ASSERT(dtrace_closef == NULL);
13203                         dtrace_closef = dtrace_getf_barrier;
13204                 }
13205         }

13207         /*
13208          * Now it's time to actually fire the BEGIN probe.  We need to disable
13209          * interrupts here both to record the CPU on which we fired the BEGIN
13210          * probe (the data from this CPU will be processed first at user
13211          * level) and to manually activate the buffer for this CPU.
13212          */
13213         cookie = dtrace_interrupt_disable();
13214         *cpu = CPU->cpu_id;
```

```
13215         ASSERT(state->dts_buffer[*cpu].dtb_flags & DTRACEBUF_INACTIVE);
13216         state->dts_buffer[*cpu].dtb_flags &= ~DTRACEBUF_INACTIVE;

13218         dtrace_probe(dtrace_probeid_begin,
13219             (uint64_t)(uintptr_t)state, 0, 0, 0, 0);
13220         dtrace_interrupt_enable(cookie);
13221         /*
13222          * We may have had an exit action from a BEGIN probe; only change our
13223          * state to ACTIVE if we're still in WARMUP.
13224          */
13225         ASSERT(state->dts_activity == DTRACE_ACTIVITY_WARMUP ||
13226             state->dts_activity == DTRACE_ACTIVITY_DRAINING);

13228         if (state->dts_activity == DTRACE_ACTIVITY_WARMUP)
13229                 state->dts_activity = DTRACE_ACTIVITY_ACTIVE;

13231         /*
13232          * Regardless of whether or not now we're in ACTIVE or DRAINING, we
13233          * want each CPU to transition its principal buffer out of the
13234          * INACTIVE state.  Doing this assures that no CPU will suddenly begin
13235          * processing an ECB halfway down a probe's ECB chain; all CPUs will
13236          * atomically transition from processing none of a state's ECBs to
13237          * processing all of them.
13238          */
13239         dtrace_xcall(DTRACE_CPUALL,
13240             (dtrace_xcall_t)dtrace_buffer_activate, state);
13241         goto out;

13243 err:
13244         dtrace_buffer_free(state->dts_buffer);
13245         dtrace_buffer_free(state->dts_aggbuffer);

13247         if ((nspec = state->dts_nspeculations) == 0) {
13248                 ASSERT(state->dts_speculations == NULL);
13249                 goto out;
13250         }

13252         spec = state->dts_speculations;
13253         ASSERT(spec != NULL);

13255         for (i = 0; i < state->dts_nspeculations; i++) {
13256                 if ((buf = spec[i].dtsp_buffer) == NULL)
13257                         break;

13259                 dtrace_buffer_free(buf);
13260                 kmem_free(buf, bufsize);
13261         }

13263         kmem_free(spec, nspec * sizeof (dtrace_speculation_t));
13264         state->dts_nspeculations = 0;
13265         state->dts_speculations = NULL;

13267 out:
13268         mutex_exit(&dtrace_lock);
13269         mutex_exit(&cpu_lock);

13271         return (rval);
13272 }

13274 static int
13275 dtrace_state_stop(dtrace_state_t *state, processorid_t *cpu)
13276 {
13277         dtrace_icookie_t cookie;

13279         ASSERT(MUTEX_HELD(&dtrace_lock));
```



```
13281         if (state->dts_activity != DTRACE_ACTIVITY_ACTIVE &&
13282             state->dts_activity != DTRACE_ACTIVITY_DRAINING)
13283                 return (EINVAL);

13285         /*
13286          * We'll set the activity to DTRACE_ACTIVITY_DRAINING, and issue a sync
13287          * to be sure that every CPU has seen it.  See below for the details
13288          * on why this is done.
13289          */
13290         state->dts_activity = DTRACE_ACTIVITY_DRAINING;
13291         dtrace_sync();

13293         /*
13294          * By this point, it is impossible for any CPU to be still processing
13295          * with DTRACE_ACTIVITY_ACTIVE.  We can thus set our activity to
13296          * DTRACE_ACTIVITY_COOLDOWN and know that we're not racing with any
13297          * other CPU in dtrace_buffer_reserve().  This allows dtrace_probe()
13298          * and callees to know that the activity is DTRACE_ACTIVITY_COOLDOWN
13299          * iff we're in the END probe.
13300          */
13301         state->dts_activity = DTRACE_ACTIVITY_COOLDOWN;
13302         dtrace_sync();
13303         ASSERT(state->dts_activity == DTRACE_ACTIVITY_COOLDOWN);

13305         /*
13306          * Finally, we can release the reserve and call the END probe.  We
13307          * disable interrupts across calling the END probe to allow us to
13308          * return the CPU on which we actually called the END probe.  This
13309          * allows user-land to be sure that this CPU's principal buffer is
13310          * processed last.
13311          */
13312         state->dts_reserve = 0;

13314         cookie = dtrace_interrupt_disable();
13315         *cpu = CPU->cpu_id;
13316         dtrace_probe(dtrace_probeid_end,
13317             (uint64_t)(uintptr_t)state, 0, 0, 0, 0);
13318         dtrace_interrupt_enable(cookie);

13320         state->dts_activity = DTRACE_ACTIVITY_STOPPED;
13321         dtrace_sync();

13323         if (state->dts_getf != 0 &&
13324             !(state->dts_cred.dcr_visible & DTRACE_CRV_KERNEL)) {
13325                 /*
13326                  * We don't have kernel privs but we have at least one call
13327                  * to getf(); we need to lower our zone's count, and (if
13328                  * this is the last enabling to have an unprivileged call
13329                  * to getf()) we need to clear the closef() hook.
13330                  */
13331                 ASSERT(state->dts_cred.dcr_cred->cr_zone->zone_dtrace_getf > 0);
13332                 ASSERT(dtrace_closef == dtrace_getf_barrier);
13333                 ASSERT(dtrace_getf > 0);

13335                 state->dts_cred.dcr_cred->cr_zone->zone_dtrace_getf--;

13337                 if (--dtrace_getf == 0)
13338                         dtrace_closef = NULL;
13339         }

13341         return (0);
13342 }
_____unchanged_portion_omitted_

14901 static void
14902 dtrace_getf_barrier()
```

```
14903 {
14904         /*
14905          * When we have unprivileged (that is, non-DTRACE_CRV_KERNEL) enablings
14906          * that contain calls to getf(), this routine will be called on every
14907          * closef() before either the underlying vnode is released or the
14908          * file_t itself is freed.  By the time we are here, it is essential
14909          * that the file_t can no longer be accessed from a call to getf()
14910          * in probe context -- that assures that a dtrace_sync() can be used
14911          * to clear out any enablings referring to the old structures.
14912          */
14913         if (curthread->t_procp->p_zone->zone_dtrace_getf != 0 ||
14914             kcred->cr_zone->zone_dtrace_getf != 0)
14915                 dtrace_sync();
14916 }

14918 /*
14919  * DTrace Driver Cookbook Functions
14920  */
14921 /*ARGSUSED*/
14922 static int
14923 dtrace_attach(dev_info_t *devi, ddi_attach_cmd_t cmd)
14924 {
14925         dtrace_provider_id_t id;
14926         dtrace_state_t *state = NULL;
14927         dtrace_enabling_t *enab;

14929         mutex_enter(&cpu_lock);
14930         mutex_enter(&dtrace_provider_lock);
14931         mutex_enter(&dtrace_lock);

14933         if (ddi_soft_state_init(&dtrace_softstate,
14934             sizeof (dtrace_state_t), 0) != 0) {
14935                 cmn_err(CE_NOTE, "/dev/dtrace failed to initialize soft state");
14936                 mutex_exit(&cpu_lock);
14937                 mutex_exit(&dtrace_provider_lock);
14938                 mutex_exit(&dtrace_lock);
14939                 return (DDI_FAILURE);
14940         }

14942         if (ddi_create_minor_node(devi, DTRACEMNR_DTRACE, S_IFCHR,
14943             DTRACEMNRN_DTRACE, DDI_PSEUDO, NULL) == DDI_FAILURE ||
14944             ddi_create_minor_node(devi, DTRACEMNR_HELPER, S_IFCHR,
14945             DTRACEMNRN_HELPER, DDI_PSEUDO, NULL) == DDI_FAILURE) {
14946                 cmn_err(CE_NOTE, "/dev/dtrace couldn't create minor nodes");
14947                 ddi_remove_minor_node(devi, NULL);
14948                 ddi_soft_state_fini(&dtrace_softstate);
14949                 mutex_exit(&cpu_lock);
14950                 mutex_exit(&dtrace_provider_lock);
14951                 mutex_exit(&dtrace_lock);
14952                 return (DDI_FAILURE);
14953         }

14955         ddi_report_dev(devi);
14956         dtrace_devi = devi;

14958         dtrace_modload = dtrace_module_loaded;
14959         dtrace_modunload = dtrace_module_unloaded;
14960         dtrace_cpu_init = dtrace_cpu_setup_initial;
14961         dtrace_helpers_cleanup = dtrace_helpers_destroy;
14962         dtrace_helpers_fork = dtrace_helpers_duplicate;
14963         dtrace_cpustart_init = dtrace_suspend;
14964         dtrace_cpustart_fini = dtrace_resume;
14965         dtrace_debugger_init = dtrace_suspend;
14966         dtrace_debugger_fini = dtrace_resume;

14968         register_cpu_setup_func((cpu_setup_func_t *)dtrace_cpu_setup, NULL);
```



```
14970         ASSERT(MUTEX_HELD(&cpu_lock));

14972         dtrace_arena = vmem_create("dtrace", (void *)1, UINT32_MAX, 1,
14973             NULL, NULL, NULL, 0, VM_SLEEP | VMC_IDENTIFIER);
14974         dtrace_minor = vmem_create("dtrace_minor", (void *)DTRACEMNRN_CLONE,
14975             UINT32_MAX - DTRACEMNRN_CLONE, 1, NULL, NULL, NULL, 0,
14976             VM_SLEEP | VMC_IDENTIFIER);
14977         dtrace_taskq = taskq_create("dtrace_taskq", 1, maxclsyspri,
14978             1, INT_MAX, 0);

14980         dtrace_state_cache = kmem_cache_create("dtrace_state_cache",
14981             sizeof (dtrace_dstate_percpu_t) * NCPU, DTRACE_STATE_ALIGN,
14982             NULL, NULL, NULL, NULL, NULL, 0);

14984         ASSERT(MUTEX_HELD(&cpu_lock));
14985         dtrace_bymod = dtrace_hash_create(offsetof(dtrace_probe_t, dtpr_mod),
14986             offsetof(dtrace_probe_t, dtpr_nextmod),
14987             offsetof(dtrace_probe_t, dtpr_prevmod));

14989         dtrace_byfunc = dtrace_hash_create(offsetof(dtrace_probe_t, dtpr_func),
14990             offsetof(dtrace_probe_t, dtpr_nextfunc),
14991             offsetof(dtrace_probe_t, dtpr_prevfunc));

14993         dtrace_byname = dtrace_hash_create(offsetof(dtrace_probe_t, dtpr_name),
14994             offsetof(dtrace_probe_t, dtpr_nextname),
14995             offsetof(dtrace_probe_t, dtpr_prevname));

14997         if (dtrace_retain_max < 1) {
14998                 cmn_err(CE_WARN, "illegal value (%lu) for dtrace_retain_max; "
14999                     "setting to 1", dtrace_retain_max);
15000                 dtrace_retain_max = 1;
15001         }

15003         /*
15004          * Now discover our toxic ranges.
15005          */
15006         dtrace_toxic_ranges(dtrace_toxrange_add);

15008         /*
15009          * Before we register ourselves as a provider to our own framework,
15010          * we would like to assert that dtrace_provider is NULL -- but that's
15011          * not true if we were loaded as a dependency of a DTrace provider.
15012          * Once we've registered, we can assert that dtrace_provider is our
15013          * pseudo provider.
15014          */
15015         (void) dtrace_register("dtrace", &dtrace_provider_attr,
15016             DTRACE_PRIV_NONE, 0, &dtrace_provider_ops, NULL, &id);

15018         ASSERT(dtrace_provider != NULL);
15019         ASSERT((dtrace_provider_id_t)dtrace_provider == id);

15021         dtrace_probeid_begin = dtrace_probe_create((dtrace_provider_id_t)
15022             dtrace_provider, NULL, NULL, "BEGIN", 0, NULL);
15023         dtrace_probeid_end = dtrace_probe_create((dtrace_provider_id_t)
15024             dtrace_provider, NULL, NULL, "END", 0, NULL);
15025         dtrace_probeid_error = dtrace_probe_create((dtrace_provider_id_t)
15026             dtrace_provider, NULL, NULL, "ERROR", 1, NULL);

15028         dtrace_anon_property();
15029         mutex_exit(&cpu_lock);

15031         /*
15032          * If DTrace helper tracing is enabled, we need to allocate the
15033          * trace buffer and initialize the values.
15034          */
```

```
15035         if (dtrace_helptrace_enabled) {
15036                 ASSERT(dtrace_helptrace_buffer == NULL);
15037                 dtrace_helptrace_buffer =
15038                     kmem_zalloc(dtrace_helptrace_bufsize, KM_SLEEP);
15039                 dtrace_helptrace_next = 0;
15040         }

15042         /*
15043          * If there are already providers, we must ask them to provide their
15044          * probes, and then match any anonymous enabling against them.  Note
15045          * that there should be no other retained enablings at this time:
15046          * the only retained enablings at this time should be the anonymous
15047          * enabling.
15048          */
15049         if (dtrace_anon.dta_enabling != NULL) {
15050                 ASSERT(dtrace_retained == dtrace_anon.dta_enabling);

15052                 dtrace_enabling_provide(NULL);
15053                 state = dtrace_anon.dta_state;

15055                 /*
15056                  * We couldn't hold cpu_lock across the above call to
15057                  * dtrace_enabling_provide(), but we must hold it to actually
15058                  * enable the probes.  We have to drop all of our locks, pick
15059                  * up cpu_lock, and regain our locks before matching the
15060                  * retained anonymous enabling.
15061                  */
15062                 mutex_exit(&dtrace_lock);
15063                 mutex_exit(&dtrace_provider_lock);

15065                 mutex_enter(&cpu_lock);
15066                 mutex_enter(&dtrace_provider_lock);
15067                 mutex_enter(&dtrace_lock);

15069                 if ((enab = dtrace_anon.dta_enabling) != NULL)
15070                         (void) dtrace_enabling_match(enab, NULL);

15072                 mutex_exit(&cpu_lock);
15073         }

15075         mutex_exit(&dtrace_lock);
15076         mutex_exit(&dtrace_provider_lock);

15078         if (state != NULL) {
15079                 /*
15080                  * If we created any anonymous state, set it going now.
15081                  */
15082                 (void) dtrace_state_go(state, &dtrace_anon.dta_beganon);
15083         }

15085         return (DDI_SUCCESS);
15086 }
```

*unchanged_portion_omitted_*

```
16005 /*ARGSUSED*/
16006 static int
16007 dtrace_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
16008 {
16009         dtrace_state_t *state;

16011         switch (cmd) {
16012         case DDI_DETACH:
16013                 break;

16015         case DDI_SUSPEND:
16016                 return (DDI_SUCCESS);
```



```
16018         default:
16019                 return (DDI_FAILURE);
16020         }

16022         mutex_enter(&cpu_lock);
16023         mutex_enter(&dtrace_provider_lock);
16024         mutex_enter(&dtrace_lock);

16026         ASSERT(dtrace_opens == 0);

16028         if (dtrace_helpers > 0) {
16029                 mutex_exit(&dtrace_provider_lock);
16030                 mutex_exit(&dtrace_lock);
16031                 mutex_exit(&cpu_lock);
16032                 return (DDI_FAILURE);
16033         }

16035         if (dtrace_unregister((dtrace_provider_id_t)dtrace_provider) != 0) {
16036                 mutex_exit(&dtrace_provider_lock);
16037                 mutex_exit(&dtrace_lock);
16038                 mutex_exit(&cpu_lock);
16039                 return (DDI_FAILURE);
16040         }

16042         dtrace_provider = NULL;

16044         if ((state = dtrace_anon_grab()) != NULL) {
16045                 /*
16046                  * If there were ECBs on this state, the provider should
16047                  * have not been allowed to detach; assert that there is
16048                  * none.
16049                  */
16050                 ASSERT(state->dts_necbs == 0);
16051                 dtrace_state_destroy(state);

16053                 /*
16054                  * If we're being detached with anonymous state, we need to
16055                  * indicate to the kernel debugger that DTrace is now inactive.
16056                  */
16057                 (void) kdi_dtrace_set(KDI_DTSET_DTRACE_DEACTIVATE);
16058         }

16060         bzero(&dtrace_anon, sizeof (dtrace_anon_t));
16061         unregister_cpu_setup_func((cpu_setup_func_t *)dtrace_cpu_setup, NULL);
16062         dtrace_cpu_init = NULL;
16063         dtrace_helpers_cleanup = NULL;
16064         dtrace_helpers_fork = NULL;
16065         dtrace_cpustart_init = NULL;
16066         dtrace_cpustart_fini = NULL;
16067         dtrace_debugger_init = NULL;
16068         dtrace_debugger_fini = NULL;
16069         dtrace_modload = NULL;
16070         dtrace_modunload = NULL;

16072         ASSERT(dtrace_getf == 0);
16073         ASSERT(dtrace_closef == NULL);

16075         mutex_exit(&cpu_lock);

16077         if (dtrace_helptrace_enabled) {
16078                 kmem_free(dtrace_helptrace_buffer, dtrace_helptrace_bufsize);
16079                 dtrace_helptrace_buffer = NULL;
16080         }

16082         kmem_free(dtrace_probes, dtrace_nprobes * sizeof (dtrace_probe_t *));
```

```
16083         dtrace_probes = NULL;
16084         dtrace_nprobes = 0;

16086         dtrace_hash_destroy(dtrace_bymod);
16087         dtrace_hash_destroy(dtrace_byfunc);
16088         dtrace_hash_destroy(dtrace_byname);
16089         dtrace_bymod = NULL;
16090         dtrace_byfunc = NULL;
16091         dtrace_byname = NULL;

16093         kmem_cache_destroy(dtrace_state_cache);
16094         vmem_destroy(dtrace_minor);
16095         vmem_destroy(dtrace_arena);

16097         if (dtrace_toxrange != NULL) {
16098                 kmem_free(dtrace_toxrange,
16099                     dtrace_toxranges_max * sizeof (dtrace_toxrange_t));
16100                 dtrace_toxrange = NULL;
16101                 dtrace_toxranges = 0;
16102                 dtrace_toxranges_max = 0;
16103         }

16105         ddi_remove_minor_node(dtrace_devi, NULL);
16106         dtrace_devi = NULL;

16108         ddi_soft_state_fini(&dtrace_softstate);

16110         ASSERT(dtrace_vtime_references == 0);
16111         ASSERT(dtrace_opens == 0);
16112         ASSERT(dtrace_retained == NULL);

16114         mutex_exit(&dtrace_lock);
16115         mutex_exit(&dtrace_provider_lock);

16117         /*
16118          * We don't destroy the task queue until after we have dropped our
16119          * locks (taskq_destroy() may block on running tasks).  To prevent
16120          * attempting to do work after we have effectively detached but before
16121          * the task queue has been destroyed, all tasks dispatched via the
16122          * task queue must check that DTrace is still attached before
16123          * performing any operation.
16124          */
16125         taskq_destroy(dtrace_taskq);
16126         dtrace_taskq = NULL;

16128         return (DDI_SUCCESS);
16129 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    9558 Fri Jun 22 23:48:25 2012
new/usr/src/uts/common/os/dtrace_subr.c
2916 DTrace in a zone should be able to access fds[]
**********************************************************
```

```
/*
 * 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);

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;
_____unchanged_portion_omitted_
```

```
**********************************************************
   46744 Fri Jun 22 23:48:25 2012
new/usr/src/uts/common/os/fio.c
2916 DTrace in a zone should be able to access fds[]
**********************************************************
```

```
/*
 * 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.
 */

/*      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>

#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.
```

```
 128  *
 129  * (T3) The nearest left ancestor (LPARENT) of node N -- that is, the nearest
 130  *      ancestor containing node N in its right child -- is given by clearing
 131  *      the LSB of N.  For example, LPARENT(111) = 110 and LPARENT(110) = 100.
 132  *      Clearing the LSB of nodes 001, 010 or 100 yields zero, reflecting
 133  *      the fact that these are leftmost nodes.  Note that this algorithm
 134  *      automatically skips generations as necessary.  For example, the parent
 135  *      of node 101 is 110, which is a *right* ancestor (not what we want);
 136  *      but its grandparent is 100, which is a left ancestor. Clearing the LSB
 137  *      of 101 gets us to 100 directly, skipping right past the uninteresting
 138  *      generation (110).
 139  *
 140  *      Note that since LPARENT clears the LSB, whereas CSIZE clears all *but*
 141  *      the LSB, we can express LPARENT() nicely in terms of CSIZE():
 142  *
 143  *      LPARENT(N) = N - CSIZE(N)
 144  *
 145  * (T4) The nearest right ancestor (RPARENT) of node N is given by:
 146  *
 147  *      RPARENT(N) = N + CSIZE(N)
 148  *
 149  * (T5) For every interior node, the children differ from their parent by
 150  *      CSIZE(parent) / 2.  In our example, CSIZE(100) / 2 = 2 = 10 binary,
 151  *      and indeed, the children of 100 are 100 +/- 10 = 010 and 110.
 152  *
 153  * Next, we'll need a few two's-complement math tricks.  Suppose a number,
 154  * N, has the following form:
 155  *
 156  *              N = xxxx10...0
 157  *
 158  * That is, the binary representation of N consists of some string of bits,
 159  * then a 1, then all zeroes.  This amounts to nothing more than saying that
 160  * N has a least-significant bit, which is true for any N != 0.  If we look
 161  * at N and N - 1 together, we see that we can combine them in useful ways:
 162  *
 163  *               N = xxxx10...0
 164  *           N - 1 = xxxx01...1
 165  *      ------------------------
 166  *      N & (N - 1) = xxxx000000
 167  *      N | (N - 1) = xxxx111111
 168  *      N ^ (N - 1) =     111111
 169  *
 170  * In particular, this suggests several easy ways to clear all but the LSB,
 171  * which by (T2) is exactly what we need to determine CSIZE(N) = 10...0.
 172  * We'll opt for this formulation:
 173  *
 174  *      (C1) CSIZE(N) = (N - 1) ^ (N | (N - 1))
 175  *
 176  * Similarly, we have an easy way to determine LPARENT(N), which requires
 177  * that we clear the LSB of N:
 178  *
 179  *      (L1) LPARENT(N) = N & (N - 1)
 180  *
 181  * We note in the above relations that (N | (N - 1)) - N = CSIZE(N) - 1.
 182  * When combined with (T4), this yields an easy way to compute RPARENT(N):
 183  *
 184  *      (R1) RPARENT(N) = (N | (N - 1)) + 1
 185  *
 186  * Finally, to accommodate fd 0 we must adjust all of our results by +/-1 to
 187  * move the fd range from [1, 2^n) to [0, 2^n - 1).  This is straightforward,
 188  * so there's no need to belabor the algebra; the revised relations become:
 189  *
 190  *      (C1a) CSIZE(N) = N ^ (N | (N + 1))
 191  *
 192  *      (L1a) LPARENT(N) = (N & (N + 1)) - 1
 193  *
```



```
 194  *      (R1a) RPARENT(N) = N | (N + 1)
 195  *
 196  * This completes the mathematical framework.  We now have all the tools
 197  * we need to implement fd_find() and fd_reserve().
 198  *
 199  * fd_find(fip, minfd) finds the smallest available file descriptor >= minfd.
 200  * It does not actually allocate the descriptor; that's done by fd_reserve().
 201  * fd_find() proceeds in two steps:
 202  *
 203  * (1) Find the leftmost subtree that contains a descriptor >= minfd.
 204  *     We start at the right subtree rooted at minfd.  If this subtree is
 205  *     not full -- if fip->fi_list[minfd].uf_alloc != CSIZE(minfd) -- then
 206  *     step 1 is done.  Otherwise, we know that all fds in this subtree
 207  *     are taken, so we ascend to RPARENT(minfd) using (R1a).  We repeat
 208  *     this process until we either find a candidate subtree or exceed
 209  *     fip->fi_nfiles.  We use (C1a) to compute CSIZE().
 210  *
 211  * (2) Find the smallest fd in the subtree discovered by step 1.
 212  *     Starting at the root of this subtree, we descend to find the
 213  *     smallest available fd.  Since the left children have the smaller
 214  *     fds, we will descend rightward only when the left child is full.
 215  *
 216  *     We begin by comparing the number of allocated fds in the root
 217  *     to the number of allocated fds in its right child; if they differ
 218  *     by exactly CSIZE(child), we know the left subtree is full, so we
 219  *     descend right; that is, the right child becomes the search root.
 220  *     Otherwise we leave the root alone and start following the right
 221  *     child's left children.  As fortune would have it, this is very
 222  *     simple computationally: by (T5), the right child of fd is just
 223  *     fd + size, where size = CSIZE(fd) / 2.  Applying (T5) again,
 224  *     we find that the right child's left child is fd + size - (size / 2) =
 225  *     fd + (size / 2); *its* left child is fd + (size / 2) - (size / 4) =
 226  *     fd + (size / 4), and so on.  In general, fd's right child's
 227  *     leftmost nth descendant is fd + (size >> n).  Thus, to follow
 228  *     the right child's left descendants, we just halve the size in
 229  *     each iteration of the search.
 230  *
 231  *     When we descend leftward, we must keep track of the number of fds
 232  *     that were allocated in all the right subtrees we rejected, so we
 233  *     know how many of the root fd's allocations are in the remaining
 234  *     (as yet unexplored) leftmost part of its right subtree.  When we
 235  *     encounter a fully-allocated left child -- that is, when we find
 236  *     that fip->fi_list[fd].uf_alloc == ralloc + size -- we descend right
 237  *     (as described earlier), resetting ralloc to zero.
 238  *
 239  * fd_reserve(fip, fd, incr) either allocates or frees fd, depending
 240  * on whether incr is 1 or -1.  Starting at fd, fd_reserve() ascends
 241  * the leftmost ancestors (see (T3)) and updates the allocation counts.
 242  * At each step we use (L1a) to compute LPARENT(), the next left ancestor.
 243  *
 244  * flist_minsize() finds the minimal tree that still covers all
 245  * used fds; as long as the allocation count of a root node is zero, we
 246  * don't need that node or its right subtree.
 247  *
 248  * flist_nalloc() counts the number of allocated fds in the tree, by starting
 249  * at the top of the tree and summing the right-subtree allocation counts as
 250  * it descends leftwards.
 251  *
 252  * Note: we assume that flist_grow() will keep fip->fi_nfiles of the form
 253  * 2^n - 1.  This ensures that the fd trees are always full, which saves
 254  * quite a bit of boundary checking.
 255  */
 256 static int
 257 fd_find(uf_info_t *fip, int minfd)
 258 {
 259         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);
}
```

*unchanged_portion_omitted_*

```
/*
 * Internal form of close.  Decrement reference count on file
 * structure.  Decrement reference count on the vnode following
 * removal of the referencing file structure.
 */
int
closef(file_t *fp)
{
        vnode_t *vp;
        int error;
        int count;
        int flag;
        offset_t offset;

        /*
         * audit close of file (may be exit)
         */
        if (AU_AUDITING())
                audit_closef(fp);
        ASSERT(MUTEX_NOT_HELD(&P_FINFO(curproc)->fi_lock));

        mutex_enter(&fp->f_tlock);

        ASSERT(fp->f_count > 0);

        count = fp->f_count--;
        flag = fp->f_flag;
        offset = fp->f_offset;

        vp = fp->f_vnode;

        error = VOP_CLOSE(vp, flag, count, offset, fp->f_cred, NULL);

        if (count > 1) {
                mutex_exit(&fp->f_tlock);
                return (error);
        }
        ASSERT(fp->f_count == 0);
        mutex_exit(&fp->f_tlock);

        /*
         * If DTrace has getf() subroutines active, it will set dtrace_closef
         * to point to code that implements a barrier with respect to probe
         * context.  This must be called before the file_t is freed (and the
         * vnode that it refers to is released) -- but it must be after the
```



```
         * file_t has been removed from the uf_entry_t.  That is, there must
         * be no way for a racing getf() in probe context to yield the fp that
         * we're operating upon.
         */
        if (dtrace_closef != NULL)
                (*dtrace_closef)();

        VN_RELE(vp);
        /*
         * deallocate resources to audit_data
         */
        if (audit_active)
                audit_unfalloc(fp);
        crfree(fp->f_cred);
        kmem_cache_free(file_cache, fp);
        return (error);
}
```

*unchanged_portion_omitted_*

```
**********************************************************
  100472 Fri Jun 22 23:48:26 2012
new/usr/src/uts/common/sys/dtrace.h
2916 DTrace in a zone should be able to access fds[]
**********************************************************
______unchanged_portion_omitted_

  96 /*
  97  * DTrace Intermediate Format (DIF)
  98  *
  99  * The following definitions describe the DTrace Intermediate Format (DIF), a
 100  * a RISC-like instruction set and program encoding used to represent
 101  * predicates and actions that can be bound to DTrace probes.  The constants
 102  * below defining the number of available registers are suggested minimums; the
 103  * compiler should use DTRACEIOC_CONF to dynamically obtain the number of
 104  * registers provided by the current DTrace implementation.
 105  */
 106 #define DIF_VERSION_1   1               /* DIF version 1: Solaris 10 Beta */
 107 #define DIF_VERSION_2   2               /* DIF version 2: Solaris 10 FCS */
 108 #define DIF_VERSION     DIF_VERSION_2   /* latest DIF instruction set version */
 109 #define DIF_DIR_NREGS   8               /* number of DIF integer registers */
 110 #define DIF_DTR_NREGS   8               /* number of DIF tuple registers */

 112 #define DIF_OP_OR       1               /* or   r1, r2, rd */
 113 #define DIF_OP_XOR      2               /* xor  r1, r2, rd */
 114 #define DIF_OP_AND      3               /* and  r1, r2, rd */
 115 #define DIF_OP_SLL      4               /* sll  r1, r2, rd */
 116 #define DIF_OP_SRL      5               /* srl  r1, r2, rd */
 117 #define DIF_OP_SUB      6               /* sub  r1, r2, rd */
 118 #define DIF_OP_ADD      7               /* add  r1, r2, rd */
 119 #define DIF_OP_MUL      8               /* mul  r1, r2, rd */
 120 #define DIF_OP_SDIV     9               /* sdiv r1, r2, rd */
 121 #define DIF_OP_UDIV     10              /* udiv r1, r2, rd */
 122 #define DIF_OP_SREM     11              /* srem r1, r2, rd */
 123 #define DIF_OP_UREM     12              /* urem r1, r2, rd */
 124 #define DIF_OP_NOT      13              /* not  r1, rd */
 125 #define DIF_OP_MOV      14              /* mov  r1, rd */
 126 #define DIF_OP_CMP      15              /* cmp  r1, r2 */
 127 #define DIF_OP_TST      16              /* tst  r1 */
 128 #define DIF_OP_BA       17              /* ba   label */
 129 #define DIF_OP_BE       18              /* be   label */
 130 #define DIF_OP_BNE      19              /* bne  label */
 131 #define DIF_OP_BG       20              /* bg   label */
 132 #define DIF_OP_BGU      21              /* bgu  label */
 133 #define DIF_OP_BGE      22              /* bge  label */
 134 #define DIF_OP_BGEU     23              /* bgeu label */
 135 #define DIF_OP_BL       24              /* bl   label */
 136 #define DIF_OP_BLU      25              /* blu  label */
 137 #define DIF_OP_BLE      26              /* ble  label */
 138 #define DIF_OP_BLEU     27              /* bleu label */
 139 #define DIF_OP_LDSB     28              /* ldsb [r1], rd */
 140 #define DIF_OP_LDSH     29              /* ldsh [r1], rd */
 141 #define DIF_OP_LDSW     30              /* ldsw [r1], rd */
 142 #define DIF_OP_LDUB     31              /* ldub [r1], rd */
 143 #define DIF_OP_LDUH     32              /* lduh [r1], rd */
 144 #define DIF_OP_LDUW     33              /* lduw [r1], rd */
 145 #define DIF_OP_LDX      34              /* ldx  [r1], rd */
 146 #define DIF_OP_RET      35              /* ret  rd */
 147 #define DIF_OP_NOP      36              /* nop */
 148 #define DIF_OP_SETX     37              /* setx intindex, rd */
 149 #define DIF_OP_SETS     38              /* sets strindex, rd */
 150 #define DIF_OP_SCMP     39              /* scmp r1, r2 */
 151 #define DIF_OP_LDGA     40              /* ldga var, ri, rd */
 152 #define DIF_OP_LDGS     41              /* ldgs var, rd */
 153 #define DIF_OP_STGS     42              /* stgs var, rs */
 154 #define DIF_OP_LDTA     43              /* ldta var, ri, rd */
```



```
 155 #define DIF_OP_LDTS     44              /* ldts var, rd */
 156 #define DIF_OP_STTS     45              /* stts var, rs */
 157 #define DIF_OP_SRA      46              /* sra  r1, r2, rd */
 158 #define DIF_OP_CALL     47              /* call subr, rd */
 159 #define DIF_OP_PUSHTR   48              /* pushtr type, rs, rr */
 160 #define DIF_OP_PUSHTV   49              /* pushtv type, rs, rv */
 161 #define DIF_OP_POPTS    50              /* popts */
 162 #define DIF_OP_FLUSHTS  51              /* flushts */
 163 #define DIF_OP_LDGAA    52              /* ldgaa var, rd */
 164 #define DIF_OP_LDTAA    53              /* ldtaa var, rd */
 165 #define DIF_OP_STGAA    54              /* stgaa var, rs */
 166 #define DIF_OP_STTAA    55              /* sttaa var, rs */
 167 #define DIF_OP_LDLS     56              /* ldls var, rd */
 168 #define DIF_OP_STLS     57              /* stls var, rs */
 169 #define DIF_OP_ALLOCS   58              /* allocs r1, rd */
 170 #define DIF_OP_COPYS    59              /* copys  r1, r2, rd */
 171 #define DIF_OP_STB      60              /* stb  r1, [rd] */
 172 #define DIF_OP_STH      61              /* sth  r1, [rd] */
 173 #define DIF_OP_STW      62              /* stw  r1, [rd] */
 174 #define DIF_OP_STX      63              /* stx  r1, [rd] */
 175 #define DIF_OP_ULDSB    64              /* uldsb [r1], rd */
 176 #define DIF_OP_ULDSH    65              /* uldsh [r1], rd */
 177 #define DIF_OP_ULDSW    66              /* uldsw [r1], rd */
 178 #define DIF_OP_ULDUB    67              /* uldub [r1], rd */
 179 #define DIF_OP_ULDUH    68              /* ulduh [r1], rd */
 180 #define DIF_OP_ULDUW    69              /* ulduw [r1], rd */
 181 #define DIF_OP_ULDX     70              /* uldx  [r1], rd */
 182 #define DIF_OP_RLDSB    71              /* rldsb [r1], rd */
 183 #define DIF_OP_RLDSH    72              /* rldsh [r1], rd */
 184 #define DIF_OP_RLDSW    73              /* rldsw [r1], rd */
 185 #define DIF_OP_RLDUB    74              /* rldub [r1], rd */
 186 #define DIF_OP_RLDUH    75              /* rlduh [r1], rd */
 187 #define DIF_OP_RLDUW    76              /* rlduw [r1], rd */
 188 #define DIF_OP_RLDX     77              /* rldx  [r1], rd */
 189 #define DIF_OP_XLATE    78              /* xlate xlrindex, rd */
 190 #define DIF_OP_XLARG    79              /* xlarg xlrindex, rd */

 192 #define DIF_INTOFF_MAX          0xffff  /* highest integer table offset */
 193 #define DIF_STROFF_MAX          0xffff  /* highest string table offset */
 194 #define DIF_REGISTER_MAX        0xff    /* highest register number */
 195 #define DIF_VARIABLE_MAX        0xffff  /* highest variable identifier */
 196 #define DIF_SUBROUTINE_MAX      0xffff  /* highest subroutine code */

 198 #define DIF_VAR_ARRAY_MIN       0x0000  /* lowest numbered array variable */
 199 #define DIF_VAR_ARRAY_UBASE     0x0080  /* lowest user-defined array */
 200 #define DIF_VAR_ARRAY_MAX       0x00ff  /* highest numbered array variable */

 202 #define DIF_VAR_OTHER_MIN       0x0100  /* lowest numbered scalar or assc */
 203 #define DIF_VAR_OTHER_UBASE     0x0500  /* lowest user-defined scalar or assc */
 204 #define DIF_VAR_OTHER_MAX       0xffff  /* highest numbered scalar or assc */

 206 #define DIF_VAR_ARGS            0x0000  /* arguments array */
 207 #define DIF_VAR_REGS            0x0001  /* registers array */
 208 #define DIF_VAR_UREGS           0x0002  /* user registers array */
 209 #define DIF_VAR_VMREGS          0x0003  /* virtual machine registers array */
 210 #define DIF_VAR_CURTHREAD       0x0100  /* thread pointer */
 211 #define DIF_VAR_TIMESTAMP       0x0101  /* timestamp */
 212 #define DIF_VAR_VTIMESTAMP      0x0102  /* virtual timestamp */
 213 #define DIF_VAR_IPL             0x0103  /* interrupt priority level */
 214 #define DIF_VAR_EPID            0x0104  /* enabled probe ID */
 215 #define DIF_VAR_ID              0x0105  /* probe ID */
 216 #define DIF_VAR_ARG0            0x0106  /* first argument */
 217 #define DIF_VAR_ARG1            0x0107  /* second argument */
 218 #define DIF_VAR_ARG2            0x0108  /* third argument */
 219 #define DIF_VAR_ARG3            0x0109  /* fourth argument */
 220 #define DIF_VAR_ARG4            0x010a  /* fifth argument */
```

```
221 #define DIF_VAR_ARG5            0x010b  /* sixth argument */
222 #define DIF_VAR_ARG6            0x010c  /* seventh argument */
223 #define DIF_VAR_ARG7            0x010d  /* eighth argument */
224 #define DIF_VAR_ARG8            0x010e  /* ninth argument */
225 #define DIF_VAR_ARG9            0x010f  /* tenth argument */
226 #define DIF_VAR_STACKDEPTH      0x0110  /* stack depth */
227 #define DIF_VAR_CALLER          0x0111  /* caller */
228 #define DIF_VAR_PROBEPROV       0x0112  /* probe provider */
229 #define DIF_VAR_PROBEMOD        0x0113  /* probe module */
230 #define DIF_VAR_PROBEFUNC       0x0114  /* probe function */
231 #define DIF_VAR_PROBENAME       0x0115  /* probe name */
232 #define DIF_VAR_PID             0x0116  /* process ID */
233 #define DIF_VAR_TID             0x0117  /* (per-process) thread ID */
234 #define DIF_VAR_EXECNAME        0x0118  /* name of executable */
235 #define DIF_VAR_ZONENAME        0x0119  /* zone name associated with process */
236 #define DIF_VAR_WALLTIMESTAMP   0x011a  /* wall-clock timestamp */
237 #define DIF_VAR_USTACKDEPTH     0x011b  /* user-land stack depth */
238 #define DIF_VAR_UCALLER         0x011c  /* user-level caller */
239 #define DIF_VAR_PPID            0x011d  /* parent process ID */
240 #define DIF_VAR_UID             0x011e  /* process user ID */
241 #define DIF_VAR_GID             0x011f  /* process group ID */
242 #define DIF_VAR_ERRNO           0x0120  /* thread errno */

244 #define DIF_SUBR_RAND                   0
245 #define DIF_SUBR_MUTEX_OWNED            1
246 #define DIF_SUBR_MUTEX_OWNER            2
247 #define DIF_SUBR_MUTEX_TYPE_ADAPTIVE    3
248 #define DIF_SUBR_MUTEX_TYPE_SPIN        4
249 #define DIF_SUBR_RW_READ_HELD           5
250 #define DIF_SUBR_RW_WRITE_HELD          6
251 #define DIF_SUBR_RW_ISWRITER            7
252 #define DIF_SUBR_COPYIN                 8
253 #define DIF_SUBR_COPYINSTR              9
254 #define DIF_SUBR_SPECULATION            10
255 #define DIF_SUBR_PROGENYOF              11
256 #define DIF_SUBR_STRLEN                 12
257 #define DIF_SUBR_COPYOUT                13
258 #define DIF_SUBR_COPYOUTSTR             14
259 #define DIF_SUBR_ALLOCA                 15
260 #define DIF_SUBR_BCOPY                  16
261 #define DIF_SUBR_COPYINTO               17
262 #define DIF_SUBR_MSGDSIZE               18
263 #define DIF_SUBR_MSGSIZE                19
264 #define DIF_SUBR_GETMAJOR               20
265 #define DIF_SUBR_GETMINOR               21
266 #define DIF_SUBR_DDI_PATHNAME           22
267 #define DIF_SUBR_STRJOIN                23
268 #define DIF_SUBR_LLTOSTR                24
269 #define DIF_SUBR_BASENAME               25
270 #define DIF_SUBR_DIRNAME                26
271 #define DIF_SUBR_CLEANPATH              27
272 #define DIF_SUBR_STRCHR                 28
273 #define DIF_SUBR_STRRCHR                29
274 #define DIF_SUBR_STRSTR                 30
275 #define DIF_SUBR_STRTOK                 31
276 #define DIF_SUBR_SUBSTR                 32
277 #define DIF_SUBR_INDEX                  33
278 #define DIF_SUBR_RINDEX                 34
279 #define DIF_SUBR_HTONS                  35
280 #define DIF_SUBR_HTONL                  36
281 #define DIF_SUBR_HTONLL                 37
282 #define DIF_SUBR_NTOHS                  38
283 #define DIF_SUBR_NTOHL                  39
284 #define DIF_SUBR_NTOHLL                 40
285 #define DIF_SUBR_INET_NTOP              41
286 #define DIF_SUBR_INET_NTOA              42
```



```
287 #define DIF_SUBR_INET_NTOA6             43
288 #define DIF_SUBR_TOUPPER                44
289 #define DIF_SUBR_TOLOWER                45
290 #define DIF_SUBR_GETF                   46

292 #define DIF_SUBR_MAX                    46      /* max subroutine value */
291 #define DIF_SUBR_MAX                    45      /* max subroutine value */

294 typedef uint32_t dif_instr_t;

296 #define DIF_INSTR_OP(i)                 (((i) >> 24) & 0xff)
297 #define DIF_INSTR_R1(i)                 (((i) >> 16) & 0xff)
298 #define DIF_INSTR_R2(i)                 (((i) >>  8) & 0xff)
299 #define DIF_INSTR_RD(i)                 ((i) & 0xff)
300 #define DIF_INSTR_RS(i)                 ((i) & 0xff)
301 #define DIF_INSTR_LABEL(i)              ((i) & 0xffffff)
302 #define DIF_INSTR_VAR(i)                (((i) >>  8) & 0xffff)
303 #define DIF_INSTR_INTEGER(i)            (((i) >>  8) & 0xffff)
304 #define DIF_INSTR_STRING(i)             (((i) >>  8) & 0xffff)
305 #define DIF_INSTR_SUBR(i)               (((i) >>  8) & 0xffff)
306 #define DIF_INSTR_TYPE(i)               (((i) >> 16) & 0xff)
307 #define DIF_INSTR_XLREF(i)              (((i) >>  8) & 0xffff)

309 #define DIF_INSTR_FMT(op, r1, r2, d) \
310         (((op) << 24) | ((r1) << 16) | ((r2) << 8) | (d))

312 #define DIF_INSTR_NOT(r1, d)            (DIF_INSTR_FMT(DIF_OP_NOT, r1, 0, d))
313 #define DIF_INSTR_MOV(r1, d)            (DIF_INSTR_FMT(DIF_OP_MOV, r1, 0, d))
314 #define DIF_INSTR_CMP(op, r1, r2)       (DIF_INSTR_FMT(op, r1, r2, 0))
315 #define DIF_INSTR_TST(r1)               (DIF_INSTR_FMT(DIF_OP_TST, r1, 0, 0))
316 #define DIF_INSTR_BRANCH(op, label)     (((op) << 24) | (label))
317 #define DIF_INSTR_LOAD(op, r1, d)       (DIF_INSTR_FMT(op, r1, 0, d))
318 #define DIF_INSTR_STORE(op, r1, d)      (DIF_INSTR_FMT(op, r1, 0, d))
319 #define DIF_INSTR_SETX(i, d)            ((DIF_OP_SETX << 24) | ((i) << 8) | (d))
320 #define DIF_INSTR_SETS(s, d)            ((DIF_OP_SETS << 24) | ((s) << 8) | (d))
321 #define DIF_INSTR_RET(d)                (DIF_INSTR_FMT(DIF_OP_RET, 0, 0, d))
322 #define DIF_INSTR_NOP                   (DIF_OP_NOP << 24)
323 #define DIF_INSTR_LDA(op, v, r, d)      (DIF_INSTR_FMT(op, v, r, d))
324 #define DIF_INSTR_LDV(op, v, d)         (((op) << 24) | ((v) << 8) | (d))
325 #define DIF_INSTR_STV(op, v, rs)        (((op) << 24) | ((v) << 8) | (rs))
326 #define DIF_INSTR_CALL(s, d)            ((DIF_OP_CALL << 24) | ((s) << 8) | (d))
327 #define DIF_INSTR_PUSHTS(op, t, r2, rs) (DIF_INSTR_FMT(op, t, r2, rs))
328 #define DIF_INSTR_POPTS                 (DIF_OP_POPTS << 24)
329 #define DIF_INSTR_FLUSHTS               (DIF_OP_FLUSHTS << 24)
330 #define DIF_INSTR_ALLOCS(r1, d)         (DIF_INSTR_FMT(DIF_OP_ALLOCS, r1, 0, d))
331 #define DIF_INSTR_COPYS(r1, r2, d)      (DIF_INSTR_FMT(DIF_OP_COPYS, r1, r2, d))
332 #define DIF_INSTR_XLATE(op, r, d)       (((op) << 24) | ((r) << 8) | (d))

334 #define DIF_REG_R0      0               /* %r0 is always set to zero */

336 /*
337  * A DTrace Intermediate Format Type (DIF Type) is used to represent the types
338  * of variables, function and associative array arguments, and the return type
339  * for each DIF object (shown below).  It contains a description of the type,
340  * its size in bytes, and a module identifier.
341  */
342 typedef struct dtrace_diftype {
343         uint8_t dtdt_kind;              /* type kind (see below) */
344         uint8_t dtdt_ckind;             /* type kind in CTF */
345         uint8_t dtdt_flags;             /* type flags (see below) */
346         uint8_t dtdt_pad;               /* reserved for future use */
347         uint32_t dtdt_size;             /* type size in bytes (unless string) */
348 } dtrace_diftype_t;
______unchanged_portion_omitted_

2215 extern dtrace_vtime_state_t dtrace_vtime_active;
```

```
2216 extern void dtrace_vtime_switch(kthread_t *next);
2217 extern void dtrace_vtime_enable_tnf(void);
2218 extern void dtrace_vtime_disable_tnf(void);
2219 extern void dtrace_vtime_enable(void);
2220 extern void dtrace_vtime_disable(void);

2222 struct regs;

2224 extern int (*dtrace_pid_probe_ptr)(struct regs *);
2225 extern int (*dtrace_return_probe_ptr)(struct regs *);
2226 extern void (*dtrace_fasttrap_fork_ptr)(proc_t *, proc_t *);
2227 extern void (*dtrace_fasttrap_exec_ptr)(proc_t *);
2228 extern void (*dtrace_fasttrap_exit_ptr)(proc_t *);
2229 extern void dtrace_fasttrap_fork(proc_t *, proc_t *);

2231 typedef uintptr_t dtrace_icookie_t;
2232 typedef void (*dtrace_xcall_t)(void *);

2234 extern dtrace_icookie_t dtrace_interrupt_disable(void);
2235 extern void dtrace_interrupt_enable(dtrace_icookie_t);

2237 extern void dtrace_membar_producer(void);
2238 extern void dtrace_membar_consumer(void);

2240 extern void (*dtrace_cpu_init)(processorid_t);
2241 extern void (*dtrace_modload)(struct modctl *);
2242 extern void (*dtrace_modunload)(struct modctl *);
2243 extern void (*dtrace_helpers_cleanup)();
2244 extern void (*dtrace_helpers_fork)(proc_t *parent, proc_t *child);
2245 extern void (*dtrace_cpustart_init)();
2246 extern void (*dtrace_cpustart_fini)();
2247 extern void (*dtrace_closef)();

2249 extern void (*dtrace_debugger_init)();
2250 extern void (*dtrace_debugger_fini)();
2251 extern dtrace_cacheid_t dtrace_predcache_id;

2253 extern hrtime_t dtrace_gethrtime(void);
2254 extern void dtrace_sync(void);
2255 extern void dtrace_toxic_ranges(void (*)(uintptr_t, uintptr_t));
2256 extern void dtrace_xcall(processorid_t, dtrace_xcall_t, void *);
2257 extern void dtrace_vpanic(const char *, __va_list);
2258 extern void dtrace_panic(const char *, ...);

2260 extern int dtrace_safe_defer_signal(void);
2261 extern void dtrace_safe_synchronous_signal(void);

2263 extern int dtrace_mach_aframes(void);

2265 #if defined(__i386) || defined(__amd64)
2266 extern int dtrace_instr_size(uchar_t *instr);
2267 extern int dtrace_instr_size_isa(uchar_t *, model_t, int *);
2268 extern void dtrace_invop_add(int (*)(uintptr_t, uintptr_t *, uintptr_t));
2269 extern void dtrace_invop_remove(int (*)(uintptr_t, uintptr_t *, uintptr_t));
2270 extern void dtrace_invop_callsite(void);
2271 #endif

2273 #ifdef __sparc
2274 extern int dtrace_blksuword32(uintptr_t, uint32_t *, int);
2275 extern void dtrace_getfsr(uint64_t *);
2276 #endif

2278 #define DTRACE_CPUFLAG_ISSET(flag) \
2279         (cpu_core[CPU->cpu_id].cpuc_dtrace_flags & (flag))

2281 #define DTRACE_CPUFLAG_SET(flag) \
```



```
2282         (cpu_core[CPU->cpu_id].cpuc_dtrace_flags |= (flag))

2284 #define DTRACE_CPUFLAG_CLEAR(flag) \
2285         (cpu_core[CPU->cpu_id].cpuc_dtrace_flags &= ~(flag))

2287 #endif /* _KERNEL */

2289 #endif  /* _ASM */

2291 #if defined(__i386) || defined(__amd64)

2293 #define DTRACE_INVOP_PUSHL_EBP          1
2294 #define DTRACE_INVOP_POPL_EBP           2
2295 #define DTRACE_INVOP_LEAVE              3
2296 #define DTRACE_INVOP_NOP                4
2297 #define DTRACE_INVOP_RET                5

2299 #endif

2301 #ifdef  __cplusplus
2302 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   64188 Fri Jun 22 23:48:26 2012
new/usr/src/uts/common/sys/dtrace_impl.h
2916 DTrace in a zone should be able to access fds[]
**********************************************************
______unchanged_portion_omitted_

 880 /*
 881  * DTrace Machine State
 882  *
 883  * In the process of processing a fired probe, DTrace needs to track and/or
 884  * cache some per-CPU state associated with that particular firing.  This is
 885  * state that is always discarded after the probe firing has completed, and
 886  * much of it is not specific to any DTrace consumer, remaining valid across
 887  * all ECBs.  This state is tracked in the dtrace_mstate structure.
 888  */
 889 #define DTRACE_MSTATE_ARGS              0x00000001
 890 #define DTRACE_MSTATE_PROBE             0x00000002
 891 #define DTRACE_MSTATE_EPID              0x00000004
 892 #define DTRACE_MSTATE_TIMESTAMP         0x00000008
 893 #define DTRACE_MSTATE_STACKDEPTH        0x00000010
 894 #define DTRACE_MSTATE_CALLER            0x00000020
 895 #define DTRACE_MSTATE_IPL               0x00000040
 896 #define DTRACE_MSTATE_FLTOFFS           0x00000080
 897 #define DTRACE_MSTATE_WALLTIMESTAMP     0x00000100
 898 #define DTRACE_MSTATE_USTACKDEPTH       0x00000200
 899 #define DTRACE_MSTATE_UCALLER           0x00000400

 901 typedef struct dtrace_mstate {
 902         uintptr_t dtms_scratch_base;            /* base of scratch space */
 903         uintptr_t dtms_scratch_ptr;             /* current scratch pointer */
 904         size_t dtms_scratch_size;               /* scratch size */
 905         uint32_t dtms_present;                  /* variables that are present */
 906         uint64_t dtms_arg[5];                   /* cached arguments */
 907         dtrace_epid_t dtms_epid;                /* current EPID */
 908         uint64_t dtms_timestamp;                /* cached timestamp */
 909         hrtime_t dtms_walltimestamp;            /* cached wall timestamp */
 910         int dtms_stackdepth;                    /* cached stackdepth */
 911         int dtms_ustackdepth;                   /* cached ustackdepth */
 912         struct dtrace_probe *dtms_probe;        /* current probe */
 913         uintptr_t dtms_caller;                  /* cached caller */
 914         uint64_t dtms_ucaller;                  /* cached user-level caller */
 915         int dtms_ipl;                           /* cached interrupt pri lev */
 916         int dtms_fltoffs;                       /* faulting DIFO offset */
 917         uintptr_t dtms_strtok;                  /* saved strtok() pointer */
 918         uint32_t dtms_access;                   /* memory access rights */
 919         dtrace_difo_t *dtms_difo;               /* current dif object */
 920         file_t *dtms_getf;                      /* cached rval of getf() */
 921 } dtrace_mstate_t;
______unchanged_portion_omitted_

1098 /*
1099  * DTrace Consumer State
1100  *
1101  * Each DTrace consumer has an associated dtrace_state structure that contains
1102  * its in-kernel DTrace state -- including options, credentials, statistics and
1103  * pointers to ECBs, buffers, speculations and formats.  A dtrace_state
1104  * structure is also allocated for anonymous enablings.  When anonymous state
1105  * is grabbed, the grabbing consumers dts_anon pointer is set to the grabbed
1106  * dtrace_state structure.
1107  */
1108 struct dtrace_state {
1109         dev_t dts_dev;                          /* device */
1110         int dts_necbs;                          /* total number of ECBs */
1111         dtrace_ecb_t **dts_ecbs;                /* array of ECBs */
1112         dtrace_epid_t dts_epid;                 /* next EPID to allocate */
```



```
1113         size_t dts_needed;                      /* greatest needed space */
1114         struct dtrace_state *dts_anon;          /* anon. state, if grabbed */
1115         dtrace_activity_t dts_activity;         /* current activity */
1116         dtrace_vstate_t dts_vstate;             /* variable state */
1117         dtrace_buffer_t *dts_buffer;            /* principal buffer */
1118         dtrace_buffer_t *dts_aggbuffer;         /* aggregation buffer */
1119         dtrace_speculation_t *dts_speculations; /* speculation array */
1120         int dts_nspeculations;                  /* number of speculations */
1121         int dts_naggregations;                  /* number of aggregations */
1122         dtrace_aggregation_t **dts_aggregations; /* aggregation array */
1123         vmem_t *dts_aggid_arena;                /* arena for aggregation IDs */
1124         uint64_t dts_errors;                    /* total number of errors */
1125         uint32_t dts_speculations_busy;         /* number of spec. busy */
1126         uint32_t dts_speculations_unavail;      /* number of spec unavail */
1127         uint32_t dts_stkstroverflows;           /* stack string tab overflows */
1128         uint32_t dts_dblerrors;                 /* errors in ERROR probes */
1129         uint32_t dts_reserve;                   /* space reserved for END */
1130         hrtime_t dts_laststatus;                /* time of last status */
1131         cyclic_id_t dts_cleaner;                /* cleaning cyclic */
1132         cyclic_id_t dts_deadman;                /* deadman cyclic */
1133         hrtime_t dts_alive;                     /* time last alive */
1134         char dts_speculates;                    /* boolean: has speculations */
1135         char dts_destructive;                   /* boolean: has dest. actions */
1136         int dts_nformats;                       /* number of formats */
1137         char **dts_formats;                     /* format string array */
1138         dtrace_optval_t dts_options[DTRACEOPT_MAX]; /* options */
1139         dtrace_cred_t dts_cred;                 /* credentials */
1140         size_t dts_nretained;                   /* number of retained enabs */
1141         int dts_getf;                           /* number of getf() calls */
1142 };
______unchanged_portion_omitted_
```

```
**********************************************************
   24056 Fri Jun 22 23:48:27 2012
new/usr/src/uts/common/sys/zone.h
2916 DTrace in a zone should be able to access fds[]
**********************************************************
______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 */
 493         /*
 494          * The following two can be read without holding any locks.  They are
 495          * updated under cpu_lock.
 496          */
 497         int             zone_ncpus;  /* zone's idea of ncpus */
 498         int             zone_ncpus_online; /* zone's idea of ncpus_online */
 499         /*
 500          * List of ZFS datasets exported to this zone.
 501          */
 502         list_t          zone_datasets;  /* list of datasets */
```

```
504         ts_label_t      *zone_slabel;   /* zone sensitivity label */
505         int             zone_match;     /* require label match for packets */
506         tsol_mlp_list_t zone_mlps;      /* MLPs on zone-private addresses */

508         boolean_t       zone_restart_init;      /* Restart init if it dies? */
509         struct brand    *zone_brand;            /* zone's brand */
510         void            *zone_brand_data;       /* store brand specific data */
511         id_t            zone_defaultcid;        /* dflt scheduling class id */
512         kstat_t         *zone_swapresv_kstat;
513         kstat_t         *zone_lockedmem_kstat;
514         /*
515          * zone_dl_list is protected by zone_lock
516          */
517         list_t          zone_dl_list;
518         netstack_t      *zone_netstack;
519         struct cpucap   *zone_cpucap;   /* CPU caps data */
520         /*
521          * Solaris Auditing per-zone audit context
522          */
523         struct au_kcontext      *zone_audit_kctxt;
524         /*
525          * For private use by mntfs.
526          */
527         struct mntelem  *zone_mntfs_db;
528         krwlock_t       zone_mntfs_db_lock;

530         struct klpd_reg         *zone_pfexecd;

532         char            *zone_fs_allowed;
533         rctl_qty_t      zone_nprocs;    /* number of processes in the zone */
534         rctl_qty_t      zone_nprocs_ctl;        /* current limit protected by */
535                                                 /* zone_rctls->rcs_lock */
536         kstat_t         *zone_nprocs_kstat;

538         /*
539          * DTrace-private per-zone state
540          */
541         int             zone_dtrace_getf;       /* # of unprivileged getf()s */
542 } zone_t;
```

*unchanged_portion_omitted_*